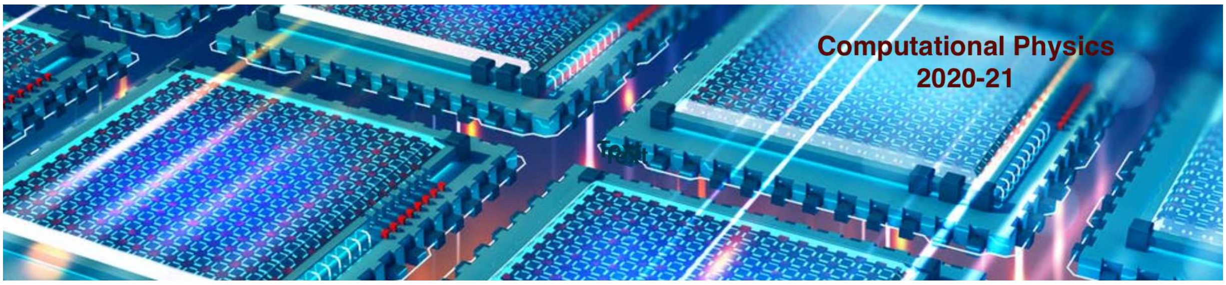
Computational Physics

numerical methods with C++ (and UNIX)

2020-21



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

LU decomposition

- ✔ Any square matrix $\mathbf{A}$ can be expressed as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$

  $\boxed{\mathbf{A} = \mathbf{L}\ \mathbf{U}}$

  - ☞ *the computation of $\mathbf{L}$ and $\mathbf{U}$ is known as LU decomposition or $\mathbf{LU}$ factorization*
  - ☞ *the factorization is not unique unless constraints on L and U are applied*

- ✔ common decompositions:

Decomposition	Constraints
Doolittle	$L_{ii} = 1$ with $i = 1, 2, ..., n$
Crout	$U_{ii} = 1$ with $i = 1, 2, ..., n$
Choleski	$\mathbf{L} = \mathbf{U}^{\mathbf{T}}$

After decomposing $\mathbf{A}$:

$\mathbf{Ax} = \mathbf{b} \Rightarrow \mathbf{LUx} = \mathbf{b}$

We have:

$\mathbf{Ly} = \mathbf{b}$ with $(\mathbf{Ux} = \mathbf{y})$

Therefore: we start getting $\mathbf{y}$ and then $\mathbf{x}$

Doolittle decomposition

✔ Consider a 3×3 $\mathbf{A}$ matrix and the respective triangular lower and upper matrices $\mathbf{L}$ and $\mathbf{U}$

$$[\mathbf{A}] = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad [\mathbf{L}] = \begin{pmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{pmatrix} \quad [\mathbf{U}] = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

✔ Making the operation: $\mathbf{A} = \mathbf{LU}$

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

Doolittle decomposition (cont.)

✔ Applying Gauss elimination: eliminating elements below pivot $(LU)_{11}$

(Row$_2$ − L_{21}Row$_1$ → Row$_2$) to eliminate (LU)$_{21}$

(Row$_3$ − L_{31}Row$_1$ → Row$_3$) to eliminate (LU)$_{31}$

$$[\mathbf{A}'] = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & U_{22}L_{32} & U_{23}L_{32} + U_{33} \end{pmatrix}$$

✔ Applying Gauss elimination: eliminating element below pivot $(LU)_{22}$

(Row$_3$ − L_{32} Row$_2$ → Row$_3$) to eliminate (LU)$_{32}$

$$[\mathbf{A}''] = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

Gauss elimination method provided us with $\mathbf{U}$ and $\mathbf{L}$ matrices!

Doolittle decomposition (cont.)

- ✔ The matrix **U** is the one that results from the Gauss elimination
- ✔ The off-diagonal elements of matrix **L** correspond to the multipliers used during Gauss elimination
- ✔ It is current pratice to store in a matrix both the upper triangular matrix and the lower triangular matrix

  the diagonal elements of the **L** matrix are not stored...

$$[\mathbf{L} \setminus \mathbf{U}] = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{pmatrix}$$

LUdecomp algorithm

```
// matrix A(n x n)
// using indices: 0, 1, ..., n-1

// Gauss elimination

loop on pivot row (k): k = 0, n-2

  loop on rows below pivot:
                     i = k+1, n-1

     - for every row:
       compute multiplier
                     A(i,k)/A(k,k)

     - transform row i:
       only elements A(i, k+1:n-1)
                      are stored

     - store mutipliers on A(i,k)

// solution now...
```


Doolittle: solution (forward subst)

- ✔ We have to solve the system $\mathbf{Ly} = \mathbf{b}$ by forward substitution

$$\begin{pmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

- ✔ forward substitution:

$$\begin{pmatrix} y_1 & = b_1 \\ L_{21}y_1 + y_2 & = b_2 \\ L_{k1}y_1 + L_{k2}y_2 + \cdots + L_{k,k-1}y_{k-1} + y_k & = b_k \end{pmatrix}$$

The solution of the equation for a generic **k row**:

$$\boxed{y_k = b_k - \sum_{j=1}^{k-1} L_{kj} y_j \qquad (k = 2, 3, ...n(\text{rows}))}$$

LUsolver algorithm

```
// forward solution (Ly=b)

//loop on rows
for (int k=0; k<n; k++) {
  double sumC = 0.;
  for (int i=0; i<k; i++) {
    sumC += y[i]*A[k][i];
  }
  y[k] = b[k] - sumC;
}

// backward solution (Ux=y)

//loop on rows
for (int k=n-1; k>=0; k--) {
  double sumC = 0.;
  for (int i=k+1; i<n; i++) {
    sumC += x[i]*A[k][i];
  }
  x[k] = (y[k] - sumC)/A[k][k];
}
```

Doolittle: solution (backward subst)

✔ We have to solve the system $\mathbf{Ux} = \mathbf{y}$ by backward substitution

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

LUsolver algorithm

```
// backward solution (Ux=y)

//loop on rows
for (int k=n-1; k>=0; k--) {
  double sumC = 0.;
  for (int i=k+1; i<n; i++) {
    sumC += x[i]*A[k][i];
  }
  x[k] = (y[k] - sumC)/A[k][k];
}
```

✔ backward substitution:

$$\begin{pmatrix} U_{33}x_3 & = y_3 \\ U_{22}x_2 + U_{23}x_3 & = y_2 \\ U_{11}x_1 + U_{12}x_2 + U_{1,3}x_3 & = y_1 \end{pmatrix}$$

The solution of the equation for a generic **k row**:

$$\boxed{x_k = \frac{1}{U_{kk}}\left(y_k - \sum_{j=k+1}^{n} A_{kj}x_j\right)}$$

Doolittle decomp: example

Solve the following system using Doolittle decomposition

$$[\mathbf{A}] = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 6 & -1 \\ 2 & -1 & 2 \end{pmatrix} \qquad [\mathbf{b}] = \begin{pmatrix} 7 \\ 13 \\ 5 \end{pmatrix}$$

Choleski decomposition

- ✔ This method uses the decomposition: $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$
- ✔ The nature of the decomposition ($\mathbf{L}\mathbf{L}^{\mathbf{T}}$) requires a symmetric $\mathbf{A}$ matrix
- ✔ It envolves the using of square root function
  - ☞ to avoid square roots of negative numbers the matrix must be *positive definite* $\Rightarrow \mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x} > \mathbf{0}$

$$[\mathbf{A}] = LL^T = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11}^2 & L_{11}L_{21} & L11L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix}$$

Choleski decomposition (cont.)

- ✔ Symmetric matrix $\Rightarrow$ $n!$ equations to solve ($n = 3 \Rightarrow$ 6eqs)

$$\begin{aligned} L_{11} &= \sqrt{A_{11}} \\ L_{21} &= A_{21}/L_{11} \\ L_{31} &= A_{31}/L_{11} \\ L_{22} &= \sqrt{A_{22} - L_{21}^2} \\ L_{32} &= (A_{32} - L_{21}L_{31})/L_{22} \\ L_{33} &= \sqrt{A_{33} - L_{31}^2 L_{32}^2} \end{aligned}$$

Matrix inversion

✔ To invert the matrix $\mathbf{A}$ we have to solve the equation:

$$\mathbf{AX} = \mathbf{I} \Rightarrow \mathbf{A}^{-1}\mathbf{AX} = \mathbf{A}^{-1}\mathbf{I} \Rightarrow \mathbf{X} = \mathbf{A}^{-1}$$

$\mathbf{I} \equiv$ is the identity matrix
$\mathbf{X} \equiv$ is the inverse of $\mathbf{A}$

✔ For inverting $\mathbf{M}$ we have to solve:

$\mathbf{A}\mathbf{x_i} = \mathbf{b_i} \qquad \mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots\mathbf{n}$

$\mathbf{b_i}$ = ith column of I

$\mathbf{x_i}$ = ith column of $\mathbf{A}^{-1}$

Sparse matrices

✔ A matrix is typically stored as a two-dimensional array or a set of vectors **Vec**, as defined in our FC course

✔ Many problems present matrices with a lot of zero's on its contents

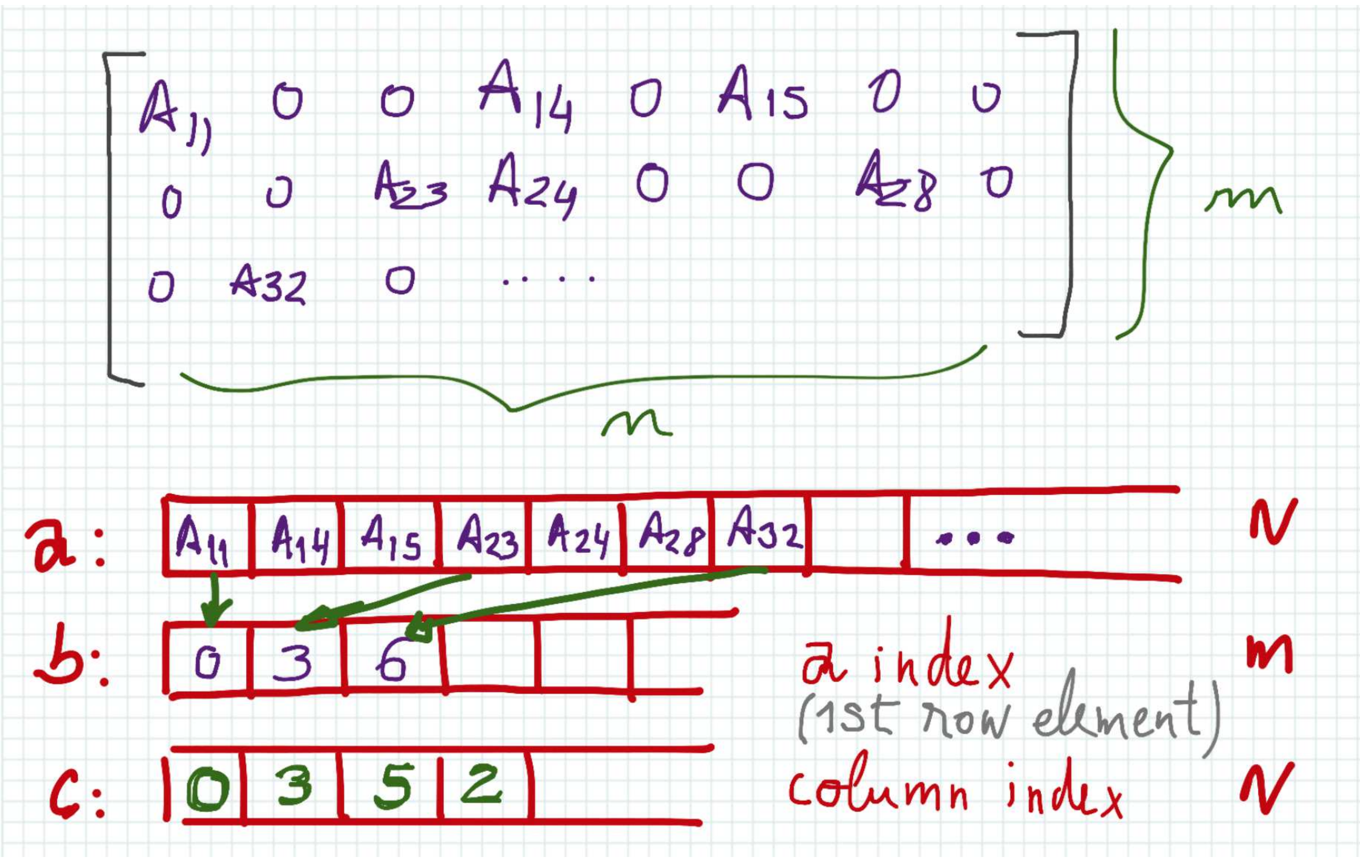
✔ Storing all members of the matrix implies a lot of of useless contents being stored in memory

✔ Many algorithms propose an optimized way of storing the matrix members

✔ Yale algorithm for storing a sparse matrix of $\mathbf{m} \times \mathbf{n}$:
uses **three arrays or Vec's** to store the N non-zero coefficients of the matrix

- **Vec a** is of length $\mathbf{N}$ and holds all the nonzero entries of matrix $\mathbf{M}$ in left-to-right top-to-bottom order.
- **Vec b** is of length $\mathbf{m}$ and contains the index in vector $\mathbf{a}$ of the first element in each row.
- **Vec c** array, contains the column index in $\mathbf{M}$ of each element of vector $\mathbf{a}$ and hence is of length $\mathbf{N}$ as well.

sparse coding



Sparse matrices storing example

$$M(4\times 6) = \begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

storage arrays:

$\mathbf{a} = (10, 20, 30, 40, 50, 60, 70, 80)$ (8 values)

$\mathbf{b} = (0, 2, 4, 7)$ (4 values)

$\mathbf{c} = (0, 1, 1, 3, 2, 3, 4, 5)$ (8 values)

note: I assume we use Vec class because the number of elements information is stored inside

Sparse matrices decoding

- ✔ **Vec a**: contains all the nonzero entries of matrix $\mathbf{M}$
- ✔ **Vec b**: contains the index in vector $\mathbf{a}$ of every first row element
- ✔ **Vec c**: contains the matrix column index of every non-null matrix element

how to get full matrix?

- ✔ loop on matrix rows
  number of rows obtained from size of **array b**
- ✔ loop on matrix row elements
  we know matrix elements from **array a** and the ones belonging to a same row from **array b**

```
vector<Vec> m;
// loop on matrix rows
for (int i=0; i<b.size(); i++) {
  Vec row(b.size()); // zeros

  // loop on matrix row elements
  for (int j=b[i]; j<b[i+1]; j++) {
     k = c[j]; // column index
     row[k] = a[k];
  }
  m.push_back(row);
}
// print matrix
m.Print();
```


Sparse matrices: full row of zero's

how to store the sparse matrix?

looking to the previous slide and sparse decoding, we need

- ✔ keep array b with the right number of rows
- ✔ to keep row filled with zero's, we cannot enter 2nd loop
  row with zero's: $\Rightarrow\ b[i] = b[i+1]$

$$M(4\times 6) = \begin{pmatrix} 10 & 20 & 0 & 0 & 25 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

storage arrays:

$\mathbf{a} = (10, 20, 25, 30, 40, 80)$ (6 values)

$\mathbf{b} = (0, 3, 5, 5, 6)$ (4+1 values)

$\mathbf{c} = (0, 1, 4, 1, 3, 5)$ (6 values)

Banded matrices

- ✔ In case a matrix present its non-zero members all grouped around the main diagonal, it is said to be of the **banded** type (common to scientific problems)
  - ☞ a **tridiagonal matrix** presents a **bandwidth=3**, i.e., at most three nonzero elements in each row (or column)
  - ☞ some of the elements in the populated diagonals can be zero (of course!)

$$[\mathbf{A}] = \begin{pmatrix} A_{11} & A_{12} & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & A_{43} & A_{55} \end{pmatrix}$$

- ✔ The banded structure of a coefficient matrix can be exploited to save storage space and computation time

Banded matrices: LU decomposition

- ✔ Let's use the Doolittle scheme to decompose the tridiagonal matrix $\mathbf{A}$

$$[\mathbf{A}] = \begin{array}{c c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & \cdots & n \end{array} & \\ \begin{pmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_2 & b_3 & c_3 & \cdots & 0 \\ 0 & 0 & a_3 & b_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ \cdots \\ n \end{array} \end{array}$$

- ✔ To reduce the row $\mathbf{k}$, i.e., to eliminate the $\mathbf{a_{k-1}}$ element we do, pivot:$\mathbf{Row_{k-1}}$

$$\mathbf{Row_k} - \mathbf{Row_{k-1}} \times \left(\frac{\mathbf{a_{k-1}}}{\mathbf{b_{k-1}}}\right) \rightarrow \mathbf{Row_k}$$
$$\mathbf{k} = \mathbf{2}, \mathbf{3}, \cdots, \mathbf{n}$$

- ✔ In the decomposition process, the reduced $\mathbf{a_i}$ elements are replaced by the multipliers $\left(\frac{\mathbf{a_{k-1}}}{\mathbf{b_{k-1}}}\right)$

$$\mathbf{a_{k-1}} = \left(\frac{\mathbf{a_{k-1}}}{\mathbf{b_{k-1}}}\right)$$
$$\mathbf{b_k} = \mathbf{b_k} - \left(\frac{\mathbf{a_{k-1}}}{\mathbf{b_{k-1}}}\right) \times \mathbf{c_{k-1}}$$
$\mathbf{c_k}$ = not affected

```
The vectors to store are:

a = [ a_1, a_2, ..., a_{n-1} ]

b = [ b_1, b_2, ..., b_{n}} ]

c = [ c_1, c_2, ..., c_{n-1} ]
```

Banded matrices: LU solution

✔ Now we have to solve the equation $\mathbf{Ax} = \mathbf{d}$, there are two equations to solve:

1) $\mathbf{Ly} = \mathbf{d}$

2) $\mathbf{Ux} = \mathbf{y}$

by respectively forward and back substitution

$$[\mathbf{L}|\mathbf{d}] = \left(\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & d_1 \\ a_1 & 1 & 0 & 0 & \cdots & 0 & d_2 \\ 0 & a_2 & 1 & 0 & \cdots & 0 & d_3 \\ 0 & 0 & a_3 & 1 & \cdots & 0 & d_4 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & 0 & a_{n-1} & 1 & d_n \end{array}\right) \quad [\mathbf{U}|\mathbf{y}] = \left(\begin{array}{cccccc|c} b_1 & c_1 & 0 & \cdots & 0 & 0 & y_1 \\ 0 & b_2 & c_2 & \cdots & 0 & 0 & y_2 \\ 0 & 0 & b_3 & \cdots & 0 & 0 & y_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & b_{n-1} & c_{n-1} & y_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & b_n & y_n \end{array}\right)$$

Iterative methods

✔ In iterative methods, we start with an initial guess for the solution $\mathbf{x}$ and then we iterate over solutions until changes are negligible

✔ The convergence of the iterative methods is only guaranteed if the coefficient matrix is diagonally dominant

- The number of iterations depend on the initial guess
- Convergence will be attained independently of the initial guess

Jacobi method

- Let's write the equation $\mathbf{Ax} = \mathbf{b}$ in scalar notation:

$$\sum_{j=1}^{n} A_{ij}\, x_j = b_i \qquad (i = 1, 2, \cdots, n)$$

- Extracting the term containing $\mathbf{x_i}$:

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

- at every iteration k: $\boxed{x_i^{(k+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1\\(i\neq j)}}^{n} A_{ij}\, x_j^{(k)}\right)}$

At every iteration $(n-1)$ multiplications are done

EqSolver class

```
class EqSolver {

public:
 EqSolver();
 EqSolver(const FCmatrix&, const Vec&); // matriz M e vector de constantes B
 // set
 void SetConstants(const Vec&);
 void SetMatrix(const FCmatrix&)
 //solving eqs
 Vec GaussEliminationSolver();
 Vec LUdecompositionSolver();
 Vec JacobiIterator(double tol=1.E-4);

private:
 //decomposição LU com |L|=1
 void LUdecomposition(FCMatrix&, vector<int>& index); // in case pivotting used
 /* return triangular matrix and changed vector of constants */
 void GaussElimination(FCmatrix&, Vec&);
 FCmatrix M; //matriz de coeffs
 Vec b; //vector de constantes
};
```

EqSolver class

```
#include "Vec.h"
#include "FCmatrixFull.h"

int main() {
  double a[]={4, 2, 1};
  double b[]={-1, 2, 0};
  double c[]={2, 1, 4};

  // make Matrix
  vector<Vec> V;
  V.push_back(Vec(3,a));
  V.push_back(Vec(3,b));
  V.push_back(Vec(3,c));
  FCmatrixFull M(V);

  // constants
  double d[]={4, 2, 9};
  Vec vc(3,d);

  // solve linear system
  EqSolver S(M,vc);
  Vec vsol = S.JacobiIterator();
}
```

Solve the system:

$$\begin{pmatrix} 4 & 2 & 1 \\ -1 & 2 & 0 \\ 2 & 1 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 9 \end{pmatrix}$$

Jacobi algorithm

```
// linear system of m unknowns
Vec x(m); Vec x_aux(m); //zero's
bool btol = false;
int it = 0.; double eps = 1.E-4;
while (!btol && (it++ < 1000)) {
  x_aux = x;
  for (int i=0; i<m; i++) {
    x[i] = 0.;
    for (int j=0; j<m; j++)
      if (i != j) x[i] += -A[i][j]*x_aux[j];
    x[i] += b[i];
    x[i] /= A[i][i];
    // guarantee that all vector entries are converging equally
    if (fabs(x[i]-x_aux[i]) < eps) btol = true;
    else btol = false;
  }
}
```


Gauss-Seidel method

- ✔ The Gauss-Seidel method improves the convergence of the Jacobi method by using every iterated variable in the step
- ✔ Consider the following system:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \Rightarrow \begin{cases} x_0 = (b_0 - A_{01}x_1 - A_{02}x_2)/A_{00} \\ x_1 = (b_1 - A_{10}x_0 - A_{12}x_2)/A_{11} \\ x_2 = (b_2 - A_{20}x_0 - A_{21}x_1)/A_{22} \end{cases}$$

- ✔ the iterations:

$$\begin{pmatrix} x_0^{(0)} \\ x_1^{(0)} \\ x_2^{(0)} \end{pmatrix} \quad \begin{pmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} = \begin{pmatrix} \left(b_0 - A_{01}x_1^{(0)} - A_{02}x_2^{(0)}\right)/A_{00} \\ \left(b_1 - A_{10}x_0^{(1)} - A_{12}x_2^{(0)}\right)/A_{11} \\ \left(b_2 - A_{20}x_0^{(1)} - A_{21}x_1^{(1)}\right)/A_{22} \end{pmatrix} \quad \begin{pmatrix} x_0^{(2)} \\ x_1^{(2)} \\ x_2^{(2)} \end{pmatrix} = \begin{pmatrix} \left(b_0 - A_{01}x_1^{(1)} - A_{02}x_2^{(1)}\right)/A_{00} \\ \left(b_1 - A_{10}x_0^{(2)} - A_{12}x_2^{(1)}\right)/A_{11} \\ \left(b_2 - A_{20}x_0^{(2)} - A_{21}x_1^{(2)}\right)/A_{22} \end{pmatrix}$$

- ✔ It can also be used to solve non-linear systems

Gauss-Seidel algorithm

```
// linear system of m unknowns
Vec x(m); //zero's
Vec x_aux(m); //zero's
bool btol = false;
int it = 0.;
double eps = 1.E-4; //tolerance

while (!btol && (it++ < 1000)) {
  x_aux = x;
  for (int i=0; i<m; i++) {
    x[i] = 0.;
    for (int j=0; j<m; j++)
       if (i != j) x[i] += -A[i][j]*x[j];
    x[i] += b[i];
    x[i] /= A[i][i];
    //guarantee that all vector entries are converging equally
    if (fabs(x[i]-x_aux[i]) < eps) btol = true;
    else btol = false;
  }
}
```


Relaxation

- The convergence of the method does not depend on the initial vector but it can be accelerated using *relaxation*
- The iterated $\mathbf{x_i}$ value is obtained from a weighted (ω) average of its previous value and the iterative formula shown before

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

ω is the *relaxation factor*

- Defining the change on x on the kth iteration without relaxation mechanism as, $\Delta x^{(k)} = |\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}|$

  After p additional iterations, a good estimate of ω can be computed at run time as,
  $\omega \simeq \frac{2}{1+\sqrt{1-(\Delta x^{(k+p)}/\Delta x^{(k)})^{1/p}}}$

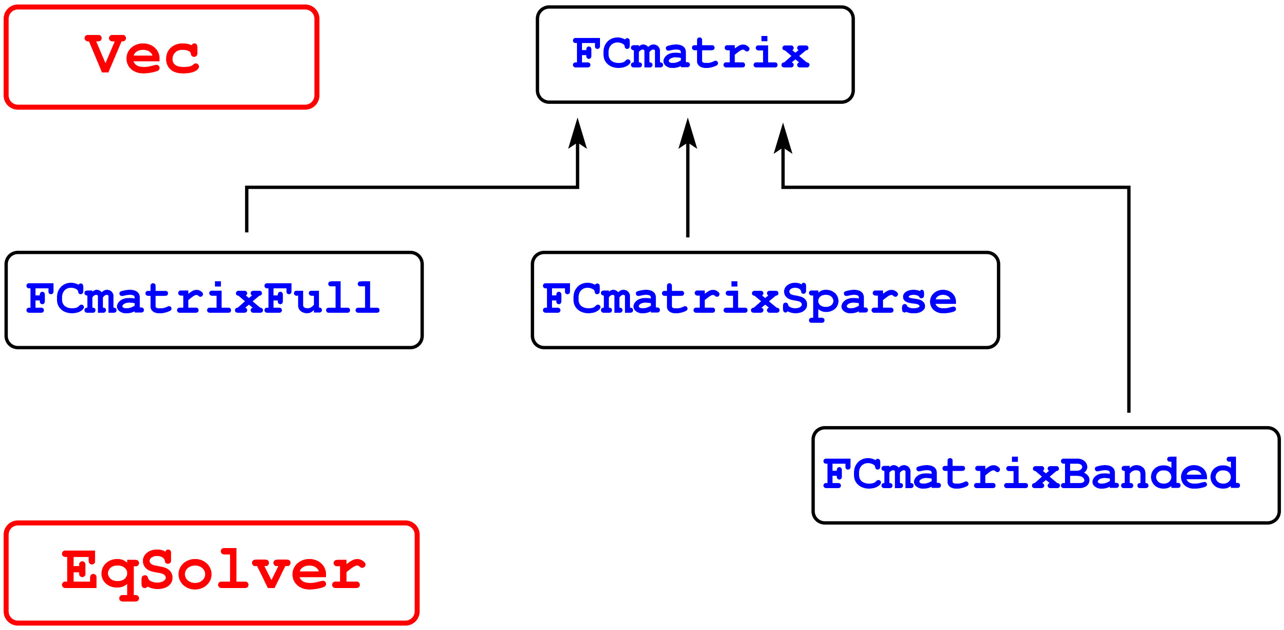
algorithm

```
- realize k iterations (~10)
  without weighting and record
  after the kth iteration the
  change on x

- realize additional p iterations
  and record the change on x for
  the last iteration

- from that iteration on, introduce
  weighting on x calculation
```

C++ class scheme



Vector class

```
class Vec {
 public:
  Vec(int i=0, double x=0); //default constructor
  Vec(int, const double*); //set N elements from array
  Vec(const Vec&); // copy constructor
  ~Vec(); //destructor

  void SetEntries (int, double*);
  int size() const; //Vec size
  double dot(const Vec&); //produto interno
  void swap(int, int); //swap Vec elements

  void Print() const; //class dump
  friend ostream& operator<<(ostream&, const Vec&);

  double& operator[] (int);
  double operator[] (int) const; //Vec is declared const

  void operator=(const Vec&);
  const Vec& operator+=(const Vec&);
  const Vec& operator-=(const Vec&);
  Vec operator+(const Vec&) const;
  Vec operator-(const Vec&) const;
  Vec operator*(const Vec&) const; //x1x2,y1y2,z1z2
  Vec operator*(double) const; //Vec.operator*(k)
  Vec operator-(); // unary operator
  friend Vec operator* (double, const Vec&);

  private:
  int N;
  double *entries;
};
```

```
int main() {
 // build vector
 double* a = new double[5]{1,2,3,4,5};
 Vec V(10, a);

 // print
 V.Print();
 cout << V << endl;

 // free mem
 delete [] a;

 // make a scoped copy
 { Vec V2(V); }

  //operator=
  Vector V3(15, 1.23543);
  R = V; //R.operator=(V)
  Vec B = (R=V); //ERROR

  //scalar
  Vec V4 = 5.34*V; //friend function
  Vec V5 = V*3.24;
}
```

FCmatrix base class

```
classe 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>);

 // pure virtual methods
 virtual Vec GetRow(int i) = 0; // retrieve row i
 virtual Vec GetCol(int i) = 0; // retrieve column i
 virtual double Determinant() = 0;

 virtual void Print();
 virtual void swapRows(int i, int j); // swap rows i,j

protected:
 vector<Vec> M;
 string classname;
};
```


FCmatrixFull class

```
classe 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&); // adicionar duas matrizes de qq tipo
 FCmatrixFull operator-(const FCmatrix&); // subtrair duas matrizes de qq tipo
 FCmatrixFull operator*(const FCmatrix&); // multiplicar duas matrizes de qq tipo
 FCmatrixFull operator*(double lambda); // multiplicar matriz de qq tipo por excalar
 FCmatrixFull operator*(const Vec&); // multiplicar matriz por Vec

 // virtual inherited
 Vec GetRow(int i); // retrieve row i
 Vec GetCol(int i); // retrieve column i
 double Determinant();
 void Print();
 void swapRows(int,int);
 ...
private:
 int rowindices[fm]; // row indices (0,1,2,...)
 int colindices[fn]; // column indices (0,1,2,..)
};
```