

C++ Vector/Matrix/Preconditioner Classes for Large Sparse System

Enrico Bertolazzi

*Dipartimento di Ingegneria Meccanica e Strutturale
Università degli Studi di Trento
via Mesiano 77, I – 38050 Trento, Italia
Enrico.Bertolazzi@ing.unitn.it*

Release 9.0

Abstract

A collection of simple, fast, and efficient classes for manipulating large vectors and large sparse matrices is presented. Its implementation in C++ programming language requires a compiler which supports templates and namespace.

Contents

1	The Basic Vector Matrix and Preconditioner Classes	3
1.1	Loading the library	3
1.2	Class <code>Vector<T></code>	4
1.2.1	Constructors	4
1.2.2	Indexing	4
1.2.3	Changing Dimension	4
1.2.4	Initialization	5
1.2.5	Assignment	5
1.3	Arithmetic Operators on <code>Vector<T></code>	6
1.3.1	<i>Scalar</i> - <code>Vector<T></code> internal operations	6
1.3.2	<i>Scalar</i> - <code>Vector<T></code> operations	6
1.3.3	<code>Vector<T></code> - <code>Vector<T></code> internal operations	7
1.3.4	<code>Vector<T></code> - <code>Vector<T></code> operations	7
1.3.5	Function of <code>Vector<T></code>	7
2	The sparse matrix classes	8
2.1	<code>SparsePattern</code> internal structure	9
2.2	<code>CCoorMatrix<T></code> internal structure	12
2.3	<code>CRowMatrix<T></code> internal structure	15
2.4	<code>CColMatrix<T></code> internal structure	16
2.5	<code>TridMatrix<T></code> internal structure	18
3	Common Function to all sparse matrices not yet considered	19
3.1	Accessing elements	19
3.2	Assignment	19
3.3	Modification of the diagonal	20
3.4	Matrix-Vector multiplication	21
3.5	Matrix inversion	22
4	Preconditiones	22
4.1	Diagonal preconditioner	22
4.2	ILDU preconditioner	22
4.3	No preconditioner	22
5	Iterative solvers	23

1 The Basic Vector Matrix and Preconditioner Classes

This classes library consist of a template class

- **Vector<T>**

which define a dense large column vector. A class

- **SparsePattern.**

Which define a pattern of the nonzero elements which can be used to construct a sparse matrix.

Some classes which define sparse matrices whose nonzero elements are stored in various different formats. The classes available are the following:

- The class **TridMatrix<T>** which implements a *tridiagonal* matrix.
- The class **CCoorMatrix<T>** which implements a sparse *Compressed Coordinate Storage* matrix.
- The class **CRowMatrix<T>** which implements a sparse *Compressed Rows Storage* matrix.
- The class **CColMatrix<T>** which implements a sparse *Compressed Columns Storage* matrix.

Some classes which define sparse preconditioner that can be used in some iterative scheme. The classes available are the following:

- **DPreco<T>** which implements the diagonal preconditioner.
- **ILDUpresco<T>** which implement an incomplete *LDU* preconditioner.

Those classes allow vectors and matrices to be formally treated in software implementations as mathematical objects in arithmetic expressions. For example if **A** is a sparse matrix and **b** is a **Vector<T>** than **A*b** means the matrix-vector product.

And finally a set of template iterative solvers.

1.1 Loading the library

To use the library you must include it by the following piece of code:

```
# include "sparselib.hh"
using namespace sparselib_load ;
```

line 2 is recommended to avoid “**sparselib:**” prefix for the library call.

1.2 Class `Vector<T>`

A `Vector<T>` is defined by specifying the type `T` and optionally its size. For example,

```
Vector<double> b, c(100) ;
```

defines `b` as a vector of double of size 0 while `c` is a vector of double of size 100. You can change the size of the vector by the methods `new_dim` as follows:

```
Vector<double> d ;  
d . new_dim(200) ;
```

so that `d` is a `Vector<double>` of size 200. There are many methods associate to a `Vector<T>` in the following paragraph they are listed.

1.2.1 Constructors

```
1 Vector<T> v ;  
2 Vector<T> v(dim) ;  
3 Vector<T> w(v) ;
```

On line 1 construct the `Vector<T> v` of size 0. On line 2 construct the `Vector<T> v` of of size `dim`. On line 3 construct the `Vector<T> w` as a copy of `Vector<T> v`.

1.2.2 Indexing

Vector instances are indexed as one-dimensional C arrays, and the index numbering follows the standard C convention, starting from zero. Let us define `v` as `Vector<T>`

```
Vector<T> v ;
```

Then, `v[i]` returns a reference to the `T&`-type `i`-th element of `v`;

1.2.3 Changing Dimension

It is possible to change the size of a `Vector<T>` object. For example

```
Vector<double> d ;  
d . new_dim(200) ;
```

the `Vector<T> d` has size 200. The method `size()` return the actual size of the `Vector<T>`. For example defining

```
Vector<double> d(123) ;
```

the method `d.size()` return 123.

1.2.4 Initialization

It is possible to initialize all the components of a `Vector<T>` to a value, for example

```
Vector<float> v(100) ;  
v = 3.14 ;
```

is equivalent to

```
Vector<float> v(100) ;  
for ( int i = 0 ; i < v . size() ; ++i ) v[i] = 3.14 ;
```

although is done more efficiently by the library.

1.2.5 Assignment

It is possible to copy the contents of a `Vector<T>` to another one as the following example show

```
1  Vector<double> a, b, c ;  
  
2  a . new_dim(100) ;  
3  b . new_dim(200) ;  
4  c . new_dim(150) ;  
  
5  c = 3 ;  
6  b = c ;  
7  a = c ;
```

the meaning of line 5 should be clear. Lines 6 and 7 are equivalent to

```
int i ;  
for ( i = 0 ; i < min(b.size(), c.size()) ; ++i ) b[i] = c[i] ;  
for ( i = 0 ; i < min(a.size(), c.size()) ; ++i ) a[i] = c[i] ;
```

where you can notice that only the values that can be stored are assigned.

It is possible to initialize many vectors same value as in the following expressions

```
1  Vector<double> a, b, c ;  
2  a . new_dim(100) ;  
3  b . new_dim(200) ;  
4  c . new_dim(150) ;  
5  a = b = c = 3 ;
```

but take attention because line 5 is not equivalent to

```
a = 3 ;  
b = 3 ;  
c = 3 ;
```

in fact line 5 is equivalent to

```

c = 3 ;
b = c ;
a = b ;

```

so that we have

- the **Vector<T>** **c** is initialized with *all* its 150 elements set to 5.
- the **Vector<T>** **b** is initialized with *only* its first 150 elements set to 1 while the remaining are undefined
- the **Vector<T>** **a** is initialized with *all* its first 100 elements set to 1.

1.3 Arithmetic Operators on Vector<T>

A set of usual arithmetic operators are explicitly defined on vector-type data. If not otherwise specified, the operators extend the corresponding scalar operation in a *component-wise* fashion. Hence, for vectors with size **dim**, the component index **i** in all the following expressions is supposed to run through 0 to **dim-1**.

Let us define the three double precision vectors **a**, **b**, and **c**, that we shall use in all the following examples

```

int const dim = 100 ;
Vector<double> a(dim), b(dim), c(dim) ;

```

The arithmetic operators defined on vectors are given in the following sections.

1.3.1 Scalar-Vector<T> internal operations

Command	Equivalence
a += 2	for all i do a[i] += 2
a -= 2	for all i do a[i] -= 2
a *= 2	for all i do a[i] *= 2
a /= 2	for all i do a[i] /= 2
i=0,1,...,a.size()-1	

1.3.2 Scalar-Vector<T> operations

Command	Equivalence
a = b + 2	for all i do a[i] = b[i] + 2
a = 3 + b	for all i do a[i] = 3 + b[i]
a = b - 2	for all i do a[i] = b[i] - 2
a = 3 - b	for all i do a[i] = 3 - b[i]
a = b * 2	for all i do a[i] = b[i] * 2
a = 3 * b	for all i do a[i] = 3 * b[i]
a = b / 2	for all i do a[i] = b[i] / 2
a = 3 / b	for all i do a[i] = 3 / b[i]
i=0,1,...,min{a.size(),b.size()}-1	

1.3.3 Vector<T>-Vector<T> internal operations

Command	Equivalence
<code>a += b</code>	for all <code>i</code> do <code>a[i] += b[i]</code>
<code>a -= b</code>	for all <code>i</code> do <code>a[i] -= b[i]</code>
<code>a *= b</code>	for all <code>i</code> do <code>a[i] *= b[i]</code>
<code>a /= b</code>	for all <code>i</code> do <code>a[i] /= b[i]</code>
<code>i=0,1,...,min{a.size(),b.size()}-1</code>	

1.3.4 Vector<T>-Vector<T> operations

Command	Equivalence
<code>b = +a</code>	for all <code>i</code> do <code>b[i] = +a[i]</code>
<code>b = -a</code>	for all <code>i</code> do <code>b[i] = -a[i]</code>
<code>i=0,1,...,min{a.size(),b.size()}-1</code>	
<code>c = a + b</code>	for all <code>i</code> do <code>c[i] = a[i] + b[i]</code>
<code>c = a - b</code>	for all <code>i</code> do <code>c[i] = a[i] - b[i]</code>
<code>c = a * b</code>	for all <code>i</code> do <code>c[i] = a[i] * b[i]</code>
<code>c = a / b</code>	for all <code>i</code> do <code>c[i] = a[i] / b[i]</code>
<code>i=0,1,...,min{a.size(),b.size(),c.size()}-1</code>	

1.3.5 Function of Vector<T>

Let be $n = \min \{a.size(), b.size()\}$,

- `T dot(Vector<T> const & a, Vector<T> const & b)`

$$\text{dot}(a,b) = \sum_{i=0}^{n-1} a[i] * b[i]$$

- `T dot_div(Vector<T> const & a, Vector<T> const & b)`

$$\text{dot_div}(a,b) = \sum_{i=0}^{n-1} a[i] / b[i]$$

- `T dist(Vector<T> const & a, Vector<T> const & b)`

$$\text{dist}(a,b) = \sqrt{\sum_{i=0}^{n-1} (a[i] - b[i])^2}$$

- `T normi(Vector<T> const & a)`

$$\text{normi}(a) = \|a\|_{\infty} = \max \{|a[0]|, |a[1]|, \dots, |a[a.size()-1]|, \}$$

- `T norm1(Vector<T> const & a)`

$$\text{norm1}(\mathbf{a}) = \|\mathbf{a}\|_1 = \sum_{i=0}^{\mathbf{a.size()}-1} |\mathbf{a}[i]|, \quad n = \mathbf{a.size()}$$

- `T norm2(Vector<T> const & a)`

$$\text{norm2}(\mathbf{a}) = \|\mathbf{a}\|_2 = \sqrt{\sum_{i=0}^{\mathbf{a.size()}-1} |\mathbf{a}[i]|^2},$$

- `T normp(Vector<T> const & a, T const & p)`

this function return

$$\text{normp}(\mathbf{a}, p) = \|\mathbf{a}\|_p = \left(\sum_{i=0}^{\mathbf{a.size()}-1} |\mathbf{a}[i]|^p \right)^{1/p},$$

- `T max(Vector<T> const & a)`

$$\text{max}(\mathbf{a}) = \max\{\mathbf{a}[0], \mathbf{a}[1], \dots, \mathbf{a}[\mathbf{a.size()}-1]\},$$

- `T min(Vector<T> const & a)`

$$\text{min}(\mathbf{a}) = \min\{\mathbf{a}[0], \mathbf{a}[1], \dots, \mathbf{a}[\mathbf{a.size()}-1]\},$$

2 The sparse matrix classes

Those classes have different internal structure but they are derived by a unique base classes **Sparse**. This class is practically empty and is used as a pivot for internal operations. It contains minimal informations and virtual functions for accessing elements of the derived classes. The methods of the class are the following

- `index_type nrows()`
`index_type ncols()`
`index_type min_size()`
`index_type max_size()`

return the dimensions of the derived sparse matrix. `max_size()` is the maximum between `nrows()` and `ncols()` while `min_size()` is the minimum.

- `index_type lower_nnz()`

return the number of not zero (allocated) of the derived sparse matrix *under* the main diagonal.

- `index_type diag_nnz()`

return the number of not zero (allocated) of the derived sparse matrix *on* the main diagonal.

- `index_type upper_nnz()`
return the number of not zero (allocated) of the derived sparse matrix *over* the main diagonal.
- `bool is_ordered()`
Some sparse matrix can be internally ordered. This simplify some operation such as accessing elements or conversion and so on. This flag return `true` if the matrix is ordered.

The following methods *are* defined in the derived classes.

- `void Begin()`
`void Next()`
`bool End()`
this three methods are useful for accessing all the nonzero elements of the derived sparse matrix. The methods `Begin()` set the iterator at the begin of the loop, `Next()` go the next item while `End()` return `true` unless we have at the end of the loop. If `S` is a sparse matrix derived by the object `Sparse` the following loop permit to access all the elements:

```
for ( S . Begin() ; S . End() ; S . Next() ) {
    // do something
}
```

- `index_type i_index()`
`index_type j_index()`
`T value(T*)`
those methods access values of the actual elements pointed by the iterator. The values of `i_index()` and `j_index()` are respectively the row and the column of the pointed element, while `value()` is its value. For example to print all the stored values of the `Sparse` object `S` we can do:

```
for ( S . Begin() ; S . End() ; S . Next() ) {
    cout << " row = " << S . i_index()
        << " col = " << S . j_index()
        << " value = " << S . value((value_type*)(0)) << endl ;
}
```

2.1 SparsePattern internal structure

The sparse pattern internal structure is essentially a sparse compressed coordinate ones. It consists of two big vector of unsigned integer which contain the coordinate of nonzero elements. We call `I` the vector that store the first coordinate, while we call `J` the vector that store the second coordinate.

For example the following 6×7 sparse matrix pattern

$$\mathbf{S} = \begin{bmatrix} * & * & & & & & * \\ & * & * & & & & \\ & & & * & * & & \\ & * & & & * & & \\ * & & * & & & * & \\ & * & & & & & * \end{bmatrix} \quad (1)$$

can be stored as follows

\mathbf{I}	0	0	0	1	1	2	2	3	3	4	4	4	5	5
\mathbf{J}	0	1	6	1	2	3	4	1	4	0	2	5	1	6

Notice that the index follows the C convention, starting from 0. The class **SparsePattern** try to manage such a structure in a simple way for the user. To define a **SparsePattern** class you can use uno of the following scripture

```

1 SparsePatter sp ;
2 SparsePatter sp(nrow, ncol, max_nnz) ;
3 SparsePatter sp(sp1) ;
4 SparsePatter sp(sobj) ;
```

- Line 1 define an empty **SparsePattern** class,
- Line 2 define a **SparsePattern** class on a pattern of **nrow** rows and **ncol** columns. The number **max_nnz** is an unsigned integer which determine the maximum number of pattern elements stored in the class, in practice is the dimension of vector **I** and **J**.
- Line 3 define a **SparsePattern** object which is the copy of **SparsePattern** object **sp1**.
- Line 4 define a **SparsePattern** object which is the pattern of nonzero of the **Sparse** object **sobj**. In this way it is possible to obtain the sparse pattern of any object derived by **Sparse**.

It is possible in any moment to change the sizes and the maximum number of nonzero by the **new_dim** methods:

```

sp . new_dim(nrow, ncol, max_nnz) ;
sp . new_dim(sp1) ;
sp . new_dim(sobj) ;
```

so that

- Line 2 is equivalent to

```
SparsePatter sp ;
sp . new_dim(nrow, ncol, max_nnz) ;
```

- Line 3 is equivalent to

```
SparsePatter sp ;
sp . new_dim(sp1) ;
```

- Line 4 is equivalent to

```
SparsePatter sp ;
sp . new_dim(sobj) ;
```

to complete the basic description of the **SparsePattern** a sintetic description of the remaining methods of the class are presented

- **insert(index_type i, index_type j)**

this method permit to insert an item of nonzero. For example pattern **S** of equation (1) can be constructed as

```
SparsePattern sp(6,7,20) ;
sp.insert(0, 0) ; sp.insert(0, 1) ; sp.insert(0, 6) ;
sp.insert(1, 1) ; sp.insert(1, 2) ;
sp.insert(2, 3) ; sp.insert(2, 4) ;
sp.insert(3, 1) ; sp.insert(3, 4) ;
sp.insert(4, 0) ; sp.insert(4, 2) ; sp.insert(4, 5) ;
sp.insert(5, 1) ; sp.insert(5, 6) ;
```

- **index_type const * getI()**
index_type const * getJ()

with this two methods it is possible to access the elements of the sparse pattern, for example continuing the previous example we have

```
index_type i = A . getI()[2] ;
index_type j = A . getJ()[2] ;
```

and **i=0** and **j=6**.

- **index_type nnz()**

This method return the number of nonzero elements in the pattern, for example continuing the previous example we have

```
index_type nz = A . nnz() ;
```

and **nz=14**.

- **index_type Getfree()**

This method return the number of nonzero elements that can be stored in the pattern, for example continuing the previous example we have

```
index_type fnz = A . get_free() ;
```

and **fnz=6**.

- **internal_order()**

This methods reorder internally the nonzero elements of the sparse pattern in such a way if **k1** ≤ **k2** we have one of the two following cases

1. **I(k1) < I(k2)**
2. **I(k1) = I(k2)** and **J(k1) ≤ J(k2)**

Moreover all duplicated entries are deleted. This method is used when we use **SparsePattern** to construct a sparse matrix.

- **bool is_ordered()**

this methods return **true** if the elements inside the sparse pattern are ordered, **false** otherwise.

- **free()** this function free the memory used by the **SparsePattern** class.

2.2 CCoordMatrix<T> internal structure

The class **CCoordMatrix<T>** implement a *Compressed Coordinate* storage sparse scheme. It consists of two big vector of unsigned integer which contain the coordinate of nonzero elements and a big one of real number which contain the values. We call **I** the vector that store the rows coordinate, **J** the vector that store the columns coordinate and **A** the vector that store the nonzero values. For example the following 6 × 7 sparse matrix

$$A = \begin{bmatrix} 1 & 2 & & & & & 9 \\ & -1 & 0 & & & & \\ & & & 3 & 4 & & \\ & 2 & & & 5 & & \\ 2 & & -2 & & & 1 & \\ & -1.5 & & & & & -1 \end{bmatrix} \quad (2)$$

can be stored as follows

I	0	0	0	1	1	2	2	3	3	4	4	4	5	5
J	0	1	6	1	2	3	4	1	4	0	2	5	1	6
A	1	2	9	-1	0	3	4	2	5	2	-2	1	-1.5	-1

Notice that the index follows the C convention, starting from 0. The class `CCoorMatrix<T>` try to manage such a structure in a simple way for the user. To define a `CCoorMatrix<T>` class you can use one of the following scripture

```
1 CCoorMatrix<double> ccoor ;
2 CCoorMatrix<double> ccoor(nrow, ncol, max_nnz) ;
3 CCoorMatrix<double> ccoor(sp) ;
4 CCoorMatrix<double> ccoor(ccoor1) ;
5 CCoorMatrix<double> ccoor(sobj) ;
```

- Line 1 define an empty `CCoorMatrix<double>` class.
- Line 2 define a `CCoorMatrix<double>` class of `nrow` rows and `ncol` columns. The number `max_nnz` is an unsigned integer which determine the maximum number of elements stored in the class, in practice is the dimension if vector `I`, `J` and `A`.
- Line 3 define a `CCoorMatrix<double>` class with the sparsity pattern defined of the `SparsePattern` class `sp`. `nrow` will be `sp.nrows()`, `ncol` will be `sp.ncols()`, `max_nnz` will be `sp.nnz()+sp.get_free()`.
- Line 4 define a `CCoorMatrix<double>` class with which is the copy of the `CCoorMatrix<double>` object `ccoor1`.
- Line 5 define a `CCoorMatrix<double>` class with which is the copy of the `Sparse` object `sobj`.

It is possible in any moment to change the sizes and the maximum number of nonzero by the `new_dim` methods:

```
ccoor . new_dim(nrow, ncol, max_nnz) ;
ccoor . new_dim(sp) ;
ccoor . new_dim(ccoor1) ;
ccoor . new_dim(sobj) ;
```

so that

- Line 2 is equivalent to

```
CCoorMatrix<double> ccoor ;
ccoor . new_dim(nrow, ncol, max_nnz) ;
```

- Line 3 is equivalent to

```
CCoorMatrix<double> ccoor ;
ccoor . new_dim(sp) ;
```

- Line 4 is equivalent to

```
CCoorMatrix<double> ccoor ;
ccoor . new_dim(ccoor1) ;
```

- Line 5 is equivalent to

```
CCoorMatrix<double> ccoor ;
ccoor . new_dim(sobj) ;
```

to complete the basic description of the **CCoorMatrix** a sintetic description of the remaining methods of the class are presented

- **value_type & insert(index_type i, index_type j)**

this method permit to insert an item in the matrix. For example matrix **A** of equation (2) can be constructed with

```
CCoorMatrix<double> A(6,7,20) ;
A.insert(0, 0) = 1      ; A.insert(0, 1) = 2      ; A.insert(0, 6) = 9 ;
A.insert(1, 1) = -1     ; A.insert(1, 2) = 0      ;
A.insert(2, 3) = 3      ; A.insert(2, 4) = 4      ;
A.insert(3, 1) = 2      ; A.insert(3, 4) = 5      ;
A.insert(4, 0) = 2      ; A.insert(4, 2) = -2     ; A.insert(4, 5) = 1 ;
A.insert(5, 1) = -1.5   ; A.insert(5, 6) = 1      ;
```

- **index_type const * getI()**
index_type const * getJ()
value_type const * getA()

with this three methods it is possible to access the elements of **CCoorMatrix<double>**, for example continuing the previous example we have

```
index_type i = A . getI()[2] ;
index_type j = A . getJ()[2] ;
value_type a = A . getA()[2] ;
```

and **i=0**, **j=6** and **a=9**.

- **index_type nnz()**

This method return the number of nonzero elements in the matrix, for example continuing the previous example we have

```
index_type nz = A . nnz() ;
```

and **nz=14**.

- **index_type get_free()**

This method return the number of nonzero elements that can be stored in the pattern, for example continuing the previous example we have

```
index_type fnz = A . get_free() ;
```

and **fnz=6**.

- `internal_order()`

This methods reorder internally the nonzero elements of `CCoordMatrix<double>` in such a way if $k1 \leq k2$ we have one of the two following cases

1. $I(k1) < I(k2)$
2. $I(k1) = I(k2)$ and $J(k1) \leq J(k2)$

Moreover all duplicated entries are deleted. This method is used when we access the elements of `CCoordMarrix` randomly. this methods return `true` if the elements inside the class `CCoordMatrix<double>` are ordered, `false` otherwise.

- `free()` this function free the memory used by the `CCoordMatrix<double>` class.

2.3 CRowMatrix<T> internal structure

The class `CRowMatrix<T>` implement a *Compressed Rows* storage sparse scheme. It consists of two big vector of unsigned integer which contain the coordinate of nonzero elements and a big one of real number which contain the values. We call **R** the vector that store the start position of each row, **J** the vector that store the column coordinate and **A** the vector that store the nonzero values. For example the sparse matrix (2) can be stored as follows

R	0	3	5	7	9	12								
J	0	1	6	1	2	3	4	1	4	0	2	5	1	6
A	1	2	9	-1	0	3	4	2	5	2	-2	1	-1.5	-1

The number of nonzero's is stored in a variable called `nnz` in for matrix **A** this number is 14. Notice that the index follows the C convention, starting from 0. The class `CRowMatrix<T>` try to manage such a structure in a simple way for the user. To define a `CCoordMatrix<T>` class you can use one of the following scripture

```

1 CRowMatrix<double> crow ;
2 CRowMatrix<double> crow(sp) ;
3 CRowMatrix<double> crow(crow1) ;
4 CRowMatrix<double> crow(sobj) ;
```

- Line 1 define an empty `CRowMatrix<double>` class.
- Line 2 define a `CRowMatrix<double>` class with the sparsity pattern defined in the `sp SparsePattern` class.
- Line 3 define a `CRowMatrix<double>` class which is the copy of the `CRowMatrix<double>` object `crow1`.
- Line 4 define a `CRowMatrix<double>` class which is the copy of the `Sparse` object `crow1`.

It is possible in any moment to change the sizes and the maximum number of nonzero by the `new_dim` method:

```

crow . new_dim(sp) ;
crow . new_dim(crow1) ;
crow . new_dim(sobj) ;

```

so that

- line 2 is equivalent to

```

CRowMatrix<double> crow ;
crow . new_dim(sp) ;

```

- line 3 is equivalent to

```

CRowMatrix<double> crow ;
crow . new_dim(crow1) ;

```

- line 4 is equivalent to

```

CRowMatrix<double> crow ;
crow . new_dim(sobj) ;

```

to complete the basic description of the **CRowMatrix<double>** a sintetic description of the remaining methods of the class are presented

- **index_type nnz()**
those methods return the number of nonzero elements stored by the matrix
- **index_type const * getR()**
index_type const * getJ()
value_type const * getA()
with this three methods it is possible to access the internal structure of **CRowMatrix<double>**.
- **nnz_stat(index_type & lnz, index_type & dnz, index_type & Unz)**
This method return the number of nonzero elements under the diagonal, on the diagonal and over the diagonal.

2.4 CColMatrix<T> internal structure

The class **CColMatrix<T>** implement a *Compressed Columns* storage sparse scheme. It consists of two big vector of unsigned integer which contain the coordinate of nonzero elements and a big one of real number which contain the values. We call **I** the vector that store the row index of each elements, **C** the vector that store the starting position of the columns and **A** the vector that store the nonzero values. For example the sparse matrix (2) can be stored as follows

I	0	4	0	1	3	5	1	4	2	2	3	4	0	5
C	0	2	6	8	9	11	12							
A	1	2	2	-1	2	-1.5	0	-2	3	4	5	1	9	-1

Notice that the index follows the C convention, starting from 0. The class `CColMatrix<T>` try to manage such a structure in a simple way for the user. To define a `CColMatrix<T>` class you can use one of the following scripture

```
1 CColMatrix<double> ccol ;
2 CColMatrix<double> ccol(sp) ;
3 CColMatrix<double> ccol(ccol1) ;
4 CColMatrix<double> ccol(sobj) ;
```

- Line 1 define an empty `CColMatrix<double>` class.
- Line 2 define a `CColMatrix<double>` class with the sparsity pattern defined in the `SparsePattern` class `sp`. `nrow` will be `sp . nrow()`, `ncol` will be `sp . ncol()`.
- Line 3 define a `CColMatrix<double>` class which is the copy of the `CColMatrix<double>` object `ccol1`.
- Line 4 define a `CColMatrix<double>` class which is the copy of the `Sparse` object `sobj`.

It is possible in any moment to change the sizes and the maximum number of nonzero by the `new_dim` method:

```
ccol . new_dim(sp) ;
ccol . new_dim(ccol1) ;
ccol . new_dim(sobj) ;
```

so that

- line 2 is equivalent to

```
CColMatrix<double> ccol ;
ccol . new_dim(sp) ;
```

- line 3 is equivalent to

```
CColMatrix<double> ccol ;
ccol . new_dim(ccol1) ;
```

- line 4 is equivalent to

```
CColMatrix<double> ccol ;
ccol . new_dim(sobj) ;
```

to complete the basic description of the `CColMatrix<double>` a sintetic description of the remaining methods of the class are presented

- `index_type nnz()`
those methods return the number of nonzero elements stored by the matrix

- `index_type const * getI()`
`index_type const * getC()`
`value_type const * getA()`

with this three methods it is possible to access the internal structure of `CColMatrix<double>`.

2.5 TridMatrix<T> internal structure

The class `TridMatrix<T>` implement a sparse band matrix. It consists of a big matrix of `value_type` which contain the values of nonzero elements. We call **M** this big matrix which represents an $n \times m$ matrix which stores the rows of nonzero. For example the following band matrix

$$A = \begin{bmatrix} 2 & -1 & & & & & \\ 1 & 2 & -2 & & & & \\ & 2 & 2 & -3 & & & \\ & & 3 & 2 & -4 & & \\ & & & 4 & 2 & -5 & \\ & & & & 5 & 2 & -6 \\ & & & & & 6 & 2 \end{bmatrix}$$

can be stored as follows

L	D	U
*	2	-1
1	2	-2
2	2	-3
3	2	-4
4	2	-5
5	2	-6
6	2	*

where * means unused elements. Notice that the index follows the C convention, starting from 0. The class `TridMatrix<T>` try to manage such a structure in a simple way for the user. To define a `TridMatrix<T>` class you can use one of the following scripture

```
1 TridMatrix<double> tm ;
2 TridMatrix<double> tm(100) ;
3 TridMatrix<double> tm(tm1) ;
```

- Line 1 define an empty `TridMatrix<double>` class.
- Line 2 define a 100×100 `TridMatrix<double>`.

- Line 3 define a `TridMatrix<double>` class which is the copy of the `TridMatrix<double> tm1`.

It is possible in any moment to change the sizes and the maximum number of nonzero by the `new_dim` method:

```
tm . new_dim(nrow,ncol,ldiag,udiag) ;
```

so that line 2 is equivalent to

```
TridMatrix<double> tm
tm . new_dim(100) ;
```

3 Common Function to all sparse matrices not yet considered

3.1 Accessing elements

- `value_type const & operator () (index_type i, index_type j)`
this operator permits to access the elements of the sparse matrix in a random way. For example given a sparse matrix **A** the code **A(i, j)** return the *reference* of the elements of the matrix at the **i**-th row and **j**-th column. If at the (i, j) coordinate there are no elements an error is produced.
- `value_type & ref(index_type i, index_type j)`
this operator permits to access the elements of the sparse matrix in a random way. For example given a sparse matrix **A** the code **A . ref(i, j)** return the *value* of the elements of the matrix at the **i**-th row and **j**-th column. If at the (i, j) coordinate there are no elements the value 0 is returned.
- `index_type Position(index_type i, index_type j)`
this operator return the position of the (i, j) elements in the internal structure. If the element (i, j) do not exist it return `max_nnz()`.

3.2 Assignment

- `set_row(index_type nr, value_type const & value)`
this function set to **value** all the elements of the **nr** row. For example if **A** is a sparse matrix

```
A . set_row(12, 0) ;
```


set to 0 the 12-th row.
- `scale_row(index_type nr, value_type const & s)`
this function set to multiply by **s** all the elements of the **nr** row.
- `set_col(index_type nc, value_type const & value)`
this function set to **value** all the elements of the **nc** column. For example if **A** is a sparse matrix

```
A . set_col(5, 1.0) ;
```

set to 1 the 5-th column.

- **scale_col(index_type nc, value_type const & s)**
this function multiply by **s** all the elements of the **nc** column.
- **operator = (value_type const & v)**
this function set **v** to all the components of the diagonal, and 0 the others elements. For example

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & 1 & \\ 1 & 2 & & & 5 \\ & & 2 & -3 & \\ & & 3 & 2 & -4 \\ & & & 4 & 2 \end{bmatrix}, \quad (\mathbf{A} = 2.23) = \begin{bmatrix} 3.23 & 0 & & 0 & \\ 0 & 3.23 & & & 0 \\ & & 3.23 & 0 & \\ & & 0 & 3.23 & 0 \\ & & & 0 & 3.23 \end{bmatrix}$$

- **operator = (Vector<Real> const & v)**
this function set **v** to the diagonal values of the sparse matrix. For example

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & 1 & \\ 1 & 2 & & & 5 \\ & & 2 & -3 & \\ & & 3 & 2 & -4 \\ & & & 4 & 2 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix},$$

$$(\mathbf{A} = \mathbf{v}) = \begin{bmatrix} 1 & 0 & & 0 & \\ 0 & 2 & & & 0 \\ & & 3 & 0 & \\ & & 0 & 0 & 0 \\ & & & 0 & 0 \end{bmatrix}$$

notice that only the first **v.size()** diagonal elements are set while the rest of the matrix is set to 0.

3.3 Modification of the diagonal

- **operator += (Real const v)**
operator -= (Real const v)

this function add (or subtract) \mathbf{v} to all the components of the diagonal. For example

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & 1 & \\ 1 & 2 & & & 5 \\ & & 2 & -3 & \\ & & 3 & 2 & -4 \\ & & & 4 & 2 \end{bmatrix}, \quad (\mathbf{A} += 2) = \begin{bmatrix} 4 & -1 & & 1 & \\ 1 & 4 & & & 5 \\ & & 4 & -3 & \\ & & 3 & 4 & -4 \\ & & & 4 & 4 \end{bmatrix}$$

- `operator += (Vector<Real> const & v)`
`operator -= (Vector<Real> const & v)`

this function add (or subtract) \mathbf{v} to the diagonal values of the sparse matrix. For example

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & 1 & \\ 1 & 2 & & & 5 \\ & & 2 & -3 & \\ & & 3 & 2 & -4 \\ & & & 4 & 2 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix},$$

$$(\mathbf{A} -= \mathbf{v}) = \begin{bmatrix} 1 & -1 & & 1 & \\ 1 & 0 & & & 5 \\ & & -1 & -3 & \\ & & 3 & 2 & -4 \\ & & & 4 & 2 \end{bmatrix}$$

notice that only the first `v.size()` diagonal elements are set while the rest of the matrix is unchanged.

3.4 Matrix-Vector multiplication

The operator `*` define the matrix-vector multiplication between a sparse matrix `Matrix<T>` and a full vector `Vector<T>`. If `M` is any of the previous defined sparse matrix and `a`, `b` and `c` are full vector and `s` is a scalar, the only supported multiplication are the following:

```

1  b = M * a ;
2  b = s * (M * a) ;
3  b = a + M * c ;
4  b = a - M * c ;
5  b = a + s * (M * c) ;
```

in line 5 notice that parentheses are necessary. This kind of operation should be sufficient for all algorithm s eventually splitting the expressions.

3.5 Matrix inversion

The operator `/` define the vector-matrix division between `Vector<T>` and a sparse matrix. This operation is actually defined only on `TridMatrix<T>` class. To solve a sparse linear system with the other sparse format you can use the template iterative solvers furnished with the library. For example in `A` is a sparse matrix and `b` is a `Vector<T>` that `b/A` is a `Vector<T>` solution of the problem $A \cdot x = b$.

4 Preconditiones

Two simple preconditioner class are included in the library:

4.1 Diagonal preconditioner

```
1  DPreco<T> D ;  
2  D . build(matrix) ;
```

On line 1 construct a diagonal preconditioner class. On line 2 the preconditioner is builded for the matrix object `matrix`.

4.2 ILDU preconditioner

```
1  ILDUPreco<T> ILDU ;  
2  ILDU . build(matrix) ;  
3  ILDU . build(matrix,pattern) ;
```

On line 1 construct an incomplete factorization preconditioner class. On line 2 the preconditioner is builded for the matrix object `matrix`. The pattern for the incomplete factorization is the same of the matrix object `matrix`.

On line 2 the preconditioner is builded for the matrix object `matrix` while the pattern for the incomplete factorization is the same of the object `pattern`. The object pattern can be any of the sparse matrix object or a `SparsePattern` object.

4.3 No preconditioner

Sometimes one do not want to use a preconditioner. The `IdPreco` class is used to build as preconditioners an identity matrix.

```
1  IdPreco<T> Id ;  
2  Id . build(matrix) ;
```

On line 1 construct a NULL preconditioner class. On line 2 the preconditioner is builded for the matrix object `matrix`.

5 Iterative solvers

Here is an example of the use of the iterative solver:

```
1  CRowMatrix<T> A ;
2  Vector<T>      b, x ;
3  ILDUPreco      P ;
4  T              epsi ;
5  int            max_iter, iter ;
6  .
7  .
8  .
9  .
10 T res = cg(A, b, x, P, epsi, max_iter, iter) ;

11 T res = bicgstab(A, b, x, P, epsi, max_iter, iter) ;

12 T res = gmres(A, b, x, P, epsi, max_sub_iter, max_iter, iter) ;
```

In the example

- **A**: is the coefficients matrix;
- **b**: is the known vector;
- **x**: is the vector which will contains the solution;
- **P**: is the preconditioner object class;
- **espi**: is the admitted tolerance;
- **max_sub_iter**: for **gmres** is the maximum number of iteration before restarting;
- **max_iter**: is the maximum number of allowable iterations;
- **iter**: is the number of iterations done;
- **tol**: the infinity norm of the last residual;
- **res**: the residual of the approximated solution;