SparseTool: an Extensible Toolkit to Handle Sparse Matrices

Enrico Bertolazzi
Dip. Ingegneria Meccanica e Strutturale – Università di Trento

SparseTool is a collection of simple and efficient classes for manipulating large vectors and large sparse matrices. Operator overloading permits to hide complex storage format and manipulate sparse matrix in simple and transparent way. The implementation is in C++ programming language using expression templates and static polymorphism programming paradigm for better performance. The toolkit consisting in only one large header file so that it does not require any installation or compilation procedure.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra – Sparse, structured, and very large systems (direct and iterative methods); G.4 [**Mathematical Software**]: User interfaces; D.1.3 [**Programming Techniques**]: Object-oriented Programming

General Terms: Algorithms, Design

Additional Key Words and Phrases: Sparse Matrices, C++

1. INTRODUCTION

Sparse matrices are widely used in scientific computation and are ubiquitous in many applications in science, engineering and finance. For example, sparse matrices can be found in computational fluid dynamics, large scale optimization, electric circuit simulation and, in general, in the numerical solution of Partial Differential Equations (PDEs). A number of these problems are easy to manage due to the regularity of the underlying data. For example, Finite Difference or Finite Element over regular grids in numerical solutions of PDEs fall in this category. Regular grids often produce structured matrices that permit efficient storage and management. In any case, other problems are built over irregular grids, thus, the resulting matrices may have arbitrary patterns. To manage general unstructured matrices, there are various storage schemes with different performance and storage overhead. The manipulation of sparse matrices, as for example, nonzero insertion and multiplication by a vector is normally an easy task. However, it may involve few or more lines of code that can make the algorithm obscure in a large application. An effective way to reduce the development effort and minimize the probability of making mistakes in the software implementation is to use libraries for the most common (and repetitive) tasks. For non sparse

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

linear algebra. *de facto* standard is provided by the BLAS/LAPACK library [Anderson et al. 1992; Demmel 1989; Oppe and Kincaid. 1990; Dongarra and Walker 1995; Dongarra et al. 1998]. This library consists in a wide collection of FORTRAN routines that cover a large number of algorithms of Linear Algebra for different types of matrices, i.e. symmetric, banded etc.

For sparse matrices the situation is less satisfactory. Indeed, there are many different formats for the sparse matrix storage that are suitable for different matrix types. For this reason, an approach similar to the BLAS/LAPACK choice would produce an explosion in the number of routines to cover all the data/algorithm combinations. Moreover only a selected subset of operations and algorithms is normally implemented, as for example, in SPARSKIT [Saad 1990] and SparseLib++ [Dongarra et al. 1994]. These libraries uses some standard sparse storage formats the most important of which are the Compressed Coordinate (CCOOR) format, the Compressed Row (CROW) format and the Compressed Column (CCOL) format [Barrett et al. 1994]. According to this approach, even if the user is forced to adhere to the choice made by the creator of the libraries every application area may have a different way of efficient storing and accessing the nonzeros entries of the sparse matrix so a storage scheme that work well for an application can be bad for another one.

To circumvent this problem the approach used in Sparse BLAS [Duff et al. 2002] is to avoid the choice of a particular storage scheme abstracting the the representation of the sparse matrix. This is done in a non-Object-Oriented language by using the concept of *handle*.

In an Object Oriented Language as the C++ programming language the storage scheme is hidden by using a combination of inheritance and polymorphism [Lippman 2000; Stroustrup 1998]. In particular, in C++ the polymorphism can be realized by using virtual methods or templates. Dynamic polymorphism, where type dependence for the actual method is resolved at runtime, is obtained by using virtual methods. This results in an overhead for calling the method. Static polymorphism implemented by using templates does not incur in the runtime overhead but needs to bound the method to the code at compile time. For very simple methods like insertion of nonzero the overhead of a virtual call can be comparable to the computational cost of the method. Thus, for performance reason, static polymorphism is preferable whenever is possible [Manzini and Mazet 2002; Veldhuizen 1998; Glass and Schuchert 1995; Musser and Saini 1996; Plauger et al. 2000].

SparseTool is an object-oriented software designed to simplify the construction and manipulation of sparse matrices in numerical solver for physics and engineering applications by hiding to the user the details of the matrix structure and its memory management. SparseTool offers a full support of variable length full vectors, some matrix-vector functions and few iterative algorithms for linear system solution. The toolkit do not offer a complete interface to sparse direct solver as in Sala et al. [2008] but instead furnish tools to easily interface with other libraries.

The toolkit can be used standalone or can be used as a glue with an external high performance sparse matrix library or a sparse linear solver.

A sparse matrix library must have one or more linear system solvers. For these reason, some well known semi-iterative methods such as BiCGSTAB [van der Vorst 1992] or GMRES [Saad and Schultz 1986; Frayssé et al. 2005] are normally found in these libraries (for example in SPARSKIT or IML++ [Dongarra et al. 1994]). Direct methods are competitive but extremely complex to implement. If the sparse matrix is in one of the standard formats (CCOOR, CROW or CCOL) a library like SuperLU [Demmel et al. 1999; Li and Demmel 2003; Li 2005], UMFPACK [Davis 2004; Davis et al. 2004; Amestoy et al. 2004] and MA48 [Duff et al. 1996] can be used. For symmetric linear system a review of available solver can be found in Gould et al. [2007].

To obtain flexibility, extensibility and good performance the toolkit uses some special C++ techniques such as Template Metaprograms and Expression Templates [Veldhuizen 1995b; 1995a].

In this last decades, several preliminary versions of SparseTool (called SparseLib) in combination with P2MESH [Bertolazzi and Manzini 2002] have been proved to be an effective device for managing sparse matrices in the implementation of numerical solvers for PDEs. Specific software developments concern fluid dynamics simulations of diffusion and transport in porous media using finite volumes [Bertolazzi and Manzini 2004b; 2004a; 2006; 2007; Manzini and Putti 2007; Manzini and Russo 2008], coupling mixed finite elements and finite volumes [Gallo and Manzini 1998b; 1998a], and applying mimetic finite difference methods [Cangiani and Manzini 2008; Beirão da Veiga and Manzini 2007].

The paper is organized as follows: in Section 1, we present the base structure of the toolkit, in Section 2, we display the base classes defining vectors and matrices, in Section 3, we expose the implemented matrix formats and auxiliary routines, in Section 4, we show the potentiality and the application of the toolkit, by discussing some simple applications, and in Section 5 we offer final remarks and conclusions.

2. BASE INTERFACE STRUCTURE

In the spirit of P2MESH the toolkit is contained in a unique header file (`SparseTool.hh`), thus, there is no need of installation or compilation. The toolkit consists of less than 7000 C++ code lines including the DOXYGEN documention [van Heesch. 1997]. The classes are parameterized and are derived from three general container classes

— `VectorBase<Type,V>`

— `Sparse<Type,Matrix>`

— `Preco<Type>`

The generic `Type` can be any standard type (e.g. `float`, `double`) or any user defined type. The classes `VectorBase<Type,V>` and `Sparse<Type,Matrix>` are parameterized with the derived classes, for example, the definition of full vector class `Vector<Type>` or CCOOR matrix `CCoorMatrix<Type>` is as follows

```
template <typedef T>
class Vector : public VectorBase<T, Vector<T> >, public std::vector<T> {
   ....
};

template <typedef T>
class CCoorMatrix : public Sparse<T, CCoorMatrix<T> > {
   ....
};
```

This piece of code, which is compliant to teh C++ standard, implements the *curiously recurring template pattern*, also know as Barton and Nackman [1994] trick. This pattern allows us to avoid the use of virtual functions, thus improving the runtime performance. There are base classes used in the toolkit:

(1) The template class `VectorBase<Type,V>` is the base class for:
— `Vector<Type>` for full vector.
— `VectorSlice<Type>` for subvector or remapping of C-pointers.

(2) The template class `Sparse<Type,Matrix>` is the base class for:
— `SparsePattern` to manage a sparse pattern.
— `CCoorMatrix<Type>` for *Compressed Coordinate Storage* matrix.
— `CRowMatrix<Type>` for *Compressed Rows Storage* matrix.
— `CColMatrix<Type>` for *Compressed Columns Storage* matrix.
— `TridMatrix<Type>` for *Tridiagonal* matrix.

(3) A set of iterative algorithm for the solution of linear system.
— `cg`: for the Conjugate Gradient method [Hestenes and Stiefel 1952].
— `gmres`: for the GMRES algorithm [Saad and Schultz 1986];
— `bicgstab`: for the BiCGSTAB algorithm [van der Vorst 1992] ;
The algorithm needs preconditioner derived from the common class `Preco<Type>`. The following standard preconditioner are defined in SparseTool:
— `IdPreco<Type>` identity matrix preconditioner;
— `DPreco<Type>` the diagonal preconditioner;
— `ILDUpreco<Type>` incomplete *LDU* preconditioner.

In addition to these classes, there is the class `MatrixMarket` to read matrices in Matrix Market format, and the function `Spy` that produce a postscript file with a "silhouette" of the nonzeros pattern for any object derived from `Sparse<Type,Matrix>`.

Remark 2.1. The schemata is a little bit simplified, in fact `SparsePattern` is derived from `SparsBase` and `Sparse<Type>` is derived from `SparseBase`. Moreover there are many other support classes, for example the trait classes [Meyers 2005] and the classes storing intermediate partial expression [Veldhuizen 1995b; 1995a].

2.1 The base class for vectors

The class `VectorBase<Type,V>` is the base class for any vector-like object of SparseTool. If a user want to extend the toolkit with its own vector-like class he or she must inherit from this base class. With `VectorBase<Type,V>` objects it is possible to write a vector expression in a simple way like in Blitz++ [Veldhuizen 1999; 1998] although only a subset af the available operators is overloaded. The operators and functions considered are showed in the following table where `OP` is any of `+`, `-`, `*` or `/` and `f1`$\in$ { `absval`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `cosh`, `sinh`, `tanh`, `sqrt`, `ceil`, `floor`, `log`, `log10` } and `f2` $\in$ { `pow`, `atan2`, `minval`, `maxval` }.

Expression	C-code equivalent `n = a.size()`
`a = s`	`for ( i = 0 ; i < n ; ++i ) a[i] = s ;`
`a OP= s`	`for ( i = 0 ; i < n ; ++i ) a[i] OP= s ;`

Expression	C-code equivalent `n = min(a.size(),b.size())`
`a = b`	`for ( i = 0 ; i < n ; ++i ) a[i] = b[i] ;`
`a OP= b`	`for ( i = 0 ; i < n ; ++i ) a[i] OP= b[i] ;`

`a = b OP s`	`for ( i = 0 ; i < n ; ++i ) a[i] = b[i] OP s ;`
`a = s OP b`	`for ( i = 0 ; i < n ; ++i ) a[i] = s OP b[i] ;`
`a = f1(b)`	`for ( i = 0 ; i < n ; ++i ) a[i] = f1(b[i]) ;`

Expression	C-code equivalent `n = min(a.size(),b.size(),c.size())`
`c = a OP b`	`for ( i = 0 ; i < n ; ++i ) c[i] = a[i] OP b[i] ;`
`c = f2(a,b)`	`for ( i = 0 ; i < n ; ++i ) c[i] = f2(a[i],b[i]) ;`

The template functions `absval`, `minval` and `maxval` correspond to `abs`, `min` and `max` of STL [Glass and Schuchert 1995; Musser and Saini 1996; Plauger et al. 2000]. The `val` suffix is added to avoid conflict with the template functions of STL. Some useful functions are also defined for the class `VectorBase<Type,V>` and are resumed in the following table:

`dot(a,b)` $= \sum_i a_i b_i$	`dist(a,b)` $= \left(\sum_i (a_i - b_i)^2\right)^{1/2}$	`dist2(a,b)` $= \sum_i (a_i - b_i)^2$
`normi(a)` $= \max_i \|a_i\|$	`norm1(a)` $= \sum_i \|a_i\|$	`norm2(a)` $= \left(\sum_i \|a_i\|^2\right)^{1/2}$
`maxval(a)` $= \max_i a_i$	`minval(a)` $= \min_i a_i$	`normp(a,p)` $= \left(\sum_i \|a_i\|^p\right)^{1/p}$
`sum(a)` $= \sum_i a_i$	`prod(a)` $= \prod_i a_i$	

Using the Expression Template technique [Veldhuizen 1995a], complex expressions involving vectors do not produce big temporaries. For example, the following code shows the effect of the Expression Template technique where `n` is the minimum of the size of the vectors involved:

```
a += ((atan2(b,c))/sin(d)-e+3.1)*f ; // is expanded to

for ( i = 0 ; i < n ; ++i )
  a[i] += ((atan2(b[i],c[i]))/sin(d[i])-e[i]+3.1)*f[i] ;
```

The choice of the minimum size in vector operation permits avoiding runtime error if the vector are of different size and simplify the resulting code.

2.2 The base class for sparse matrices

Sparse matrix classes have different internal structures but are derived from a unique base classes `Sparse<Type,Matrix>`. This class is used as a pivot class for matrix/vector mixed operations. It contains minimal information and methods to access elements of the derived classes. To improve the performance, virtual methods are avoided by using the Barton–Nackman trick for the vector and matrix classes . The public methods of this class are listed in the following table:

Method	Description
`Type & operator (i,j)`	Return the reference of the `(i,j)` element of matrix `A`. If the element does not exists an error is issued.
`Type const & value(i,j)`	Return the reference of the `(i,j)` element of matrix `A`. If the element does not exists a reference to a `0` value is returned.
`bool exists(i,j)`	**`true`** if `(i,j)` is a nonzero of the sparse matrix.

`Type const & operator [i]`	The constant reference of the `i`-th element of the internal vector of the nonzeros of the matrix.
`unsigned numRows()`	Row size of the matrix
`unsigned numCols()`	Column size of the matrix
`unsigned minSize()`	min(`numRows()`,`numCols()`)
`unsigned maxSize()`	max(`numRows()`,`numCols()`)
`unsigned nnz()`	Total number of stored values
`void nnz(l,d,u)`	Total number of stored values under the diagonal `l`, on the diagonal `d` and over the diagonal `u`
`bool isOrdered{}`	Return **`true`** is the internal structure is oredered
`void setZero()`	Set to `0` all the nonzeros of the sparse matrix
`void scaleValues(val)`	Multiply by `val` all the nonzeros
`void setRow(nr,val)`	Set to `val` all the nonzeros of the `nr` row
`void setColumn(nc,val)`	Set to `val` all the nonzeros of the `nc` column
`void scaleRow(nr,val)`	Multiply by `val` the `nr` row
`void scaleColumn(nc,val)`	Multiply by `val` the `nc` column

A small number of operators to initialize a matrix or set values on its diagonal is provided by the toolkit. The operators are resumed in the following table

Expression	C-code equivalent `n = A.minSize()`
`A = s`	`for ( i = 0 ; i < n ; ++i ) A(i,i) = s ;`
`A OP= s`	`for ( i = 0 ; i < n ; ++i ) A(i,i) OP= s ;`
Expression	**C-code equivalent** `n = min(A.minSize(),b.size())`
`A = b`	`for ( i = 0 ; i < n ; ++i ) A(i,i) = b[i] ;`
`A OP= b`	`for ( i = 0 ; i < n ; ++i ) A(i,i) OP= b[i] ;`

Elements iterators. To access the nonzeros elements of a sparse matrix, the user can invoke the `operator()` with `exists(i,j)`. This way of looping on elements is very inefficient especially if the matrix is very sparse. To access the nonzeros elements more efficiently in a sequential way, SparseTool uses a simple iterator-like combination of methods. These methods must be defined in the derived classes and are the core of SparseTool.

Method	Looping on elements
`void Begin()`	Initialize iterator
`void Next()`	Advance the iterator to next element
`bool End()`	**`false`** when all elements are iterated
Method	**Accessing elements**
`unsigned row()`	Row index of the actual element
`unsigned column()`	Column index of the actual element
`Type &  value()`	The value of the actual element

If `S` is a sparse matrix derived from the object `Sparse<Type,Matrix>`, the following loop permits accessing all the elements and print all the stored values:

```
for ( S . Begin() ; S . End() ; S . Next() )
  cout << " row    = " << S . row()
       << " column = " << S . column()
       << " value  = " << S . value() << '\n' ;
```

Depending on the internal structure of the sparse matrix, the realization of iterator methods, can be easy or complex to implement The big advantage in using iterators is the abstraction of the sparse matrix, which, for example, simplifies the conversion between formats. In this way, if a user adds its own sparse matrix class derived from `Sparse<Type,Matrix>`, all the classes of the toolkit are able to convert from the user class to the SparseTool classes. Finally, if the user is able to convert to its sparse storage format using only the iterators, then it can import any other matrix storage format of SparseTool.

Matrix multiplication. The base class `Sparse<Type,Matrix>` provides two methods that are used as a building blocks for expressions involving matrix-vector multiplications. These methods implement the multiplication of a matrix by a vector and the multiplication of the transpose matrix by a vector.

Method	Expression equivalent
`A.add_S_mul_M_mul_V( res, s, x )`	`res += s * (A*x)`
`A.add_S_mul_Mt_mul_V( res, s, x )`	`res += s * (A^x)` (transpose multiply)

These operations are implemented by using the iterators previously described so that they are available in all the derived classes. The use of iterators may be inefficient for some storage format. Therefore, these methods are normally overloaded in the derived class to gain a better performance.

Constructor. The classes derived from `Sparse<Type,Matrix>` – with the exception of `TridMatrix<Type>` – have the same constructors. The constructors initialize the sparse matrix structure to an empty sparse matrix or to a sparse matrix whose pattern is a copy of the pattern of another `Sparse<Type,Matrix>` object.

Constructor	Description
`Matrix()`	Create an empty sparse object of 0×0 size
`Matrix(nrow, ncol)`	Create an empty sparse object of `nrow` $\times$ `ncol` size
`Matrix(nrow, ncol, nnz)`	Create an empty parse object of `nrow` $\times$ `ncol` size with preallocated room for `nnz` values
`Matrix(pattern)`	Create a sparse object getting size and allocation information from the object `pattern` of type derived from `Sparse<Type,Matrix>`
`Matrix(pattern,cmp)`	Create a sparse object getting size and allocation information from the object `pattern` of type derived from `Sparse<Type,Matrix>`. Only the index such that `cmp(i,j)==true` are added to the object.

For each constructor, there is a method `resize` having the same argument that reinitializes the `Sparse<Type,Matrix>` object. The comparison method is implemented by using `cmp` a class with overloaded the operator `()` used as a functor [Musser and Saini 1996; Plauger et al. 2000]. For example to select the lower triangular part of a sparse matrix the comparator can be:

```
struct lower_triangular {
  bool operator () ( unsigned i, unsigned j ) const { return i > j ; }
}
```

In the constructors of the previous Table, `nnz` is the estimated size of nonzeros for preallocating the memory of the sparse object. If not specified, the default value is 100, however, if necessary this size is enlarged transparently to the user by internal reallocation of the memory. The user can gain efficiency by preallocating in advance enough storage (`nnz`) for the pattern hence avoiding costly memory reallocation.

2.3 The base class for preconditioners

The class `Preco<P>` is the base class for the implementation of a preconditioner. Preconditioners are widely used to accelerate convergence of many iterative methods and formally are matrices that approximate the matrix to be preconditioned and easy to invert. Using a preconditioner formally require a matrix-vector multiplication so that only this last operation is defined in the class. The base class has only two methods: a method for the construction of the preconditioner and a method for the application of the preconditioner to a vector.

Method	Description
`P.build(M)`	Build the preconditioner for the object `M` of type derived from `Sparse<T,Matrix>`
`P.assPreco(res,v)`	Apply the preconditioner to the vector `v`. This method is called from `VectorBase<T,V>` object `res` in expression like `res = P*v`

2.4 Operator overloading and expression templates

To simplify the use of the toolkit, a number of mathematical operators are overloaded thus allowing expression involving matrix and vector in a human readable way. For expressions involving only scalars and vectors, the expression templates technique is used to avoid temporaries and to obtain the peak performance [Veldhuizen 1995b; 1995a; 1998]. The same technique can be used in principle for any expressions involving vectors and matrices, however, in this case it become difficult to parse and optimize the resulting software implementation. For example, it is impossible to implement efficiently an expression like

```
res = A * (B * c + d) ;
```

where `A` and `B` are matrices and `res`, `c` and `d` are vectors, without the introduction of temporary vectors. By introducing the temporary vector “e” we can split the expression as follows

```
e = B * c + d ; res = A * e ;
```

and this expression can be parsed in an efficient way by SparseTool.

To simplify the toolkit and maintain good performance, a reduced number of expressions involving matrix-vector products are supported while matrix-matrix products are not considered. The expressions supported are listed in the following table

Expression	Description	Expression	Description
`r = A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{A}\boldsymbol{x}$	`r OP= A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{A}\boldsymbol{x}$
`r = s*(A*x)`	$\boldsymbol{r} \leftarrow s\boldsymbol{A}\boldsymbol{x}$	`r OP= s*(A*x)`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $s\boldsymbol{A}\boldsymbol{x}$
`r = b + A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{b} + \boldsymbol{A}\boldsymbol{x}$	`r OP= b + A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} + \boldsymbol{A}\boldsymbol{x}$
`r = b - A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$	`r OP= b - A*x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$
`r = b + s*(A*x)`	$\boldsymbol{r} \leftarrow \boldsymbol{b} - s\boldsymbol{A}\boldsymbol{x}$	`r OP= b + s*(A*x)`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} - s\boldsymbol{A}\boldsymbol{x}$
`r = A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$	`r OP= A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{A}^T\boldsymbol{x}$
`r = s*(A^x)`	$\boldsymbol{r} \leftarrow s\boldsymbol{A}^T\boldsymbol{x}$	`r OP= s*(A^x)`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $s\boldsymbol{A}^T\boldsymbol{x}$
`r = b + A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{b} + \boldsymbol{A}^T\boldsymbol{x}$	`r OP= b + A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} + \boldsymbol{A}^T\boldsymbol{x}$
`r = b - A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A}^T\boldsymbol{x}$	`r OP= b - A^x`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} - \boldsymbol{A}^T\boldsymbol{x}$
`r = b + s*(A^x)`	$\boldsymbol{r} \leftarrow \boldsymbol{b} - s\boldsymbol{A}^T\boldsymbol{x}$	`r OP= b + s*(A^x)`	$\boldsymbol{r} \leftarrow \boldsymbol{r}$ `OP` $\boldsymbol{b} - s\boldsymbol{A}^T\boldsymbol{x}$
`OP` is one of `+` `-`			

Although in limited number, these expressions should be enough in most of applications eventually splitting expressions in two or more subexpressions.

3. THE CLASSES FOR LINEAR ALGEBRA IMPLEMENTATIONS

3.1 Vector and matrix formats

Class `Vector<Type>`. This class extends the STL class `vector<Type>` so that all the algorithms and functionality of the class `std::vector<Type>` are available [Plauger et al. 2000] in addition to the methods and operators of the class `VectorBase<Type,V>`.

Class `VectorSlice<Type>`. This class is used to partition vectors of the class `Vector<Type>` or remap a C-array into a `VectorBase<Type,V>` object. For example, the following code shows an example of the use of `VectorSlice<Type>`

```
Vector<double> v(100) ; double w[100] ; VectorSlice a, b, c, d ;
a.slice( v, 0, 50 ) ; b.slice( v, 50, 100 ) ;
c.slice( a, a+50 )  ; d.slice( a+50, a+100 ) ;
```

In the previous code fragment, vectors `a`, `b`, `c`, `d` inherit all the operators and functions previously described for class `VectorBase<Type,V>`. Notice that this class does not allocate memory so that the object pointed by a `VectorSlice<Type>` instance must not be release or resized to avoid unpredictable errors.

`SparsePattern`. This class implement the *Compressed Coordinate* structure which consists of two vectors of unsigned integers that contain the coordinate of nonzero elements. We call `I` the vector that stores the first coordinate and `J` the vector that stores the second coordinate. For example, the following 3×7 sparse matrix pattern

$$
\boldsymbol{S} = \begin{bmatrix} * & * & & & & & * \\ & * & * & & & & \\ & & & * & * & & \end{bmatrix}, \qquad \begin{array}{c|ccccccc} \mathtt{I} & 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ \hline \mathtt{J} & 0 & 1 & 6 & 1 & 2 & 3 & 4 \end{array} \tag{1}
$$

is stored in the vectors `I` and `J`. Notice that the index starts from 0 following the C convention. The class `SparsePattern` manages such a structure in a simple way for the user.

To complete the basic description of the `SparsePattern`, we list the remaining methods in the following table

Method	Description
`SparsePattern & insert(i,j)`	insert a nonzero at position `(i,j)`
`void internalOrder()`	order the pattern and eliminate duplicate elements
`bool isOrdered()`	return **`true`** if the pattern is ordered

For example pattern $\boldsymbol{S}$ in equation (1) can be constructed as

```
SparsePattern sp_for_S(3,7) ;
sp_for_S . insert(0,0) . insert(0,1) . insert(0,6) . insert(1,1) ;
sp_for_S . insert(1,2) . insert(2,3) . insert(2,4) ;
Sp_for_S . internalOrder() ;
```

Notice that the method `insert` return a reference of `SparsePattern` so that it permits a multiple call with the same object. The method `internalOrder()` reorder the nonzero elements eliminating duplicate elements. The order is such that if $k_1 \leq k_2$ we have one of the two following cases

(1) $\texttt{I}(k_1) < \texttt{I}(k_2)$

(2) $\texttt{I}(k_1) = \texttt{I}(k_2)$ and $\texttt{J}(k_1) \leq \texttt{J}(k_2)$

This order is the column-oriented scheme that has been widely used in sparse matrix computation. This method is considered when we use `SparsePattern` to construct a sparse matrix or before using random access to the sparse matrix elements.

`CCoorMatrix<Type>`. This class implements a *Compressed Coordinate* storage sparse scheme. It is essentially the class `SparsePattern` with in addition the vector `A` that stores the nonzero values. For example the following 3×7 sparse matrix

$$\boldsymbol{M} = \left[\begin{array}{c:c:c:c:c:c:c} 1 & 2 & & & & & 9 \\ \hdashline & -1 & 0 & & & & \\ \hdashline & & & 3 & 4 & & \end{array}\right], \qquad \begin{array}{c|ccccccc} \texttt{I} & 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ \hline \texttt{J} & 0 & 1 & 6 & 1 & 2 & 3 & 4 \\ \hline \texttt{A} & 1 & 2 & 9 & -1 & 0 & 3 & 4 \end{array} \qquad (2)$$

is stored in the vectors `A`, `I` and `J`. The methods of the class `CCoorMatrix<Type>` are the same of the class `SparsePattern` with the difference that `insert(i,j)` returns the reference of the inserted element so that it can be initialized. For example matrix $\boldsymbol{M}$ of equation (2) can be constructed with the following piece of code:

```
CCoorMatrix<double> A(6,7) ;
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.internalOrder() ;
```

When we execute the `internalOrder()` method, if a pair of indices `(i,j)` occurs more than once the corresponding values are accumulated. Notice that the sparse pattern of $\boldsymbol{M}$ is the same of $\boldsymbol{S}$ of equation (1) so that the following commands produce the same results of the previous code:

```
CCoorMatrix<double> A(sp_for_S) ;
A(0,0) = 1 ; A(0,1) = 2 ; A(0,6) = 9 ; A(1,1) = -1 ; A(1,2) = 0 ;
A(2,3) = 3 ; A(2,4) = 4 ; A . internalOrder() ;
```

Therefore, we can use a sparse pattern (from any derived class from `Sparse<Type,Matrix>`) to allocate and initialize a sparse matrix.

`CRowMatrix<Type>` and `CColMatrix<Type>`. These classes implement the *Compressed Rows* (CROW) and *Compressed Columns* (CCOL) sparse storage scheme, respectively. The CROW format substitutes the vector column coordinate `I` with a row pointer `R` while the CCOL format substitutes vector row coordinate `J` with a column pointer `C`. For example the sparse matrix (2) can be stored in the two previous formats as follows

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

C	0	1	3	4	5	6	6	7
I	0	0	1	1	3	3	0	
A	1	2	-1	0	3	4	9	

Matrices of type CROW and CCOL cannot be constructed incrementally (so `insert`, `isOrdered` and `internalOrder` method are not available) but need a pattern from a `Sparse<Type,Matrix>` object to be initialized. For example, the matrix $\boldsymbol{M}$ of equation (2) can be built using the CROW storage as follows:

```
CRowMatrix<double> A(sp_for_S) ;
A(0,0) = 1 ; A(0,1) = 2 ; A(0,6) = 9 ; A(1,1) = -1 ; A(1,2) = 0 ;
A(2,3) = 3 ; A(2,4) = 4 ;
```

`TridMatrix<Type>`. This class is used to manage tridiagonal matrices. It consists of 3 vector `L`, `D`, `U` for the three diagonals. For example the following tridiagonal matrix can be stored as follows:

$$\boldsymbol{T} = \begin{bmatrix} 2 & -1 & & \\ 1 & 2 & -2 & \\ & 2 & 2 & -3 \\ & & 3 & 2 \end{bmatrix}$$

L	D	U
1	2	-1
2	2	-2
3	2	-3
	2	

To define a `TridMatrix<Type>` class we can use one of the constructors listed in the following table:

Constructor	Description
`TridMatrix()`	create an empty 0 size tridiagonal matrix
`TridMatrix(n)`	create an empty `n` × `n` tridiagonal matrix
`TridMatrix(TridMatrix const & T)`	create a copy of tridiagonal matrix `T`

For `TridMatrix<Type>` objects, the operator `/` is overloaded to define the solution of a tridiagonal system. For example:

```
TridMatrix<double> T; Vector<double> b, x ;
...
x = b / T ; // solve the system T*x == b
```

Accessing the internal structure. Sometimes it is useful to access to the internal structure of a sparse matrix. This occurs when SparseTool is interfaced with a high performance direct solver or, in general, when is used with other sparse matrix libraries. To access to the internal structure, the following methods are available:

Method	Description
`Vector<Type>            & getA()`	return the pointer to the values internal vector `A`
`Vector<unsigned> const & getI()`	only for CCOOR and CCOL return the rows index vector
`Vector<unsigned> const & getJ()`	only for CCOOR and CROW return the column index vector
`Vector<unsigned> const & getR()`	only for CROW return the pointer to the rows index displacement
`Vector<unsigned> const & getC()`	only for CCOL return the pointer to the column index displacement

Preconditioners. SparseTool furnishes only two standard preconditioners: the diagonal preconditioner `DPreco<Type>` and the incomplete *LDU* preconditioner `ILDUPreco<Type>`. The *LDU* preconditioner has the additional builder method: `build(A,pattern)`, where the pattern of the incomplete factorization is loaded from the object `pattern` and not from the matrix `A`. For the preconditioner only the `/` operator is overloaded and means the application of the preconditioner.

Expression	Description
`r = x / P`	$\boldsymbol{r} \leftarrow \boldsymbol{P}^{-1}\boldsymbol{x}$

Sometimes we do not want to use a preconditioner. In such a case, the `IdPreco` class allows to use the identity matrix as preconditioners.

3.2 Iterative solvers

SparseTool implements a minimal set of iterative solvers for linear systems. Two solvers applies to general unsymmetric matrices and one to simmetric and positive definite matrices. The iterative solvers are `cg`, implementing the Conjugate Gradient method, `gmres`, implementing the GMRES algorithm, and `bicgstab`, implementing the BiCGSTAB algorithm. The parameters for calling the iterative solvers are summarized in the following table:

Method
`res = cg(A, b, x, P, epsi, maxIter, iter, pstream = NULL)`
`res = bicgstab(A, b, x, P, epsi, maxIter, iter, pstream = NULL)`
`res = gmres(A, b, x, P, epsi, maxSiter, maxIter, iter, pstream)`

`res`	the infinity norm of the last residual
`A`	input: coefficient matrix
`b`	input: rhs of the linear system
`x`	input/output: guess and the solution of the linear system
`P`	input: a preconditioner for `A`
`epsi`	input: admitted tolerance
`maxIter`	maximum number of allowable iteration
`maxSiter`	for `gmres` is the maximum number of iteration before restarting
`iter`	performed iteration

`stream`	a pointer of object of type `ostream` if not `NULL` used for messages

3.3 Matrix Market interface

A standard way to store and exchange large sparse matrices is to save the matrices accordingly to some *file exchange format*. For sparse matrices, there are two widely used formats: the Harwell-Boeing (HB) Sparse Matrix Format [Duff et al. 1989] and Matrix Market (MM) Sparse Matrix Format [Boisvert et al. 1997]. The HB format is strongly dependent on the FORTRAN programming language and is difficult to manage in a different language without using a sophisticated parser. The alternative is to use a dedicated FORTRAN routine to manage such a format. The design of SparseTool is based on a unique header file, i.e. `SparseTool.hh`, that contains the whole library so that the HB format is not supported. The MM format is easier to manage; therefore, a simple support is included in the toolkit through the class `MatrixMarket`. In any case, free software product are available to convert from one format to another one [Demmel and Yelick 1999; Boisvert et al. 1997]. The methods to read a MM file and load it into `Sparse<Type,Matrix>` object are the following:

Method	Description
`void read(fname)`	read the file in MM format specified by the string `fname`. The result is stored in the class instance.
`void read(istream & s)`	read the file in MM format from the opened stream `s`. The result is stored in the class instance.
`void print(ostream & s)`	print to the stream object `s` some information about the last load matrix.
`void load(V)`	copy the last loaded array to the vector `V`
`void load(pattern)`	copy the pattern of the last loaded matrix in the `SparsePattern` object `sparse`
`void load(M)`	copy the last loaded matrix in the object `M` which is derived from `Sparse<Type,Matrix>` class. I.e. it can be for example a `CCoorMatrix<Type>` object.

The following code shows how `MatrixMarket` should be used:

```
MatrixMarket mm ; // define the object mm to manage Matrix Market file
CCoorMatrix<double> A ;      // an empty sparse CCOOR matrix
SparsePattern sp ;           // an empty sparse pattern
mm . read("hor__131.mtx") ;  // read matrix and store in mm object
mm . load_pattern( sp ) ;    // extract the pattern
mm . load_matrix( A ) ;      // copy loaded matrix in sparse matrix A
```

The class `MatrixMarket` is not a `template` class because the value type of the nonzeros is specified by the format. The available type in MM format are `int`, `double` and `complex<double>`. The representation of sparse matrices in MM format is of type CCOOR, so that `MatrixMarket` stores in this form the loaded matrix. Full matrices are stored in column major order. Sometimes it can be useful to access directly the structure of the loaded matrix with the following methods:

Method	Description
`unsigned numRows()`	number of rows of loaded matrix
`unsigned numCols()`	number of columns of loaded matrix
`unsigned nnz()`	total number of nonzeros
`unsigned coor_type()`	type of storage 0 = by coordinate, 1 = full matrix in column major order
`unsigned value_type()`	value type: 0 = no values, only pattern, 1 = values of type `int`, 2 = values of type `double`, 3 = values of type `complex<double>`.
`unsigned matrix_type()`	0 = general matrix, 1 = symmetric matrix, 2 = skew symmetrix matrix, 3 = Hermitian matrix.
`Vector<unsigned> const & getI()`	return the vector of the row index
`Vector<unsigned> const & getJ()`	return the vector of the column index
`Vector<unsigned> const & getAi()`	return the vector of nonzeros for matrices with `int` type entries
`Vector<unsigned> const & getAr()`	return the vector of nonzeros for matrices with `double` type entries
`Vector<unsigned> const & getAc()`	return the vector of nonzeros for matrices with `complex<double>` type entries

To write a sparse pattern or matrix in MM format the following functions are available:

—`MatrixMarketSaveToFile(fname,A,vType,mType)`: save the sparse matrix `A` to file with name `fname` and with value type `vType` with matrix type `mType`.

—`MatrixMarketSaveToFile(fname,S,mType)`: save the sparse pattern `S` to file with name `fname` with matrix type `mType`.

3.4 Spy command

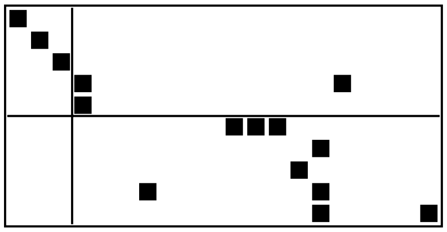
Graphics visualization of the structure of a sparse matrix is often a useful tool. The function `Spy`, writes a postscript file for the "silhouette" of the nonzero structure of `A`. The result is similar to the MATLAB `spy` command [Gilbert et al. 1992].

Methods	
`void Spy(fname,M,size)`	`void Spy(fname,M,size,rl,cl)`
`void Spy(stream,M,size)`	`void Spy(stream,M,size,rl,cl)`
Parameters	
`string const & fname`	the name of the output file
`Sparse<Type,Matrix> const & M`	pattern to be drawn
`double size`	dimension of the plot in *cm*
`vector<unsigned> const * rl`	vector containing the index of horizontal line to be drawn. If for example `(*rl)[0]` contains 5 then a line between row 4 and 5 is drawn.
`vector<unsigned> const * rc`	vector containing the index of vertical line to be drawn.

The command accepts any object derived from `Sparse<Type,Matrix>`, so that any derived class can plot its pattern by the command `Spy`. For example the following code

```
#include "SparseTool.hh"
using namespace SparseToolLoad ; using namespace std ;

int main() {
  SparsePattern sp(10,20) ;
  sp.insert(0,0)  .insert(1,1) .insert(2,2) .insert(3,3) .insert(3,15) ;
  sp.insert(4,3)  .insert(5,10).insert(5,11).insert(5,12).insert(6,14) ;
  sp.insert(7,13) .insert(8,6) .insert(8,14).insert(9,14).insert(9,19) ;
  sp.internalOrder() ;
  vector<unsigned> rc, rr ; rc . push_back(3) ; rr . push_back(5) ;
  Spy( "S.eps", sp, 12.0, &rr, &rc ) ;
  return 0 ;
}
```

produces the file `S.eps`, which contains the following picture:



The command `Spy` is the translation in C++ of the routine `pspltm` of SPARSKIT with minor adjustment.

4. APPLICATION EXAMPLES

4.1 Some examples of usage

The following two examples illustrate the simplicity of the use of SparseTool.

Elliptic Equation. We want to approximate by finite difference the solution of the following problem:

$$\begin{aligned} \nabla^2 u(x,y) &= 1, \qquad (x,y) \in \Omega = (0,1)\times(0,1), \\ u(x,y) &= 0, \qquad (x,y) \in \partial\Omega. \end{aligned} \tag{3}$$

By setting $h = 1/n$ and $x_i = y_i = i\,h$, the finite differences approximation u_{ij} of the exact solution $u(x_i, y_j)$ satisfies the following equations:

$$\begin{aligned} u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} &= h^2, \qquad i,j = 1,2,\ldots,n-1 \\ u_{0,j} = u_{n,j} = u_{i,0} = u_{i,n} &= 0, \qquad i,j = 1,2,\ldots,n-1 \end{aligned}$$

This constitutes a sparse linear system in the unknown u_{ij}. The solver is contained in the 22 lines of the following code.

```
#include "SparseTool.hh"
using namespace ::SparseToolLoad ; using namespace ::std;
```

```
3  int main() {
4    unsigned            iter, n = 10, neq = (n-1)*(n-1) ;
5    CCoorMatrix<double> A(neq,neq,5*neq) ;
6    Vector<double>      b(neq), x(neq) ;
7    ILDUPreco<double>   P ;
8    b = 1.0/n*n ;
9    for ( unsigned nr=0 ; nr < neq ; ++nr ) {
10     unsigned i = nr % (n-1), j = nr / (n-1) ;
11     A . insert(nr,nr) = -4 ;
12     if ( i > 0   ) A . insert(nr,nr-1)     = 1 ;
13     if ( i < n-2 ) A . insert(nr,nr+1)     = 1 ;
14     if ( j > 0   ) A . insert(nr,nr-(n-2)) = 1 ;
15     if ( j < n-2 ) A . insert(nr,nr+(n-2)) = 1 ;
16   }
17   A . internalOrder() ; P . build(A) ;
18   double res = bicgstab( A, b, x, P, 1E-10, 100u, iter, &cout );
19   b -= A*x ;
20   cout << "Verify Residual = " << normi(b) << '\n' ;
21   return 0;
22 }
```

— Line 1 includes the SparseTool library.

— Line 2 loads the toolkit function, the classes contained in `SparseToolDefs` namespace, and the standard namespace.

— Line 5 instantiates a `neq` $\times$ `neq` CROW sparse matrix class with room for $5 \cdot$ `neq` preallocated nonzeros.

— Line 6 instantiates two full vector of size `neq`. `b` will be the r.h.s of the linear system, `x` will be its solution.

— Line 7 instantiates an uninitialized object of the incomplete *LDU* preconditioner class.

— Line 8 initializes the r.h.s vector `b` to $1/h^2$.

— Lines 9–16 performs a loop on the internal nodes of the mesh and sets the matrix coefficients.

   — Line 10 gets the `(i,j)`-node coordinates from the equation number.

   — Line 11 sets a diagonal element of the matrix `A`.

   — Lines 12 – 15 sets the non-diagonal element if this is not on the boundary.

— Line 17 consolidates the matrix `A` and builds the sparse *LDU* preconditioner.

— Line 18 calls the `bicgstab` solver.

— Line 19 computes the true residual for checking purpose.

The output produced by the code is listed here:

```
iter = 1 residual = 0.986534
iter = 2 residual = 0.0629651
iter = 3 residual = 0.00287048
iter = 4 residual = 0.000128241
iter = 5 residual = 5.94365e-06
iter = 6 residual = 7.34315e-07
iter = 7 residual = 9.54058e-08
iter = 8 residual = 7.98651e-09
iter = 9 residual = 1.19073e-10
iter = 10 residual = 5.46827e-11
Verify Residual = 8.24221e-11
```

This code is incomplete, for example, the solution is not saved and our aims is only to show the key idea. However, a full program solving problem (3) can be written with less than 100 code lines.

Least square problem. Minimize the norm of an overdetermined sparse linear system:

$$
\boldsymbol{x} = \arg\min_{z} \|\boldsymbol{A}\boldsymbol{z} - \boldsymbol{b}\|_2 \quad \text{where} \quad \boldsymbol{A} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & & \\ & & 1 & 2 \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}
$$

The solution of this problem is the solution of the linear system:

$$
\boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T \boldsymbol{b},
$$

which can be written as the solution of the following augmented linear system [Gilbert et al. 1992]:

$$
\begin{pmatrix} \boldsymbol{I} & \boldsymbol{A} \\ \boldsymbol{A}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{r} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{pmatrix}
$$

The coefficients matrix is symmetric but not positive definite. For this reason, the Conjugate Gradient method cannot be applied. Instead, the BiCGSTAB method with an incomplete *LDU* preconditioner works pretty well. A possible implementation with SparseTool uses iterators to build the block matrix and `VectorSlice<double>` to access the solution. A sample code is depicted below.

```
#include "SparseTool.hh"
#include <fstream>
using namespace SparseToolLoad ; using namespace std ;

int main() {
  unsigned i, iter, nr = 7, nc = 4, nnz = 12 ;
  unsigned I[] = { 0, 4, 5, 1, 4, 5, 2, 4, 6, 3, 4, 6 } ;
  unsigned J[] = { 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3 } ;
  unsigned V[] = { 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2 } ;

  CCoorMatrix<double> A(nr,nc),   M(nr+nc,nr+nc) ;
  Vector<double>      rhs(nr+nc), X(nr+nc), bf1(nr), bf2(nc)  ;

  for ( i = 0 ; i < nnz ; ++i ) A . insert(I[i],J[i]) = V[i] ;
  A . internalOrder() ;

  for ( A . Begin() ; A . End() ; A . Next() ) {
    unsigned i = A . row(), j = A . column() ;
    M . insert(i,j+nr) = A . value() ;
    M . insert(j+nr,i) = A . value() ;
  }
18   for ( i = 0  ; i < nr    ; ++i ) M . insert(i,i) = 1 ;
19   M . internalOrder() ;

20   VectorSlice<double> b1, b2, r, x ;
21   b1 . slice( rhs, 0, nr ) = 1 ; b2 . slice( rhs, nr, nc+nc ) = 0 ;
22   r  . slice( X,   0, nr ) = 0 ; x  . slice( X,   nr, nr+nc ) = 0 ;

23   ILDUPreco<double> P ; P . build(M) ;
24   double res = bicgstab( M, rhs, X, P, 1E-15, 100u, iter, &cout );

25   bf1 = b1 - (A*x) ; bf2 = A^bf1 ;
26   cout << "solution = " << x << "\nresidual = " << normi(bf2) << '\n' ;
27   return 0 ;
28 }
```

— Lines 6–8 are the array with the matrix $\boldsymbol{A}$ in CCOOR format.
— Lines 11–12 build matrix $\boldsymbol{A}$ from the C array `I`, `J` and `V`.
— Lines 13–20 build coefficients block matrix from sparse matrix $\boldsymbol{A}$.
— Lines 20–22 map vectors `b1`, `b2`, `r`, `x` to the corresponding block of the partitioned system.
— Line 23 defines and builds the incomplete *LDU* preconditioner.
— Line 24 solves the linear system applying BiCGSTAB.
— Line 25 checks the residual $\boldsymbol{A}^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})$.

The output produced by the code is listed here:

```
iter = 1 residual = 0.502522
iter = 2 residual = 0.00687773
iter = 3 residual = 5.17208e-15
iter = 4 residual = 1.03036e-16
solution = [0.416667 , 0.333333 , 0.416667 , 0.333333]
residual = 4.44089e-16
```

4.2 Extending the toolkit

The toolkit is very flexible and can be easily extended. To show how this can be done, we discuss the interface with SuperLU and UMFPACK. The original interface are hidden in the classes `UMF` and `SuperLU` that has only two public methods aslisted in the table:

Methods	
`int load(A)`	load matrix `A` to the solver class `UMF` or `SuperLU` and perform sparse *LU* factorization. Matrix `A` should be of type `CColMatrix<double>`. If `A` is derived from `Sparse<double,MT>` the a conversion is done intenally. If some error occur the method return a value different from `0`.
`int solve(b,x,t)`	solve the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with matrix `A` previously loaded. If the boolean `t=true` then solve $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. In some error occur the method return a value different from `0`. If `t` is not specified it is set to `false`.

The following code sketches how to use the `UMF` class

```
1  #include "SparseTool.hh"
2  #include "SparseToolUmf.hh"
3  using namespace SparseToolLoad ; using namespace std ;

4  int main() {
5    MatrixMarket mm ; UMF umf ; CCoorMatrix<double> A ;
6    Vector<double> x, rhs, resid ;

7    mm . read( "af23560.mtx" ) ; mm . load( A ) ;

8    unsigned n = A . numRows() ;
9    x . resize( n ) ; rhs . resize( n ) ; residual . resize( n ) ;
10   rhs = 1 ;

11   umf . load(A) ; umf . solve(rhs,x) ;
12   residual = rhs - A*x ;

13   cout << "\nresidual = " << normi( residual ) << '\n' ;
14   return 0 ;
15 }
```

— Lines 1 includes the SparseTool toolkit.

— Lines 2 includes the interface extension for UMFPACK.

— Lines 7 reads the matrix `af23560.mtx` in Matrix Market format and loads it into CCOOR matrix `A`.

— Lines 8–10 initializes work vectors and r.h.s.

— Lines 11 factorizes and solves the linear system using UMFPACK.

— Line 12 computes the residual for checking purpose.

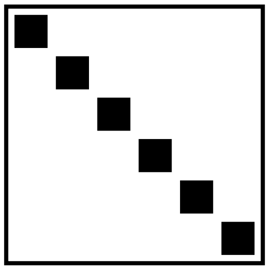
Extending the toolkit is thus easy but require the knowledge of C++ and the sparse matrix format as in very simple example based on the class `DiagMatrix<Type>`, which is a possible implementation of a sparse diagonal matrix type.

```
#include "SparseTool.hh"
namespace SparseToolDefs {
  template <typename T>
  class DiagMatrix : public Sparse<T,DiagMatrix<T> > {
  public:
    typedef Sparse<T,DiagMatrix<T> > SPARSE ; typedef T valueType ;
  private:
    Vector<valueType> D ; mutable indexType ipos ;
  public:
    DiagMatrix(void) {} ;
    DiagMatrix( indexType n ) { resize(n) ; }

    DiagMatrix<T> & operator = (DiagMatrix<T> const & M)
    { D = M.D ; return *this ; }

    void resize( indexType n )
    { D.resize(n+1) ; D=0 ; SPARSE::setup(n,n) ;
      SPARSE::sp_diag_nnz = SPARSE::sp_nnz = n ; }
17      valueType const & value( indexType i, indexType j ) const
18      { return i==j ? D(i) : D(SPARSE::sp_nnz) ; }

19      valueType const &
20      operator () ( indexType i, indexType j ) const { return D(i) ; }

21      valueType &
22      operator () ( indexType i, indexType j ) { return D(i) ; }

23      // ITERATOR
24      void Begin (void) const { ipos = 0 ; }
25      void Next  (void) const { ++ipos ; }
26      bool End   (void) const { return ipos < SPARSE::sp_nrows ; }

27      indexType         row    (void) const { return ipos ; }
28      indexType         column (void) const { return ipos ; }
29      valueType const & value  (void) const { return D(ipos) ; }

30      template <typename VR, typename VB> void
31      add_S_mul_M_mul_V( VectorBase<T,VR> & res, T const & s,
32                         VectorBase<T,VB> const & x ) const
33      { res += s * D * x ; }

34      template <typename VR, typename VB> void
35      add_S_mul_Mt_mul_V( VectorBase<T,VR> & res, T const & s,
36                          VectorBase<T,VB> const & x ) const
37      { res += s * D * x ; }
38    } ;
39    SPARSELIB_MUL_STRUCTURES(DiagMatrix)
40  }

41  namespace SparseToolLoad { using SparseToolDefs::DiagMatrix ; }
42  using namespace SparseToolLoad ; using namespace std ;

43  int main() {
44    DiagMatrix<double> DM(6) ; Vector<double> a(6), b(6) ;
45    DM(0,0) = 1 ; DM(1,1) = 2 ; DM(2,2) = 3 ;
46    DM(3,3) = 4 ; DM(4,4) = 5 ; DM(5,5) = 6 ;
47    b = 1 ; a = b - DM * b ;
48    cout << DM << "\na=" << a << '\n' ;
49    Spy( "D.eps", DM, 12.0 ) ;
50    return 0 ;
51  }
```

As one can notice, the class `DiagMatrix` inherits from the base class `Sparse` and by itself thorough the mechanism of the Barton and Nackman trick. In this class, the operator `()` is defined in lines 19–22 and the iterator is defined in lines 24–29. Lines 30–37 redefine the methods `add_S_mul_M_mul_V` and `add_S_mul_Mt_mul_V`. These methods are already defined in the base classes by the use of the iterator. Normally, this methods should be redefined for performance reason. The macro in line 33 defines the glue for the use of the class in vector/matrix expressions. The code in line 43–51 shows an example of use of this class, which produces the following output:

```
size    = 6 x 6
nnz     = 6  (diag=6, lower=0, upper=0)
ordered = YES
(0,0) = 1
(1,1) = 2
(2,2) = 3
(3,3) = 4
(4,4) = 5
(5,5) = 6

a=[0 , -1 , -2 , -3 , -4 , -5]
```

figure `D.eps`:



4.3 Performance tests

To check performance, a simple comparison with Sparse BLAS and SPARSKIT is done. The test is performed by comparing the computational time of a vector/matrix product. The matrices are taken from the Matrix Market collection while the test are performed on a 2.16 GHz Intel Core Duo. All the tests code are compiled using the Intel Fortran and C++ Compiler version 10.1 with maximum optimization level. The results are shown in the following table, where numbers are the estimated millions of double precision floating point multiplications by second.

Matrix	CCOOR	CCOL	CROW	Sparse BLAS	SPARSKIT
`hor__131.mtx`	189	202	189	213	259
`af23560.mtx`	150	202	212	130	227
`s3dkt3m2.mtx`	159	218	228	136	253
`plat1919.mtx`	182	250	347	228	304

These numbers are only indicative but they show that SparseTool has good average performance. In any case if a user needs peak performance on a specific architecture, a specialized sparse matrix class can be derived.

5. CONCLUSIONS

In this paper we presented SparseTool, an Object-Oriented interface software designed for handling sparse matrices. One of the most important features of SparseTool is its simplicity of usage and extensibility. Although small, this toolkit can be used standalone to develop non trivial code for sparse matrix manipulation and solving sparse linear systems. The use of Expression Templates and static polymorphism permits obtaining good performance while maintaining an intuitive user interface. SparseTool easily link to other high performance sparse matrix libraries such as SuperLU or UMFPACK, therefore, it can be used as a glue with other high performance sparse matrix libraries.

REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software 30,* 3, 381–388.

Anderson, E., Anderson, E., Bai, Z., Bai, Z., Bischof, C., Bischof, C., Demmel, J., Demmel, J., Dongarra, J., Dongarra, J., Croz, J. D., Croz, J. D., Greenbaum, A., Hammarling, S., Hammarling, S., McKenney, A., Ostrouchov, S., Ostrouchov, S., Sorensen, D., and Sorensen, D. 1992. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

Barton, J. J. and Nackman, L. R. 1994. *Scientific and engineering C++*. Addison-Wesley.

Beirão da Veiga, L. and Manzini, G. 2007. An a-posteriori error estimator for the mimetic finite difference approximation of elliptic problems. To appear in Int. J. Numer. Methods Engrg.

Bertolazzi, E. and Manzini, G. 2002. Algorithm 817 P2MESH: generic object-oriented interface between 2-D unstructured meshes and FEM/FVM-based PDE solvers. *ACM Trans. Math. Softw. 28,* 1, 101–132.

Bertolazzi, E. and Manzini, G. 2004a. A cell-centered second-order accurate finite volume method for convection-diffusion problems on unstructured meshes. *Mathematical Models and Methods in Applied Sciences 14,* 8, 1235–1260.

Bertolazzi, E. and Manzini, G. 2004b. Limiting strategies for polynomial reconstructions in the finite volume approximation of the linear advection equation. *Applied Numerical Mathematics 49,* 3–4, 277–289.

Bertolazzi, E. and Manzini, G. 2006. A second-order maximum principle preserving finite volume method for steady convection-diffusion problems. *SIAM Journal on Numerical Analysis 43,* 5, 2172–2199.

Bertolazzi, E. and Manzini, G. 2007. On vertex reconstructions for cell-centered finite volume approximations of 2-D anisotropic diffusion problems. *Mathematical Models and Methods in Applied Sciences 17,* 1, 1–32.

Boisvert, R. F., Pozo, R., Remington, K., Barrett, R. F., and Dongarra, J. J. 1997. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*. Chapman & Hall, Ltd., London, UK, UK, 125–137. `http://math.nist.gov/MatrixMarket/`.

Cangiani, A. and Manzini, G. 2008. Flux reconstruction and solution post-processing in mimetic finite difference methods. *Comput. Methods Appl. Mech. Engrg. 197/9-12*, 933–945.

Davis, T. A. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software 30,* 2, 196 – 199.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004. Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 377–380.

Demmel, J. 1989. LAPACK: A portable linear algebra library for supercomputers. In *IEEE Control Systems Society Workshop on Computer-Aided Control System Design,*. IEEE.

Demmel, J. and Yelick, K. 1999. Berkley Benchmarking and Optimization Group. `http://bebop.cs.berkeley.edu/`.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications 20,* 3, 720–755.

Dongarra, J., Lumsdaine, A., Pozo, R., and Remington., K. 1994. A sparse matrix library in c++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*. 214–218.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. 1998. *Numerical linear algebra for high-performance computers*. Software, Environments, and Tools, vol. 7. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

Dongarra, J. J. and Walker, D. W. 1995. Software libraries for linear algebra computations on high performance computers. *SIAM Rev. 37,* 2, 151–180.

Duff, I. S., Duff, I. S., Reid, J. K., and Reid, J. K. 1996. The design of ma48: a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Softw. 22,* 2, 187–226.

Duff, I. S., Grimes, R. G., and Lewis, J. G. 1989. Sparse matrix test problems. *ACM Trans. Math. Softw. 15,* 1, 1–14.

Duff, I. S., Heroux, M. A., and Pozo, R. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw. 28,* 2, 239–267.

Frayssé, V., Giraud, L., Gratton, S., and Langou, J. 2005. Algorithm 842: A set of gmres routines for real and complex arithmetics on high performance computers. *ACM Trans. Math. Softw. 31,* 2, 228–238.

Gallo, C. and Manzini, G. 1998a. 2-D numerical modeling of bioremediation in heterogeneous saturated soils. *Transport in Porous Media 31*, 67–88.

Gallo, C. and Manzini, G. 1998b. A mixed finite element/finite volume approach for solving biodegradation transport in groundwater. *International Journal on Numerical Methods in Fluids 26,* 5, 533–556.

Gilbert, J. R., Moler, C., and Schreiber, R. 1992. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications 13,* 1, 333–356.

Glass, G. and Schuchert, B. 1995. *The STL Primer*. Prentice Hall PTR.

Gould, N. I. M., Scott, J. A., and Hu, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw. 33,* 2, 10.

Hestenes, M. R. and Stiefel, E. 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49,* 6.

Li, X. S. 2005. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw. 31,* 3, 302–325.

Li, X. S. and Demmel, J. W. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw. 29,* 2, 110–140.

Lippman, S. B. 2000. *Essential C++*. Addison Wesley.

Manzini, G. and Mazet, S. 2002. An object-oriented interface for the dynamic memory management of sparse discrete mathematical operators in numerical scientific applications. *Software - Practice and Experience 32,* 7, 621–644.

Manzini, G. and Putti, M. 2007. Mesh locking effects in the finite volume solution of 2-D anisotropic diffusion equations. *Journal of Computational Physics 220,* 2, 751–771.

Manzini, G. and Russo, A. 2008. A finite volume method for advection-diffusion problems in convection-dominated regimes. *Comp. Methods Appl. Mech. Engrg 197/13-16*, 1242–1261.

Meyers, S. 2005. *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs*, third ed. Addison-Wesley Professional.

Musser, D. and Saini, A. 1996. *STL tutorial & reference guide: C++ programming with the Standard Template Library*. Addison-Wesley.

Oppe, T. C. and Kincaid., D. R. 1990. Are there iterative BLAS? Tech. Rep. CNA-240, Center for Numerical Analysis, University of Texas at Austin.

Plauger, P., Stepanov, A. A., Lee, M., and Musser, D. R. 2000. *The C++ Standard Template Library*. Prentice Hall.

Saad, Y. 1990. SPARSKIT: A basic toolkit for sparse matrix computation. Tech. rep., Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign.

Saad, Y. and Schultz, M. H. 1986. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput. 7,* 3, 856–869.

Sala, M., Stanley, K. S., and Heroux, M. A. 2008. On the design of interfaces to sparse direct solvers. *ACM Trans. Math. Softw. 34,* 2, 1–22.

Stroustrup, B. 1998. *The C++ Programming Language (third edition)*. Addison-Wesley.

van der Vorst, H. A. 1992. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput. 13,* 2, 631–644.

van Heesch., D. 1997. Source code documentation generator tool. `http://www.stack.nl/~dimitri/doxygen/`.

Veldhuizen, T. L. 1995a. Expression templates. *C++ Report 7,* 5, 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

Veldhuizen, T. L. 1995b. Using C++ template metaprograms. *C++ Report 7,* 4, 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

Veldhuizen, T. L. 1998. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*. Lecture Notes in Computer Science. Springer-Verlag.

Veldhuizen, T. L. 1999. The Blitz++ Home Page. `http://oonumerics.org/blitz/`.