Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXVII, págs. 2907-2925 (artículo completo)
Alberto Cardona, Mario Storti, Carlos Zuppa. (Eds.)
San Luis, Argentina, 10-13 Noviembre 2008

A TENSOR LIBRARY FOR SCIENTIFIC COMPUTING

A.C. Limache[a] and P.S. Rojas Fredini[b]

[a]*International Center of Computational Methods in Engineering (CIMEC) INTEC-CONICET. Santa Fe, Argentina.* http://www.cimec.com.ar/alimache

[b]*Department of Informatics, FICH, National University of the Litoral (UNL). Santa Fe, Argentina.*

Keywords: LTensor, scientific computing, tensor library, C++ library, indicial notation.

Abstract. The majority of physical phenomena and their computational simulations are described mathematically in terms of tensors and their different algebraic operations. Possibly the most used tensors are the ones of rank 1 and 2, which correspond to the algebraic concepts of vectors and matrices, respectively. Nevertheless, higher rank tensors (specially 3 and 4) appear at all times in different branches of physics and in numerical methods. One of the major drawbacks of high performance computing is that the code necessary to perform such tensor operations looks different and it is several lines longer than the corresponding one-line mathematical representation. Here we present a C++ tensor library, called LTensor, that we have developed using modern concepts of object oriented design and expression templates. As it will be shown, the LTensor library is able to mimic the classical indicial notation and follows Einstein convention about indices. Furthermore, it has other additional features than distinguish it from other libraries based on similar concepts: dynamic dimension size, arbitrary contraction order, customizable storage, inherited class structure, arbitrary looping positions on indicial notations, etc.

1 INTRODUCTION

The C++ programming language has interesting features that make it an excellent choice for scientific and engineering applications. One of these known features is operator overloading. Operator overloading makes it possible to write algebraic expressions containing vectors or matrices in a similar way one would write them in a piece of paper. For example given a matrix (Tensor2) A and three vectors (Tensor1) b, c, d one can perform the following algebraic operation:

```
c = A*b+d;
```
(1)

once one declares them as:

```
Tensor2 A;
Tensor1 b,c,d;
```

But this level of abstraction comes at a high cost, since tensors are usually implemented using temporary objects (Veldhuizen, 1995). The code generated by expression (1) is equivalent to:

```
Tensor1 t1 = A*b;
Tensor1 t2 = t1+d;
c = t2;
```

Each of the above expressions uses a loop to evaluate the operation, so the compiler generates three sequential loops to accomplish the original expression. This represents a big overhead compared to the classic C-programming style where the desired operation can be accomplished with the following code:

```
double **A;
double *b,*c,*d;

// here goes allocation and initialization

for (int i=0;i<dim;i++)
{
     for(int j=0;j<dim;j++)
     {
          c[i]+= A[i][j] * b[j];
     }
     c[i]+= d[i];
}
```

Using the C-programming style only two loops are needed, which results in a shorter evaluation time. We also have another benefit: the temporaries are not necessary. However, in the C-style approach the syntax is far more complex and less intuitive than the operation overloading alternative defined in eq. (1). Also if we need to do a minor change in the operation, for example, if instead of an inner contraction we want to compute an outer contraction, we need to modify the routine completely. This is specially annoying when working with arbitrary tensor contractions. The situation gets worst when dealing with higher order tensors, where nested loops make the code error prone and harder to follow.

Because of these reasons some people usually prefer interpreted languages like Matlab, Octave, Python, etc., which offer a clear syntax and are easier to use, but their performance is by far not comparable to native C or FORTRAN performance.

The above described panorama has changed drastically in favor of C++ with the work by Veldhuizen and his colleagues (Veldhuizen, 1995a; Veldhuizen and Jernigan, 1997) who developed a technique known as **template expressions**. They described how to use an unintended template feature to evaluate expressions in a single pass by building trees of expression objects. This opened a new world of possibilities, and motivated the development of new libraries which exploded this technique (Veldhuizen, 1998; Landry, 2002; Ahlander ; Jeremic , B. , Sture, S , 1998; Ilyin, V. , Kryukov, A , 1996; Blinn, 2000). Along with this technique emerged another one called **template metaprogramming (**Veldhuizen, 1995b; Veldhuizen, 1999) , which allowed the generation of code at compile time with some restrictions.

FTensor (Landry, 2002) and Blitz (Veldhuizen, 1998) have had the biggest influence at the time of developing the present library. FTensor is highly focused on performance, at the cost of some sacrifices regarding flexibility, like the impossibility to change the size of a particular dimension of an array at runtime. It incorporates indicial notation with Einstein convention to C++ syntax in a natural way, making possible to write expressions like the one defined in Eq. (1) as:

```
Tensor2 A;
Tensor1 c,b;
c(i)=A(i,j)*b(j)+d(i);                                    (2)
```

This not only offers an improved legibility but also is efficient, evaluating the expression on a single loop, with no temporaries.

On the other hand Blitz library offers much more flexibility but the syntax for Einstein notation is not as clear as the previous one in the case of contractions.

Taking these ideas, a library was developed named LTensor, featuring multi-indexed arrays up to rank 4, Einstein notation with a natural syntax, dynamic dimension size, and an inheritance structure offering a good balance between flexibility, performance and legibility.

2 DESIGN OVERVIEW

The library was designed with flexibility in mind. An inherited class structure was chosen like in Fig. 1. The main class is Marray which inherits from Base the main functionality. The other important class is TExpr; this is the one that allows the implementation of the index notation as it will be shown later.

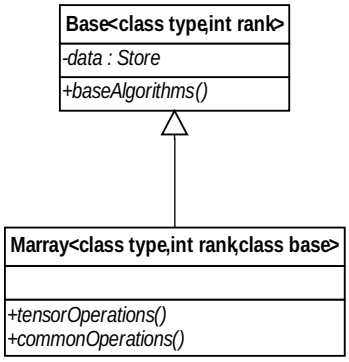

Figure 1: Inheritance design

2.1 Base class

Base is the base class templatized with rank and type. It is responsible of offering the basic functionality for an array of **rank=1, 2**, etc. and of **type=int, double, complex**, etc. It manages the memory where the data will be stored, it is in charge of allocation an de-allocation of memory and basic operations on the data. The member **data** in Fig.1 is only a guideline, depending on the base class implementation it can have a total different shape, although there is an interface that must be met in order to provide all the methods needed by the Marray class.

This way, is possible to have different behaviors and optimizations regarding the natural structure of the data for the problem at hand.

2.2 Marray class

This is probably the most important class from the perspective of the programmer. This class is specialized on the template parameter **rank,** allowing different implementations depending on the tensor rank. This could look like a design fault, but saves many runtime instructions that otherwise would be needed on each method to determine the rank of the tensor at hand.

The first template parameter indicates the type of the data that will be stored in the array. The third parameter defines which will be the class to inherit from. Although there are different specializations of this class, it should be noted that the **Base** is the same on all of them. So **Base** must provide functionality for all ranks.

The class **Marray** implements all the required methods to support indexed expressions, generic functionality and numeric algorithms referred as **commonOperations()** in Fig. 1. No data is actually stored on this class; it acts as a wrapper adding the functions described earlier.

Because of this design pattern, it is possible to have different types of tensors like sparse, symmetric, etc. But it is not limited to different tensor types; it is possible to have arbitrary memory ordering, for example, FORTRAN-style arrays.

The equals and index operators of **Marray** call the **Base** equals and index operator respectively, as shown in Fig. 2 for the case of the index operator. **Marray** class doesn't know where the data comes from, nor if it is preprocessed or altered in some way. This is what gives freedom of implementation to the **Base** class. A simple change of the **Base** class results in a tensor with the characteristics defined by the base and the functionality – including the index expressions- of the **Marray** class.

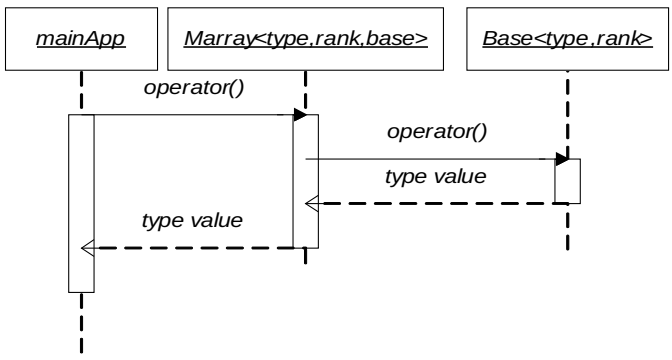

Figure 2: Sequence of operator relying

2.3 Tensor Expressions

One of the main features of the present library is the possibility to write complex operations using the Einstein summation convention in a natural way. For this task two new classes were needed to act as index entities: **IndexF** and **IndexG,** as shown in Fig. 3.

IndexG is the simplest class. It is a container of a char character which plays the role of the index identifier. When these indices are used the compiler knows that loops have to performed over all the dimension of the indicated tensor component. The IndexF class has an additional member named **indexes** which is used to specify the positions the index will loop over.

The main class involved in the tensor expressions is TExpr, which is a container for objects of type determined by its own template parameter. This way TExpr holds pointer to objects, whose only restriction is to have the **operator()** defined.

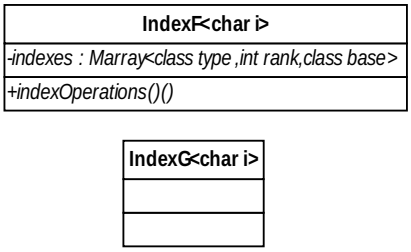

Figure 3: Index classes

Generically speaking there are two classes of objects a **TExpr** can hold:

1. Unary Objects
2. Binary Objects

The first one is an object that holds a pointer to a **Marray** or to another **TExpr** object, but only one. The Unary object can apply modifications to the object it holds, like making it negative or scaling by a constant for example.

The second is an object that holds two pointers instead of one, to another **TExpr**, or Marray. This object performs binary operations between them.

Both of them have the **operator ()** defined, allowing the creation of trees of **TExpr** as shown in Fig. 4.

There are objects **TExpr** for the different ranks supported by the library, named **TExprN** with N=1..4. Those objects have operations defined between them by operator overloading, and between them and the **Marray** class. Those operators are the ones in charge of doing the loop and assignation along the **TExpr** dimensions, because in the end the **TExpr** represents a complex expression that can be indexed. The calling sequence is shown in Fig. 5 for a simple case. It can be seen how the indexing operator is spread up the tree and every object applies the operation it represents. For example in Fig. 5, the BinaryObject could be an add operator, adding the two **Expr2** and returning the result. This way, in a single for loop is possible to evaluate the whole expression.

The Binary and Unary Objects fall in different categories depending on the task they perform:

1. **Encapsulation Object**: this object works as an encapsulation for Marray objects. It makes possible to encapsulate Marray objects of different dimensions than the TExpr container. It is useful in the cases where constant indexes appear on expressions, lowering the number of free indexes. This class also provides the mechanisms for the

two kinds of indexes described earlier to work, keeping a reference to the indexes in the case of an IndexF, and returning the correct value of the contained expression according to them.

2. **Binary Operators:** the objects falling under this category perform binary operations on the two contained TExprs, and return another TExpr with the result of the op. One of the most relevant is the Contraction object that given two TExpr with their associated indexes returns the results of the contraction.
3. **Unary Operators:** the most common operations involving objects from this category include Marray sign inversion and operations involving scalars.

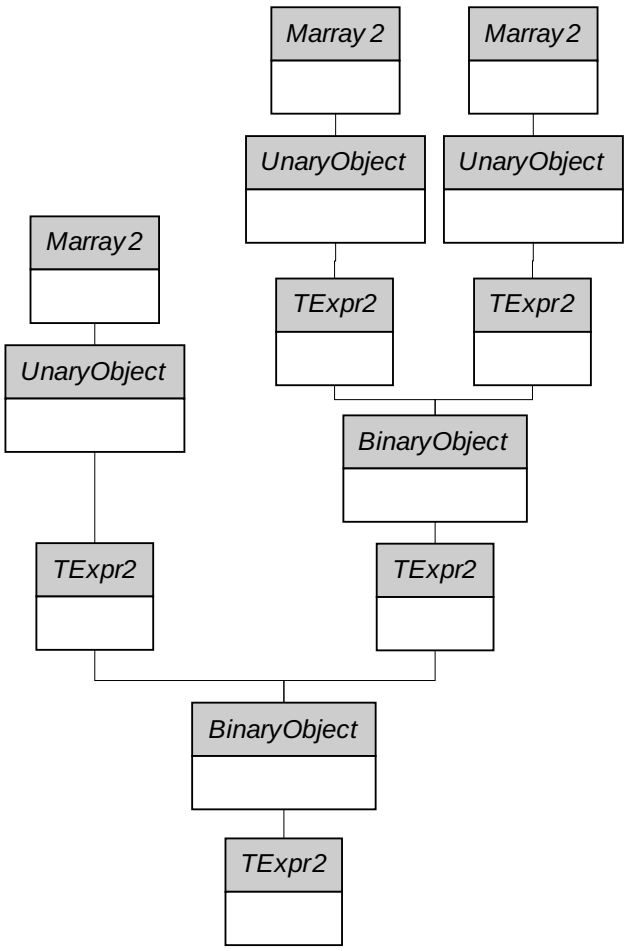

Figure 4: Simplified tree expression representation

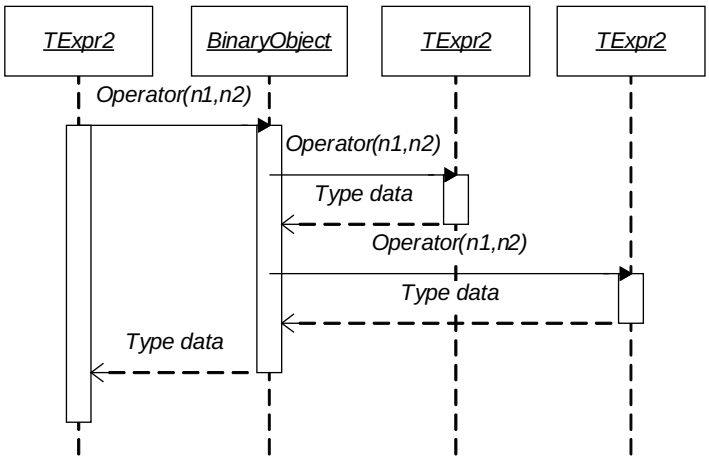

Figure 5: Expression calling sequence

2.4 Equal operator overloading

This is the key for the described hierarchy to work. The responsible of performing the evaluation of the **Texpr** is the operator equal (**operator=**) of the **Marray** class and the operators equal of the **TExpr** classes. Each one of them performs a loop along the indexes of the **Texprs** evaluating the whole tree on each iteration, and assigning the returned value to the left side of the assignation, that of course can be another **TExpr** tree.

3 IMPLEMENTATION

Currently two base classes are implemented to act as the Base object, **TinyArrayBase** and **ArrayBase**, each one of them designed for different usage scenarios.

The Marray classes provide implementations of common numeric algorithms, but of course, lack of possible optimizations according to the type of the Base. In the cases where those algorithms could perform better due to a characteristic of the Base the implementations should be provided by it.

As it is well known, there is always a balance between flexibility and performance. In developing the current library some performance penalties were accepted in exchange for a clearer code. This allows the occasional reader to understand faster what the algorithm is performing, instead of going though thousands of cryptic instructions. The programmer is relieved from the burden of implementing complex index contractions and other operations that can be accomplished in a human friendly way by this library.

3.1 TinyArrayBase class

This is a simple class with minimum functionality in favor of speed. It is intended to be used when the size of the Marrays is small, and should be used by default if no other base is provided. It's highly optimized for speed and has a very little overhead on the functions called by the Marray class. It always works with C-style arrays, not being possible to change this.

The internal storage of the data is a normal C-array with a mapping corresponding to the rank. A copy is always made when the **operator=** is called, with no possibilities of working with references. In this way each TinyArrayBase is the owner of its memory.

This class provides the different operator() overloads in order to work with any of the possible Marray ranks.

3.2 ArrayBase class

This class was designed with functionality in mind. It offers a lot of features, and possibilities in exchange for a bigger penalty in performance than the case of the **TinyArrayBase**. This class is intended to be used when the dimensions of the Marray are big. Some of the most relevant features are:

1. **Arbitrary dimensions ordering:** the ordering of the dimensions of this class can be provided by the user. It allows working with C-style arrays (default), FORTRAN-style arrays, or any ordering provided by the user. This ordering is used to store the data in memory. This permits implementing FORTRAN algorithms, directly without worries about the penalty for the dimensions ordering. Or using a custom storage order according to the problem at hand. The memory is effectively ordered by this parameter, to benefit from less cache misses, less memory reads, etc.
2. **View/Storage model:** None of these classes is the owner of its storage. Each instance has a pointer to a **StorageClass** that can be customized to fit different needs. GeneralStore is the common storage provided for this class. Each time an **ArrayBase** is instanced it creates a new GeneralStore and keeps a pointer to it. Each **ArrayBase** acts like a view of the storage, it does not own it. The store manages the memory, allocation, deallocation, ordering, etc. And the **ArrayBase** behaves like a filter: it allows seeing the store completely, with a determined stride, etc.
3. **Copy behaviour:** As said before, this class does not own its storage, so each time the equal operator is used, the view properties are copied, not the data itself. The storage remains unique, and is shared amongst all the **ArrayBase** -views- pointed to him. So modifying one, modifies all of them, because they share the data. This applies when working with the same data Type. When assigning **ArrayBase** of different Type, a new storage is created, and the data casted to this new Type.
4. **Storage Lifetime:** The storage works with a reference counting scheme in a very similar way to smart pointers. When the reference count falls to zero it de-allocates the memory.

As shown above, this class is very useful, when working with big **Marrays**, because no memory copies are made, and with the views is possible to work with parts of the big **Marray**, as if they were smaller **Marrays**.

3.3 Performance penalties

As stated before some performance penalties were admitted in pos of a better code legibility and programming flexibility for the intended user. The presence of the inherited hierarchy is the first thing to take into account; to minimize the impact of it, extensive inlining was used in the most performance critic methods. An example of this is the **operator()** which is the principal actor on the performance play. A lot of methods use the const signature also to counter this problem.

Another major performance hit resides in the dynamic dimensions. This makes impossible to use **template metaprograms** on all the loops along the dimensions for some kind of operations.

However, we must emphasize that the performance penalties can be considered a cheap trade-off if one consider the easy of use of the resulting programming syntax: the Ltensor user can write tensor operations in a simple, intuitive, concise form, as natural as writing tensor formulas by hand.

3.4 Performance optimizations

On the other hand, some performance optimizations were made. As shown before, an extensive use of expression templates is made. This traduces in single loop evaluations for complex expressions.

The use of restricted pointers also provided a big performance boost. Special care was taken when using that kind of pointers.

Although the dimensions of the **Marrays** are fixed, the rank is not. So some **meta programming** was used along the rank loops, to unroll them.

Perhaps the most used technique in this library is specialization. Almost every templatized class has specializatons. This is due to two reasons. First, a lot of processing time is saved if some decisions are made at compile time. For example, providing one specialization for each rank of a **Marray**, saves the need of checking at running time the rank of the tensor, and allows the use of metaprogramming on each specialization. Second, and probably more important, is the fact that in order to have all their features the expressions objects rely heavily on specialization. For example, if we have a contraction of two indexes, the compiler searchs along all the specializations of the overloaded method **operator*** and instantiates only the one that matches, so at runtime there is no need to check which indexes contracts or which indices are free, avoiding a run-time logic of index contraction. Of course this means implementing all the possible specializations of encapsulations, contractions, index permutations, etc. This required an extense programming work but this work, once done, allows handling all the universe of possible contractions and algebraic operations.

An example is given. If we consider the standard matrix vector product (in indicial notation):

$$c_i = A_{ij} * b_j$$

with the LTensor library we can compute it by writing it in an identical format:

```
IndexG <'i'> i;
IndexG <'j'> j;

c(i)=A(i,j)*b(j);
```
(3)

The expression on the right side of eq. (3) will generate a contraction **BinaryObject** containing pointers to A and b. This is not enough to define the contractions, so another parameter should define the contraction itself. This is achieved with the IndexG objects. Those objects provide the template chars that permit the specialization of the contraction object. In this case the compiler will instantiate the method **operator*()** that contracts along index j.

3.5 Algorithms

The library includes some of the most common algorithms used in numeric calculus. Most of them belong to rank 1 and 2 of Marray Objects. The most relevant are:

1. Norm zero
2. Norm infinite
3. Norm N
4. Quicksort
5. Inverse
6. Gaussian elimination
7. LU factorization

8. Linear System Solver
9. Cholesky factorization
10. Determinant

3.6 STL compatibility

The library is fully compatible with the STL iterators. This allows the use of STL algorithms on the Marray classes. The only thing to be taken into account is the order the iterator uses to visit all the positions. The ordering is fixed, and correspond to a C-style array. Meaning the Marrays are iterated in a row-wise manner. Providing the iterators is responsibility of the base classes, giving them the possibility to perform optimizations depending on the structure each one has.

3.7 Tensor Expressions

The tensor expressions support only lower indexes, and are fully compliant with the Einsten summation convention. They support also the presence of scalars in the expressions.

The expressions support arbitrary contractions of any form up to rank 4. These contractions can be done along all the elements of a given dimension, using IndexG indices, or can be done along specific indexes of a container, using IndexF indices. In both cases the validation of the expressions is done in an implicit way by specialization. This makes it easy to find wrong formulas, and typing errors. The validation does not only check the dummy indexes but also validates the free indexes with the right side term, resulting in a full expression validation at compile time.

To accomplish the features described above it was necessary to implement one by one all the possible contractions and permutations for the expressions up to rank 4. Although it was a tedious task, it was a one-time job, and permits the strict validation described above with no overheads at runtime.

3.8 Serialization

The library provides mechanisms to serialize and de-serialize from disk. This allows loading Marrays from space separated files, making it easier to share data amongst previous applications. This also allows exporting data for post processing or visualization.

4 SYNTAX AND COMMON OPERATIONS

In this section a brief overview of the library syntax and features will be given.

4.1Arrays operations

```
Marray <type,rank,base=default> a;
```

This is the default syntax when creating an Marray Object. The first and second parameters are obligatory indicating the type of the data and the rank of the Marray. The third parameter is optional, taking the class TinyArrayBase as default. Then, with the following type-definition:

```
typedef Marray<double,1> DTensor1;
typedef Marray<double,2> Dtensor2;
typedef Marray<double,3> DTensor3;
```

we can define sets of tensor objects of rank 1, rank 2 (i.e. vectors and matrices) or rank 3.

For example,

```
DTensor1 a(6), b(6), c(4), d(4);
DTensor2 A(4,6), B(4,6);                                    (4)
DTensor3 E(3,3,3);
```

Below are shown additional examples of some common supported operations:

```
a=b;
a(0)=10.0;
A(2,3)=1.0;
A+=B;
```

Note that the Marray constructor receives the tensor dimensions as the first parameters, and these parameters can be (optionally) followed by a default initialization value.

4.2 Iterators

```
DTensor1::iterator it;
it=a.begin();
while(it!=a.end()){
   //some operations
   it++
}
```

The LTensor iterators are STL compatible, and provide an easy way to loop through all the elements of an Marray in a linear way. The implementations of each iterators is provided by the base class, to exploit the natural characteristic of the data. For example a sparse base, won't be iterated in the same way as a dense ones would.

4.3 Tensor Expression

As said before two classes of indexes can be used as indexes of tensor expressions. The first one, named IndexG is a simple class with only one template parameter: a *char*, which uniquely identifies an index in a tensor expression. For example three different IndexG objects can be declared as:

```
IndexG <'i'> iG;
IndexG <'j'> jG;                                            (5)
IndexG <'k'> kG;
```

It must be noticed that despite the name given to the index object, the char determines which index it represents. The IndexG class uses the templatized char similarly as it is used in the Index class of the FTensor library however one major difference is that the IndexG class does not require any additional "dimensional" parameter as the Ftensor's class does. Note that in eq. (5) we have named the index objects ending with a "G" so as to clearly identify the type of index. IndexG objects mean that the whole dimension they index will be used in the expression. For example, with the LTensor library and the declarations given in Eqs. (4) and (5), we can compute simultaneously a matrix vector product and a vector addition:

$$c_i = A_{ij} * b_j + d_i \tag{6}$$

in the same natural way:

```
c(iG)=A(iG,jG)*b(jG)+d(iG);
```
(7)

The compiler and the library will do the job for us and perform automatically the inner product, looping the jG index along the tensor dimension of b and the second tensor dimension of A, and looping the iG index along the dimension of vector c.

Note, that if we use a wrong index to perform the product, as in:

```
c(iG)=A(iG,jG)*b(kG)+d(iG);
```

the compiler will throw us an error letting us know of our mistake.

We have created another index class, named IndexF, to have an additional flexibility which is to have the capacity to loop over a specified set of indices along a tensor dimension. In other words, on the contrary to IndexG objects, with IndexF objects we can loop not only along the entire index dimension but along any subset of index values, which can be given or changed at runtime. IndexF objects can be declared as follows:

```
IndexF <'i'> iF(init,end,stride);
IndexF <'j'> jF(Marray);
IndexF <'k'> kF(size);
```

The declaration of IndexF objects is a bit more complex than the one of IndexGs, because besides the char representing the index, we can set an integer array representing the container positions this index will use when participating in an expression.

As an example of their use, note that if we define two indexF objects:

```
IndexF <'i'> iF(4);
IndexF <'j'> jF(0,6,2);
```

and the expression:

```
c(iF)=A(iF,jF)*b(jF);
```

we will be able to perform a reduced matrix vector product, where the inner product, contracting along the jF index loops only along the even positions of the second dimensions of A, and the even positions of b. On the other hand, since the index iF has been set a dimension of 4, it will still loop along the whole dimension of c.

As seen above the IndexF iF, acts as an IndexG looping through the whole dimension, this produces a little overhead because iF needs to hold an array indicating those positions. Ideally we would like to not to have to define this redundant index. To fix this, we improved further the features of the library in order to allow the presence of both type of indexes (IndexF and IndexG) in the same Expression. So now it is possible to write:

```
c(iG)=A(iG,jF)*b(jF);
```

and no overhead is present, because as said before IndexG acts only as a container for the template parameter.

The use of IndexF indexes introduces some performance penalties, but allows performing some operations that couldn't be achieved without having to write long portions of code every time the need arises. A practical example of the utility of IndexFs is when performing assembly operations in computational mechanics codes (Limache, A. and Idelsohn, S. 2007;

Limache A. 2008). Using these indexes, portions of elemental matrices can be easily inserted into global matrices. Or one can work directly over the global matrix, but with the indexes looping on the elemental matrix contained.

5 PERFORMANCE TESTS

Tests were made comparing the performance of the present library versus the hand coded version of the same algorithm. The base used for the tests is TinyArrayBase because is the one oriented to performance. All the times are measured in seconds.

The following contraction is used for the tests

```
a(iG)=b(iG,jG)*c(jG)+d(iG);                                    (7a)
```

This test was run changing the size of the Arrays involved. Table I shows the results of the current test. The relative performance (hand coded time/LTensor time) is above 1 for all the tests. This means the LTensor implementation performs better than the hand coded version shown in Appendix 9.1. This improvement is mainly due to the use of restricted pointers, that speed up array indexing. The oscillation in the graph correspond to memory management issues. For very small Marrays the improvement is more noticeable because the overhead of the indexing operator doesn't have a big impact, this changes as the size increases. But the performance superiority remains along all the tests. The behavior can be seen in Fig. 6 where it is shown that for big Marrays the performance tends to stabilize. On average the relative performance of the LTensor computation is 2,50 times better than the naive C-coded version.

Size	Ltensor	Hand Coded	Performance
3	1,67E-06	2,09E-06	1,25
10	1,95E-06	4,19E-06	2,15
50	7,61E-06	3,40E-05	4,47
100	2,49E-05	9,01E-05	3,62
500	1,30E-03	2,00E-03	1,54
1000	4,40E-03	8,40E-03	1,91
3000	3,30E-02	7,00E-02	2,12
5000	6,00E-02	1,80E-01	3,00

Table 1: Relative Performance Table vs Hand Coded

The tensor expression (7a) was also tested against a naive tensor implementation based on standard C++ operator overloading, this naive implementation is included in the Appendix 9.2. The results are shown in Fig. 7. From the figure it can be seen that the LTensor approach performs better than the C++ overloading approach (despite the overhead of the indexing operators, and the inherited hierarchy). One of the biggest performance penalties of the standard C++ overloading approach is caused by the copy of temporals. Each binary operators generate temporals, that are copied and returned. This results in big memory operations due to data copy, cache misses and pagination algorithms. The LTensor approach does not generate temporals, so the memory management problem does not appear in the operation. The performance results are summarized in Table 2.

Again a relative performance factor (overloading time/LTensor time) was calculated and is shown in Fig. 8. There, the benefits of using the **LTensor** library can be clearly seen, specially for big size **Marrays**. It must be also said that the classical overloading approach does not allow arbitrary contractions, so we can only define very specific contractions and

can not go farther than implementing the inner matrix-vector product. Another tests were run to compare the operator= for both kinds of Marray Bases. As shown in Fig. 9 the operator equals (operator=) for TinyArrayBase takes more time than the ArrayBase one. The data is shown in Table III. As explained earlier, this is due to the fact that the TinyArrayBase class works by copying the data between objects and the ArrayBase class works as a view of the storage.

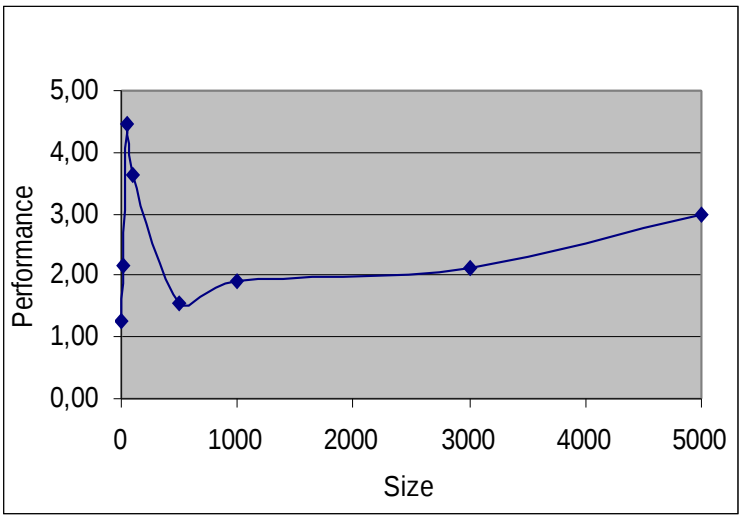

Figure 6: Relative performance vs hand coded

Operator overloading	Relative performance
4,67E-06	2,80
9,70E-06	4,97
5,15E-05	6,77
1,38E-04	5,55
2,08E-03	1,60
9,29E-03	2,11
8,20E-01	24,85
4,99E+00	83,17

Table 2: Classic operation overloading apporach

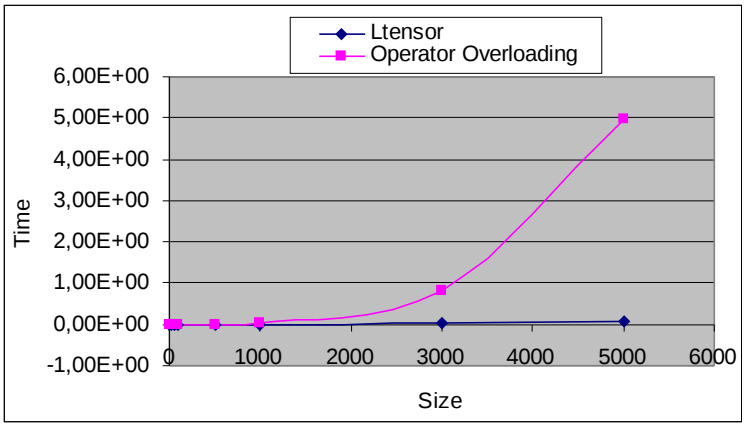

Figure 7: LTensor– Operator overloading comparison

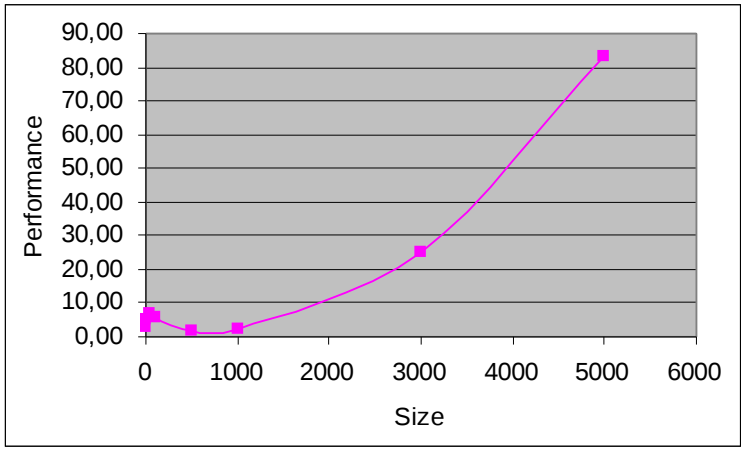

Figure 8: Relative performance of classic C++ overloading

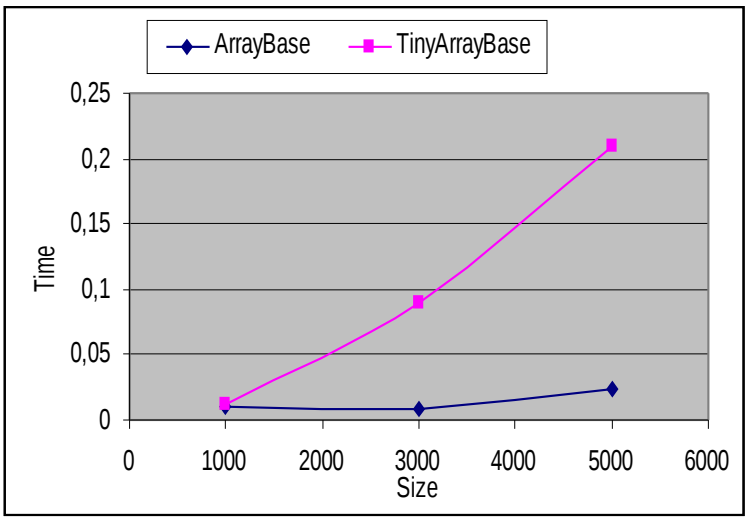

Figure 9: Operator equals (operator=) comparison

Size	ArrayBase (s)	TinyArrayBase (s)
1000	0,009457	0,01218
3000	0,0081	0,089
5000	0,024	0,21

Table 3: Operator equals (operator=) test

6 EXAMPLES

6.1 MulPhys

The LTensor library has been successfully employed in an advanced computational mechanics code named MulPhys (Limache A. 2008). As we know the Finite Element Method (FEM) requires the computation of tetrahedron volumes as part of the numerical requirements, mathematically the volume of a tetrahedron with sides a, b and c, is given by:

$$vol = \frac{(a \times b) \cdot c}{6.0} \tag{8}$$

where x denotes de cross vector product. Using indicial notation and the Levi-Civita tensor E, we can write the above expression as:

$$vol = (E_{ijk} a_j b_k c_i) / 6.0 \tag{9}$$

well, using the LTensor we can compute the volume exactly in the same concise way:

```
vol = (E(iG,jG,kG)*a(jG)*b(kG)*c(iG))/6.0;
```

it works, it is simple and extremely efficient.

6.2 Linear Elasticity

Linear elasticity is used extensively in structural analysis and engineering design. Its constitutive equation is

$$\sigma_{ij} = C_{ijkl} \varepsilon_{kl} \tag{10}$$

where σ_{ij} is the Cauchy stress tensor, C_{ijkl} is the elasticity tensor and ε_{kl} the strain tensor. Using the LTensor library we can compute Eq. (10) as:

```
Sigma(iG,jG)=C(iG,jG,kG,lG)*E(kG,lG);
```

The above implementation is simple, clear and doesn't require temporal variables or initializations like the C-style coding of the same equation.

6.3 Arbitrary Contractions

The library is suitable for a lot of computational scenarios where arbitrary contractions are needed in order to perform specific operations. For example :

```
A(iG,jG,kG,lG)=B(iG,mG)*C(mG,jG)*D(kG,lG) + E(iG,jG,kG,lG)/2.0;
```

permits the manipulation of high order tensors with ease, even if those arise as a result of lower order tensor operations.

6.4 Matrix Assembly Operations

Given iF and jF, the indexes of a global tensor A where the assembly must be done and local indexes iG and jG of the local tensor a, the assembly operation can be achieved by simply doing:

```
A(iF,jF)=a(iG,jG);
```

This operation uses IndexFs to iterate over arbitrary positions in the tensor A and IndexGs to iterate along the whole dimension of a. This shows the potential of mixing both types of indexes in a practical situation. Objects iF and iG have to have the same template parameter to comply with the index convention. The same requirement must hold for the "j" indices: jG and jF.

7 CONCLUSIONS

A new tensor library was presented. The library offers tensor indicial notation support with Einstein summation convention. It uses a concise, simple and natural syntax, letting the programmer write complex tensor formulas in the same way one would write them in a piece of paper. It performs compile-time verification, and allows arbitrary contractions, not restricting the user to the common cases. It also allows operations of matrix compositions, very used in scientific applications.

The library uses the template expression technique to provide single loop expression evaluation. It provides an inherited hierarchy allowing the user to customize the Marray class for the specific needs of the problem at hand, allowing to perform optimizations regarding the special characteristic of the problem. This is done by passing the operator equals and the index operator to the base class. Also iterators are provided to allow full customization.

A balance between optimization and flexibility was achieved. The performance when using the TinyArray class as the base class surpass a C-coded version. On the other hand, it has also a much better performance than the classic C++ overloading approach.

The library is a nice solution to those who are tired of the long pieces of code of old C-coding style or the slow down of C++ standard approach, specially in collaborative software development enviroments where better legibility is required. And of course specially when performance is an important factor.

8 FUTURE WORK

The next steps regarding the present library would be to provide a bigger spectrum of base classes like sparse storage, or symmetric tensors.

Parallel processing is a must in scientific applications, in order to support this, evaluations are being made to write an interface with the library PETSc.

9 APPENDIX

9.1 Naive C-style implementation

```
double **db=new double*[size];
double *da=new double[size];
double *dc=new double[size];
double *dd=new double[size];
 for(int i=0;i<size;i++){
    db[i]= new double[size];
 }
 //here goes array initalization
 //
 for(int i=0;i<size;i++){
     for(int j=0;j<size;j++){
          da[i]+=db[i][j]*dc[j]+dd[i];
     }
     da[i] += dd[i];
}
```

9.2 Naive C++ overloading implementation

```
class Vector{
private:
      double *data;
      int size;
```

```
public:
      Vector(int size){
            this->size=size;
            data=new double[size];
      }
      double &operator()(int n1)
      {
            return data[n1];
      }
      double operator()(int n1)const
      {
            return data[n1];
      }
      void operator=(const Vector &a){
            for(int i=0;i<size;i++)
                  data[i]=a(i);

      }
      Vector(const Vector &b){
            this->size=b.size;
            data=new double[size];
            for(int i=0;i<size;i++)
                  data[i]=b(i);
      }
      Vector operator+(const Vector &a){
            Vector ret(size);
            for(int i=0;i<size;i++)
                  ret(i)=a(i)+operator()(i);
            return ret;
      }
};


class Array{
private:
      double** data;
      int size;
public:
      Array(int size)
      {
            this->size=size;
            data=new double*[size];
            for(int i=0;i<size;i++)
                  data[i]=new double[size];
      }
      Array(Array &b){
            this->size=b.size;
            data=new double*[size];
            for(int i=0;i<size;i++)
                  data[i]=new double[size];
            for(int i=0;i<size;i++)
                  for(int j=0;j<size;j++)
                        data[i][j]=b(i,j);
      }
      double &operator()(int n1,int n2)
      {
            return data[n1][n2];
      }
```

```
        double operator()(int n1,int n2)const
        {
              return data[n1][n2];
        }
        void operator=(const Array &a){
              for(int i=0;i<size;i++)
                    for(int j=0;j<size;j++)
                          data[i][j]=a(i,j);

        }
        Vector operator*(const Vector &a){
              Vector ret(size);
              for(int i=0;i<size;i++){
                    ret(i)=0.0;
                    for(int j=0;j<size;j++){
                          ret(i)+ operator()(i,j)*a(j);
                    }
              }
              return ret;
        }
  };
```

REFERENCES

Ahlander K., Einsum. *http://www.ii.uib.no/~krister/EinSum*.

Blinn J.F., "Optimizing C++ Vector Expressions," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 97-103, Jul/Aug, 2000.

Ilyin V., Kryukov A., ATENSOR-REDUCE program for tensor simplification. *Institute of Nuclear Physics, Moscow State University* , 1996.

Jeremic B., Sture S., Tensor Objects in Finite Element Programming. *International Journal for Numerical Methods in Engineering*, Vol. 41:113-126 , 1998.

Landry W., Implementing a High Performance Tensor Library, *http://www.oonumerics.org/FTensor/FTensor.pdf , 2002.*

Limache A.C. and Idelsohn S.R., "On the Development of Finite Volume Methods for Computational Solid Mechanics", *Mecanica Computacional, AMCA; vol. XXVI: 827843; October 2007. ISSN 16666070.*

Limache A.C., MulPhys, Simulation of Physical Phenomena by Computers; *CIMEC-CONICET, Argentina*, http://www.cimec.org.ar/alimache. *2008*.

Veldhuizen T., Arrays in Blitz++, *Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments* (ISCOPE'98), 1998.

Veldhuizen T., C++ Templates as Partial Evaluation. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), 1999.*

Veldhuizen T., Expression Templates. *C++ Report*, Vol. 7 No. 5:26-31, 1995

Veldhuizen T., Jernigan M. E.; Will C++ be faster than FORTRAN?. *Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments* (ISCOPE'97), 1997.

Veldhuizen T. , Using C++ Template Metaprograms. *C++ Report*, Vol. 7 No. 4, 1995.