Published in :

International Journal for Numerical Methods in Engineering

Volume 41, 113-126 (1998)

Tensor Objects in Finite Element Programming

Boris Jeremić [3] **Stein Sture** [4]

University of Colorado at Boulder, Colorado, U.S.A.

[3] Graduate research assistant,
Department of Civil, Environmental, and Architectural Engineering,
Campus Box 428,
University of Colorado at Boulder,
Colorado 80309-0428, U.S.A.
phone (303) 492-7112, fax (303) 492-7317
Email: Boris.Jeremic@Civil.Colorado.edu

Current adrress:
Assistant Professor,
Department of Civil and Environmental Engineering,
University of California,
Davis, CA 95616,
tel: (530) 754 9248, fax: (530) 752 7872
Email: Jeremic@ucdavis.edu

[4] Professor and Chairman,
Department of Civil, Environmental, and Architectural Engineering,
University of Colorado, Boulder
Campus Box 428,
University of Colorado at Boulder,
Colorado 80309-0428, U.S.A.
Email: Stein.Sture@Civil.Colorado.edu

Abstract

This paper describes a novel programming tool, **nDarray**, which is designed using an Object Oriented Paradigm (OOP) and implemented in the C++ programming language. Finite element equations, represented in terms of multidimensional tensors are easily manipulated and programmed. The usual matrix form of the finite element equations are traditionally coded in FORTRAN, which makes it difficult to build and maintain complex program systems. Multidimensional data systems and their implementation details are seldom transparent and thus not easily dealt with and usually avoided. On the other hand, OOP together with efficient programming in C++ allows building new concrete data types, namely tensors of any order, thus hiding the lower level implementation details. These concrete data types prove to be quite useful in implementing complicated tensorial formulae associated with the numerical solution of various elastic and elastoplastic problems in solid mechanics. They permit implementing complex nonlinear continuum mechanics theories in an orderly manner. Ease of use and the immediacy of the **nDarray** programming tool in constitutive driver programming and in building finite element classes will be shown.

Key Words: Object Oriented Programming, Tensor Analysis, Constitutive Driver Programming, Finite Element Programming.

1 Introduction

In implementing complex programming systems for finite element computations, the analyst is usually faced with the challenge of transforming complicated tensorial formulae to a matrix form. Considerable amount of time in solving problems by the finite element method is often devoted to the actual implementation process. If one decides to use FORTRAN, a number of finite element and numerical libraries are readily available. Although quick results can be produced in solving simpler problems, when implementing complex small deformation elastoplastic or large deformation elastic and elastoplastic algorithms, C++ provides clear benefits.

Some of the improvements C++ provides over C and FORTRAN are classes for encapsulating abstractions, the possibility of building user–defined concrete data types and operator overloading for expressing complex formulae in a natural way. In the following we shall show that the **nDarray** tool will allow analysts to be a step closer to the problem space and a step further away from the underlying machine.

As most analysts know, the intention[15] behind C++ was not to replace C. Instead, C was extended with far more freedom given to the program designer and implementor. In C and FORTRAN, large applications become collections of programs and functions, order and the structure are left to the programmer. The C++ programming language embodies the OOP, which can be used to simplify and organize complex programs. One can build a hierarchy of derived classes and nest classes inside other classes. A concern in C and FORTRAN programming languages is handling data type conflicts and data which are being operated on or passed. The C++ programming language extends the definition of type to include abstract data types. With abstract data types, data can be encapsulated with the methods that operate on it. The C++ programming language offers structure and mechanisms to handle larger, more complex programming systems. Object Oriented technology, with function and operator overloading, inheritance and other features, provides means of attacking a problem in a natural way. Once basic classes are implemented, one can concentrate on the physics of a problem. By building further abstract data types one can describe the physics of a problem rather that spend time on the lower level programming issues. One should keep in mind the adage, credited to the original designer and implementor of C++ programming language, Bjarne Stroustrup: "C makes it easy to shoot yourself in the foot, C++ makes it harder, but when you do, it blows away your whole leg".

Rather than attempting here to give a summary of Object Oriented technology we will suggest useful references for readers who wish to explore the subject in greater depth.[2] The current language definition is given in the *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*.[1] Detailed description of language evolution and main design decisions are given by Stroustrup.[15]

Useful sets of techniques, explanations and directions for designing and implementing robust C++ code are given in books[3][5] and journal articles[9].[16]

Increased interest in using Object Oriented techniques for finite element programming has resulted in a number[4][6][7][11][12][14][19] of experimental developments and implementations. Programming techniques used in some of the papers are influenced by the FORTRAN programming style. Examples provided in some of the above mentioned papers are readable by C++ experts only. It appears that none of the authors have used Object Oriented techniques for complex elastoplasticity computations.

2 nDarray Programming Tool

2.1 Introduction to the nDarray Programming Tool

The **nDarray** programming tool is a set of classes written in the C++ programming language. The main purpose of the package is to facilitate algebraic manipulations with matrices, vectors and tensors that are often found in computer codes for solving engineering problems. The package is designed and implemented using the Object Oriented philosophy. Great care has been given to the problem of cross–platform and cross–compiler portability. Currently, the **nDarray** set of classes has been tested and running under the following C++ compilers:

- Sun CC on SunOS and Solaris platforms,
- IBM xlC on AIX RISC/6000 platforms,
- Borland C++ and Microsoft C++ on DOS/Windows platforms,
- CodeWarrior C++ on Power Macintosh platform,
- GNU g++ on SunOS, SOLARIS, LINUX, AIX, HPUX and AMIGA platforms.

2.2 Abstraction Levels

nDarray tool has the following simple class hierarchy:

```
nDarray_rep, nDarray
    matrix
        vector
    tensor
```

Indentation of class names implies the inheritance level. For example, class vector is derived from class matrix, which, in turn is derived from classes nDarray and nDarray_rep. The idea is to subdivide classes into levels of abstraction, and hide the implementation from end users. This means that the end user can use the **nDarray** tool on various levels.

- At the highest level of abstraction, one can use tensor, matrix and vector objects without knowing anything about the implementation and the inner workings. They are all designed and implemented as concrete data types. In spite of the very powerful code that can be built using Object Oriented technology, it would be unwise to expect proficiency in Object Oriented techniques and the C++ programming language from end users. It was our aim to provide power programming with multidimensional data types to users with basic knowledge of C.

- At a lower abstraction level, users can address the task of the actual implementation of operators and functions for vector, matrix and tensor classes. A number of improvements can be made, especially in optimizing some of the operators.

- The lowest level of abstraction is associated with nDarray and nDarray_rep classes. Arithmetic operators[1] are implemented at this level.

Next, classes are described from the base and down the inheritance tree. Later we focus our attention on **nDarray** usage examples. Our goal is to provide a useful

[1] Like addition and subtraction.

programming tool, rather than to teach OOP or to show C++ implementation. For readers interested in actual implementation details, source code, examples and makefiles are available at `http://civil.colorado.edu/nDarray/`

2.2.1 nDarray_rep class

The nDarray_rep class is a data holder and represents an n dimensional array object. A simple memory manager, implemented with the *reference counting idiom*[3] is used. The memory manager uses rather inefficient built–in C memory allocation functions. Performance can be improved if one designs and implements specially tailored allocation functions for fast heap manipulations. Another possible improvement is in using memory resources other than heap memory. Sophisticated memory management introduced by the reference counting is best explained by Coplien.[3] The nDarray_rep class is not intended for stand–alone use. It is closely associated with the nDarray class.

The data structure of nDarray_rep introduces a minimal amount of information about a multidimensional array object. The actual data are stored as a one–dimensional array of double numbers. Rank, total number of elements, and array of dimensions are all that is needed to represent an multidimensional object. The data structure is allocated dynamically from the heap, and memory is reclaimed by the system after the object has gone out of scope.

2.2.2 nDarray class

The nDarray class together with the nDarray_rep class represents the abstract base for derived multidimensional data types: matrices, vectors and tensors. Objects derived from the nDarray class are generated dynamically by constructor functions at the first appearance of an object and are destroyed at the end of the block in which the object is referenced. The reference counting idiom provides for the object's life continuation after the end of the block where it was defined. To extend an object's life, a standard C++ compiler would by default call constructor functions, thus making the entire process of returning large objects from functions quite inefficient. By using reference counting

idiom, destructor and constructor functions manipulate reference counter which results in a simple copying of a pointer to nDarray_rep object. By using this technique, copying of large objects is made very efficient.

constructor function	description
`nDarray(int rank_of_nD=1, double initval=0.0)`	default
`nDarray(int rank_of_nD, const int *pdim, double *val)`	from array
`nDarray(int rank_of_nD, const int *pdim, double initval)`	from scalar value
`nDarray(const char *flag, int rank_of_nD, const int *pdim)`	unit nDarrays
`nDarray(const nDarray & x)`	copy-initializer
`nDarray(int rank_of_nD, int rows, int cols, double *val)`	special for matrix
`nDarray(int rank_of_nD, int rows, int cols, double initval)`	special for matrix

Table 1: nDarray constructor functions.

Objects can be created from an array of values, or from a single scalar value, as shown in Table 1. Some of the frequently used multidimensional arrays are predefined and can be constructed by sending the proper flag to the constructor function. For example by sending the "`I`" flag one creates Kronecker delta δ_{ij} and by sending "`e`" flag, one creates a rank 3 Levi-Civita permutation tensor e_{ijk}. Functions and operators common to multidimensional data types are defined in the nDarray class, as described in Table 2. These common operators and functions are inherited by derived classes. Occasionally, some of the functions will be redefined, overloaded in derived classes. In tensor multiplications we need additional information about indices. For example $C_{il} = (A_{ijk}+B_{ijk})*D_{jkl} \stackrel{\text{coded}}{\longrightarrow}$ `C=(A("ijk")+B("ijk"))*D("jkl")`, the temporary in brackets will receive ijk indices, to be used for multiplication with D_{jkl}. It is interesting to note[9] that operator `+=` is defined as a member and `+` is defined as an inline function in terms of `+=` operator.

operator or function	left value	right value	description
`=`	nDarray	nDarray	nDarray assignment
`+`	nDarray	nDarray	nDarray addition
`+=`	nDarray	nDarray	nDarray addition
`-`		nDarray	unary minus
`-`	nDarray	nDarray	nDarray subtraction
`-=`	nDarray	nDarray	nDarray subtraction
`*`	double	nDarray	scalar multiplication (from left)
`*`	nDarray	double	scalar multiplication (from right)
`==`	nDarray	nDarray	nDarray comparison
`val(...)`	nDarray		reference to members of nDarray
`cval(...)`	nDarray		members of nDarray
`trace()`	nDarray		trace of square nDarray
`eigenvalues()`	nDarray		eigenvalues of rank 2 square nDarray
`eigenvectors()`	nDarray		eigenvectors of rank 2 square nDarray
`General_norm()`	nDarray		general p-th norm of nDarray
`nDsqrt()`	nDarray		square root of nDarray
`print(...)`	nDarray		generic print function

Table 2: Public functions and operators for nDarray class.

2.2.3 Matrix and Vector Classes

The matrix class is derived from the nDarray class through the public construct. It inherits common operators and functions from the base nDarray class, but it also adds its own set of functions and operators. Table (3) summarizes some of the more important additional functions and operators for the matrix class. The vector class defines vector objects and is derived and inherits most operators and data members from the matrix class. Some functions, like copy constructor, are overloaded in order to handle specifics

operator or function	left value	right value	description
`=`	matrix	matrix	matrix assignment
`*`	matrix	matrix	matrix multiplication
`transpose()`	matrix		matrix transposition
`determinant()`	matrix		determinant of a matrix
`inverse()`	matrix		matrix inversion

Table 3: Matrix class functions and operators (added on nDarray class definitions).

of a vector object.

2.2.4 Tensor Class

The main goal of the tensor class development was to provide the implementing analyst with the ability to write the following equation directly into a computer program:

$$d\sigma_{mn} = -{}^{old}r_{ij}T^{-1}_{ijmn} - d\lambda\, E_{ijkl}\,{}^{n+1}m_{kl}T^{-1}_{ijmn}$$

as:

```
dsigma = -(r("ij")*Tinv("ijmn")) - dlambda*((E("ijkl")*dQods("kl"))*Tinv("ijmn"));
```

Instead of developing theory in terms of indicial notation, then converting everything to matrix notation and then implementing it, we were able to copy formulae directly from their indicial form to the C++ source code.

In addition to the definitions in the base nDarray class, the tensor class adds some specific functions and operators. Table 4 summarizes some of the main new functions and operators. The most significant addition is the tensor multiplication operator. With the help of a simple indicial parser, the multiplication operator contracts or expands indices and yields a resulting tensor of the correct rank. The resulting tensor receives proper indices, and can be used in further calculations on the same code statement.

operator or function	left value	right value	description
`+`	tensor	tensor	tensor addition
`-`	tensor	tensor	tensor subtraction
`*`	tensor	tensor	tensor multiplication
`transpose0110()`	tensor		$A_{ijkl} \rightarrow A_{ikjl}$
`transpose0101()`	tensor		$A_{ijkl} \rightarrow A_{ilkj}$
`transpose0111()`	tensor		$A_{ijkl} \rightarrow A_{iljk}$
`transpose1100()`	tensor		$A_{ijkl} \rightarrow A_{jikl}$
`transpose0011()`	tensor		$A_{ijkl} \rightarrow A_{ijlk}$
`transpose1001()`	tensor		$A_{ijkl} \rightarrow A_{ljki}$
`transpose11()`	tensor		$a_{ij} \rightarrow a_{ji}$
`symmetrize11()`	tensor		symmetrize second order tensor
`determinant()`	tensor		determinant of 2nd order tensor
`inverse()`	tensor		tensor inversion (2nd, 4th order)

Table 4: Additional and overloaded functions and operators for tensor class.

3 Finite Element Classes

3.1 Stress, Strain and Elastoplastic State Classes

The next step in our development was to use the **nDarray** tool classes for constitutive level computations. The simple extension was design and implementation of infinitesimal stress and strain tensor classes, namely stresstensor and straintensor. Both classes are quite similar, they inherit all the functions from the tensor class and we add some tools that are specific to them. Both stress and strain tensors are implemented as full second order 3×3 tensors. Symmetry of stress and strain tensor was not used to save storage space. Table 5 summarizes some of the main functions added on for the stresstensor class.

operator or function	description
`Iinvariant1()`	first stress invariant I1
`Iinvariant2()`	second stress invariant I2
`Iinvariant3()`	third stress invariant I3
`Jinvariant2()`	second deviatoric stress invariant J2
`Jinvariant3()`	third deviatoric stress invariant J3
`deviator()`	stress deviator
`principal()`	principal stresses on diagonal
`sigma_octahedral()`	octahedral mean stress
`tau_octahedral()`	octahedral shear stress
`xi()`	Haigh–Westergard coordinate ξ
`rho()`	Haigh–Westergard coordinate ρ
`p_hydrostatic()`	hydrostatic stress invariant
`q_deviatoric()`	deviatoric stress invariant
`theta()`	θ stress invariant (Lode's angle)

Table 5: Additional methods for stress tensor class.

Further on, we defined an elastoplastic state, which according to incremental theory of elastoplasticity with internal variables, is completely defined with the stress tensor and a set of internal variables. This definition led us to define an elastoplastic state termed class ep_state. Objects of type ep_state contain a stress tensor and a set of scalar or tensorial internal variables[2].

[2]Internal variables can be characterized as tensors of even order, where, for example, zero tensor is a scalar internal variable associated with isotropic hardening and second order tensors can be associated with kinematic hardening.

3.2 Material Model Classes

With all the previous developments, the design and implementation of various elastoplastic material models was not a difficult task. A generic class Material_Model defines techniques that form a framework for small deformation elastoplastic computations. Table 6 summarizes some of the main methods defined for the Material_Model class in terms of yield (F) and potential (Q) functions.

operator or function	description
`F`	F Yield function value
`dFods`	$\partial F/\partial\sigma_{ij}$
`dQods`	$\partial Q/\partial\sigma_{ij}$
`d2Qods2`	$\partial^2 Q/\partial\sigma_{ij}\partial\sigma_{kl}$
`dpoverds`	$\partial p/\partial\sigma_{ij}$
`dqoverds`	$\partial q/\partial\sigma_{ij}$
`dthetaoverds`	$\partial\theta/\partial\sigma_{ij}$
`d2poverds2`	$\partial^2 p/\partial\sigma_{ij}\partial\sigma_{kl}$
`d2qoverds2`	$\partial^2 q/\partial\sigma_{ij}\partial\sigma_{kl}$
`d2thetaoverds2`	$\partial^2\theta/\partial\sigma_{ij}\partial\sigma_{kl}$
`ForwardPredictorEPState`	Explicit predictor elastoplastic state
`BackwardEulerEPState`	Implicit return elastoplastic state
`ForwardEulerEPState`	Explicit return elastoplastic state
`BackwardEulerCTensor`	Algorithmic tangent stiffness tensor
`ForwardEulerCTensor`	Continuum tangent stiffness tensor

Table 6: Some of the methods in material model class.

It is important to note that all the material model dependent functions are defined as virtual functions. Integration algorithms are designed and implemented using template algorithms, and each implemented material model appends its own yield and potential

functions and appropriate derivatives. Implementation of additional material models requires coding of yield and potential functions and respective derivative functions.

3.3 Stiffness Matrix Class

Starting from the incremental equilibrium of the stationary body, the principle of virtual displacements and with the finite element approximation of the displacement field $u \approx \hat{u}_a = H_I \bar{u}_{Ia}$, the weak form of equilibrium can be expressed as[20]

$$\bigcup_m \int_{V^m} H_{I,b}\, E_{abcd}\, H_{J,d}\, dV^m\, \bar{u}_{Jc} = \bigcup_m \int_{V^m} f_a\, H_I\, dV^m \quad \text{or} \quad \left(f_{Ia}\left(\bar{u}_{Jc}\right)\right)_{int} = \lambda \left(f_{Ia}\right)_{ext}$$

where E_{abcd} is the *constitutive tangent stiffness tensor*[3]. The element stiffness tensor is recognized as

$$k^e_{aIcJ} = \int_{V^m} H_{I,b}\, {}^{tan}E_{abcd}\, H_{J,d}\, dV^m$$

This generic form for the finite element stiffness tensor is easily programmed with the help of the **nDarray** tool. A simple implementation example is provided later. It should be noted that the element stiffness tensor in this case is a four–dimensional tensor. It is the task of the assembly function to collect proper terms for addition in a global stiffness matrix.

4 Examples

4.1 Tensor Examples

Some of the basic tensorial calculations with tensors are presented. Tensors have a default constructor that creates a first order tensor with one element initialized to 0.0:

```
tensor t1;
```

Tensors can be constructed and initialized from a given set of numbers:

[3]Which may be continuum or algorithmic[8] tangent stiffness tensor

```
static double t2values[] = {  1,2,3,
                              4,5,6,
                              7,8,9  };
tensor t2( 2, DefDim2, t2values); // order 2; 3x3 tensor (like matrix)
```

Here, `DefDim2`, `DefDim3` and `DefDim4` are arrays of dimensions for the second, third and fourth order tensor[4]. A fourth order tensor with 0.0 value assignment and dimension 3 in each order ($3 \times 3 \times 3 \times 3$) is constructed in the following way:

```
tensor ZERO(4,DefDim4,0.0);
```

Tensors can be multiplied using indicial notation. The following example will do a tensorial multiplication of previously defined tensors `t2` and `t4` so that $tst1 = t2_{ij}t4_{ijkl}t4_{klpq}t2_{pq}$. Note that the memory is dynamically allocated to accept the proper tensor dimensions that will result from the multiplication[5]

```
tensor tst1 = t2("ij")*t4("ijkl")*t4("klpq")*t2("pq");
```

Inversion of tensors is possible. It is defined for 2 and 4 order tensors only. The fourth order tensor inversion is done by converting it to matrix, inverting that matrix and finally converting matrix back to tensor.

```
tensor t4inv_2 = t4.inverse();
```

There are two built–in tensor types, *Levi-Civita permutation tensor* e_{ijk} and *Kronecker delta tensor* δ_{ij}

```
tensor e("e",3,DefDim3);     // Levi-Civita permutation tensor
tensor I2("I", 2, DefDim2);  // Kronecker delta tensor
```

Trace and determinant functions for tensors are used as follows

```
double deltatrace = I2.trace();
double deltadet = I2.determinant();
```

Tensors can be compared to within a square root of *machine epsilon*[6] tolerance

[4]In this case dimensions are 3 in every order.

[5]In this case it will be zero dimensional tensor with one element.

[6]Machine epsilon (*macheps*) is defined as the smallest distinguishable positive number (in a given precision, i.e. float (32 bits), double (64 bits) or long double (80 bits), such that $1.0 + macheps > 1.0$ yields true on the given computer platform. For example, double precision arithmetics (64 bits), on the `Intel 80x86` platform yields `macheps= 1.08E-19` while on the `SUN SPARCstation` and `DEC` platforms `macheps= 2.22E-16`.

```
tensor I2again = I2;
if ( I2again == I2 )
  printf("I2again == I2  TRUE (OK)");
else
  printf("I2again == I2  NOTTRUE");
```

4.2 Fourth Order Isotropic Tensors

Some of the fourth order tensors used in continuum mechanics are built quite readily. The most general representation of the fourth order isotropic tensor includes the following fourth order unit isotropic tensors[7]

```
tensor I_ijkl = I2("ij")*I2("kl");
```

The resulting tensor `I_ijkl` will have the correct indices, $I\_ijkl_{ijkl} = I2_{ij}I2_{kl}$. Note that `I_ijkl` is just a name for the tensor, and the `_ijkl` part reminds the implementor what that tensor is representing. The real indices, $*_{ijkl}$ in this case, are stored in the tensor object, and can be used further or changed appropriately. The next tensor that is needed is a fourth order unit tensor obtained by transposing the previous one in the minor indices,

```
tensor I_ikjl = I_ijkl.transpose0110();
```

while the third tensor needed for representation of general isotropic tensor is constructed by using similar transpose function

```
tensor I_iljk = I_ijkl.transpose0111();
```

The inversion function can be checked for fourth order tensors:

```
tensor I_ikjl_inv_2 = I_ikjl.inverse();
if ( I_ikjl == I_ikjl_inv_2 )
  printf(" I_ikjl == I_ikjl_inv_2 (OK) !");
else
  printf(" I_ikjl != I_ikjl_inv_2 !");
```

Creating a symmetric and skew symmetric unit fourth order tensors gets to be quite simple by using tensor addition and scalar multiplication

[7] Remember that `I2` was constructed as the Kronecker delta tensor δ_{ij}.

```
tensor I4s  = (1./2.)*(I_ikjl+I_iljk);
tensor I4sk = (1./2.)*(I_ikjl-I_iljk);
```

Another interesting example is a numerical check of the $e-\delta$ identity[10] ($e_{ijm}e_{klm} = \delta_{ik}\delta_{jl} - \delta_{il}\delta_{jk}$)

```
tensor id = e("ijm")*e("klm") - (I_ikjl - I_iljk);
if ( id == ZERO )
  printf(" e-delta identity HOLDS !! ");
```

4.3 Elastic Isotropic Stiffness and Compliance Tensors

The linear isotropic elasticity tensor E_{ijkl} can be built from Young's modulus E and Poisson's ratio ν

```
double Ey = 20000; // Young's modulus of elasticity
double nu = 0.2;   // Poisson's Ratio
tensor E = ((2.*Ey*nu)/(2.*(1.+nu)*(1-2.*nu)))*I_ijkl + (Ey/(1.+nu))*I4s;
```

Similarly, the compliance tensor is

```
tensor D = (-nu/Ey)*I_ijkl + ((1.0+nu)/Ey)*I4s;
```

One can multiply the two and check if the result is equal to the symmetric fourth order unit tensor

```
tensor test = E("ijkl")*D("klpq");
if ( test == I4s )
  printf(" test == I4s  TRUE  (OK up to sqrt(macheps)) ");
else
  printf(" test == I4s  NOTTRUE   ");
```

The linear isotropic elasticity and compliance tensors can be obtained in a different way, by using Lamé constants λ and μ

```
double lambda = nu * Ey / (1. + nu) / (1. - 2. * nu);
double mu = Ey / (2. * (1. + nu));
tensor E = lambda*I_ijkl + (2.*mu)*I4s;         // stiffness tensor
tensor D = (-nu/Ey)*I_ijkl  + (1./(2.*mu))*I4s; // compliance tensor
```

4.4 Second Derivative of θ Stress Invariant

As an extended example of **nDarray** tool usage, the implementation for the second derivative of the stress invariant θ (Lode angle) is presented. The derivative is used for implicit constitutive integration schemes applied to three invariant material models. The original equation reads:

$$\begin{aligned}
\frac{\partial^2\theta}{\partial\sigma_{pq}\partial\sigma_{mn}} &= \\
&- \left(\frac{9}{2}\frac{\cos 3\theta}{q^4\sin(3\theta)} + \frac{27}{4}\frac{\cos 3\theta}{q^4\sin^3 3\theta}\right) s_{pq}\, s_{mn} + \frac{81}{4}\frac{1}{q^5\sin^3 3\theta}\, s_{pq}\, t_{mn} + \\
&+ \left(\frac{81}{4}\frac{1}{q^5\sin 3\theta} + \frac{81}{4}\frac{\cos^2 3\theta}{q^5\sin^3 3\theta}\right) t_{pq}\, s_{mn} - \frac{243}{4}\frac{\cos 3\theta}{q^6\sin^3 3\theta}\, t_{pq}\, t_{mn} + \\
&+ \frac{3}{2}\frac{\cos(3\theta)}{q^2\sin(3\theta)}\, p_{pqmn} - \frac{9}{2}\frac{1}{q^3\sin(3\theta)}\, w_{pqmn}
\end{aligned}$$

where:

$$q = \sqrt{\frac{3}{2}s_{ij}s_{ij}} \quad ; \quad \cos 3\theta = \frac{3\sqrt{3}}{2}\frac{\frac{1}{3}s_{ij}s_{jk}s_{ki}}{\sqrt{(\frac{1}{2}s_{ij}s_{ij})^3}} \quad ; \quad s_{ij} = \sigma_{ij} - \frac{1}{3}\sigma_{kk}\delta_{ij}$$

$$w_{pqmn} = s_{np}\delta_{qm} + s_{qm}\delta_{np} - \frac{2}{3}s_{qp}\delta_{nm} - \frac{2}{3}\delta_{pq}s_{mn} \quad ; \quad p_{pqmn} = \delta_{mp}\delta_{nq} - \frac{1}{3}\delta_{pq}\delta_{mn}$$

and the implementation follows:

```
tensor Yield_Criteria::d2thetaoverds2(stresstensor & stress)
 {
  tensor ret( 4, DefDim4, 0.0);
  tensor I2("I", 2, DefDim2);
  tensor I_pqmn = I2("pq")*I2("mn");
  tensor I_pmqn = I_pqmn.transpose0110();
  double J2D = stress.Jinvariant2();
  tensor s = stress.deviator();
  tensor t = s("qk")*s("kp") - I2*(J2D*(2.0/3.0));
  double theta = stress.theta();
```

```
  double q_dev = stress.q_deviatoric();
//setting up some constants
  double c3t    = cos(3*theta);
  double s3t    = sin(3*theta);
  double s3t3   = s3t*s3t*s3t;
  double q3     = q_dev * q_dev * q_dev;
  double q4     = q3 * q_dev;
  double q5     = q4 * q_dev;
  double q6     = q5 * q_dev;
  double tempss = -(9.0/2.0)*(c3t)/(q4*s3t)-(27.0/4.0)*(c3t/(s3t3*q4));
  double tempst = +(81.0/4.0)*(1.0)/(s3t3*q5);
  double tempts = +(81.0/4.0)*(1.0/(s3t*q5))+(81.0/4.0)*(c3t*c3t)/(s3t3*q5);
  double temptt = -(243.0/4.0)*(c3t/(s3t3*q6));
  double tempp  = +(3.0/2.0)*(c3t/(s3t*q_dev*q_dev));
  double tempw  = -(9.0/2.0)*(1.0/(s3t*q3));
  tensor s_pq_d_mn = s("pq")*I2("mn");
  tensor s_pn_d_mq = s_pq_d_mn.transpose0101();
  tensor d_pq_s_mn = I2("pq")*s("mn");
  tensor d_pn_s_mq = d_pq_s_mn.transpose0101();
  tensor p = I_pmqn  - I_pqmn*(1.0/3.0);
  tensor w = s_pn_d_mq+d_pn_s_mq - s_pq_d_mn*(2.0/3.0)-d_pq_s_mn*(2.0/3.0);
// finally
  ret = (s("pq")*s("mn")*tempss + s("pq")*t("mn")*tempst +
         t("pq")*s("mn")*tempts + t("pq")*t("mn")*temptt +
         p*tempp + w*tempw );
   return ret;
 }
```

4.5 Application to Computations in Elastoplasticity

A useful application of the previously described classes is for elastoplastic computations. If the Newton iterative scheme is used at the global equilibrium level, then in order to preserve a quadratic rate, a *consistent*, *algorithmic tangent stiffness* (ATS) tensor should be used. For a general class of three–invariant, non–associated, hardening or softening material models, ATS is defined[8] as:

$$ {}^{cons}E^{ep}_{pqmn} = R_{pqmn} - \frac{R_{pqkl}\,{}^{n+1}H_{kl}\,{}^{n+1}n_{ij}R_{ijmn}}{{}^{n+1}n_{ot}R_{otpq}\,{}^{n+1}H_{pq} + {}^{n+1}\xi_*\ h_*} $$

where

$$ m_{kl} = \frac{\partial Q}{\partial \sigma_{kl}} \quad ; \quad n_{kl} = \frac{\partial F}{\partial \sigma_{kl}} \quad ; \quad \xi_* = \frac{\partial F}{\partial q_*} \quad ; \quad T_{ijmn} = \delta_{im}\delta_{nj} + \Delta\lambda\ E_{ijkl}\,\frac{\partial m_{kl}}{\partial \sigma_{mn}} $$

$$ H_{kl} = {}^{n+1}m_{kl} + \Delta\lambda \frac{\partial m_{kl}}{\partial q_*} h_* \quad ; \quad R_{mnkl} = \left({}^{n+1}T_{ijmn}\right)^{-1} E_{ijkl} $$

A straightforward implementation of the above tensorial formula follows:

```
double Ey = Criterion.E();
double nu = Criterion.nu();
tensor Eel = StiffnessTensorE(Ey,nu);
tensor I2("I", 2, DefDim2);
tensor I_ijkl = I2("ij")*I2("kl");
tensor I_ikjl = I_ijkl.transpose0110();
tensor m = Criterion.dQods(final_stress);
tensor n = Criterion.dFods(final_stress);
double lambda = current_lambda_get();
tensor d2Qoverds2 = Criterion.d2Qods2(final_stress);
tensor T = I_ikjl + Eel("ijkl")*d2Qoverds2("klmn")*lambda;
tensor Tinv = T.inverse();
tensor R = Tinv("ijmn")*Eel("ijkl");
double h_  =  h(final_stress);
double xi_ = xi(final_stress);
double hardMod_ = h_ * xi_;
tensor d2Qodqast2 = d2Qoverdqast2(final_stress);
tensor H = m + d2Qodqast2 * lambda * h_;
```

```
//
    tensor upper = R("pqkl")*H("kl")*n("ij")*R("ijmn");
    double lower = (n("ot")*R("otpq"))*H("pq")).trace();
    lower = lower + hardMod_;
    tensor Ep = upper*(1./lower);
    tensor Eep =  R - Ep; // elastoplastic ATS constitutive tensor
```

This ATS tensor can be used further in building finite element stiffness tensors, as will be shown in our next example.

4.6 Stiffness Matrix Example

By applying a numerical integration technique to the stiffness matrix equation

$$k^e_{aIcJ} = \int_{V^m} H_{I,b}\, E_{abcd}\, H_{J,d}\, dV^m$$

individual contributions are summed into the element stiffness tensor. This process can be implemented on a integration point level by using the **nDarray** tool as

```
K  =  K  +  H("Ib") * E("abcd") * H("Jd") * weight ;
```

It is interesting to note the lack of loops at this level of implementation. However, there exists a loop over integration points which contributes stiffness to the element tensor.

5 Performance Issues

In the course of developing the **nDarray** tool, execution speed was not a priority or issue that we tried to perfect. The benefit of being able to implement and test various numerical algorithms in a straightforward manner was the main concern. The efficiency of the **nDarray** tool when compared with FORTRAN or C was never assessed. In all honesty, some of the formulae implemented in C++ with the help of the **nDarray** tool would be difficult to implement in FORTRAN or C. The entire question of efficiency of the **nDarray** as compared to FORTRAN or C codes might thus remain unanswered for the time being.

The efficiency of C++ for numerical computations has been under consideration[13] for some time now. Poor efficiency and possible remedies for improving efficiency of C++ computations has been reported in the literature[13][17].[18] Novel techniques, such as *Template Expressions*[17] can be used to achieve and sometimes surpass the performance of hand–tuned FORTRAN or C codes.

6 Summary and Future Directions

A novel programming tool, named **nDarray**, has been presented which facilitates implementation of tensorial formulae. It was shown how OOP and efficient programming in C++ allows building of new concrete data types, in this case tensors of any order. In a number of examples these new data types were shown to be useful in implementing tensorial formulae associated with the numerical solution of various elastic and elasto-plastic problems with the finite element method. The **nDarray** tool is been used in developing of the **FEMtools** tools library. The **FEMtools** tools library includes a set of finite elements, various solvers, solution procedures for non–linear finite element system of equations and other useful functions.

Acknowledgment

The authors gratefully acknowledge support by NASA Grant NAS8-38779 from Marshall Space Flight Center. The authors wish to thank Professor Carlos Felippa of University of Colorado at Boulder for introducing us to the subject of Object Oriented Programming with finite elements and Professor Egidio Rizzi of Politecnico di Milano for initially describing the benefits of such a programming tool. The authors also wish to thank reviewers for helpful comments and suggestions.

References

[1] ANSI/ISO, Washington DC. *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*, April 1995. Doc. No.

ANSI X3J16/95-0087 ISO WG21/N0687.

[2] Grady Booch. *Object Oriented Analysis and Design with Applications.* Series in Object–Oriented Software Engineering. Benjamin Cummings, second edition, 1994.

[3] James O. Coplien. *Advanced C++, Programming Styles and Idioms.* Addison – Wesley Publishing Company, 1992.

[4] Pompiliu Donescu and Tod A. Laursen. A generalized object–oriented approach to solving ordinary and partial differential equations using finite elements. *Finite Elements in Analysis and Design*, 22:93–107, 1996.

[5] Bruce Eckel. *Using C++.* Osborne McGraw – Hill, 1989.

[6] D. Eyheramendy and Th. Zimmermann. Object–oriented finite elements II. a symbolic envornment for automatic programming. *Computer Methods in Applied Mechanics and Engineering*, 132:277–304, 1996.

[7] Bruce W. R. Forde, Ricardo O Foschi, and Siegfried F. Steimer. Object – oriented finite element analysis. *Computers and Structures*, 34(3):355–374, 1990.

[8] Boris Jeremić and Stein Sture. Implicit integrations in elasto–plastic geotechnics. *International Journal of Mechanics of Cohesive–Frictional Materials*, 2:165–183, 1997.

[9] Andrew Koenig. C++ columns. *Journal of Object Oriented Programming*, 1989 - 1993.

[10] Jacob Lubliner. *Plasticity Theory.* Macmillan Publishing Company, New York., 1990. QA 931 . L939 1990 ISBN 0–02-372161-8.

[11] G. R. Miller. An object – oriented approach to structural analysis and design. *Computers and Structures*, 40(1):75–82, 1991.

[12] R. M. V. Pidaparti and A. V. Hudli. Dynamic analysis of structures using object–oriented techniques. *Computers and Structures*, 49(1):149–156, 1993.

[13] Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10(5):458–462, Sept/Oct 1996.

[14] S.-P. Scholz. Elements of an object – oriented FEM++ program in C++. *Computers and Structures*, 43(3):517–529, 1992.

[15] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison–Wesley Publishing Company, 1994.

[16] Various Authors. The C++ report: Columns on C++, 1991-.

[17] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[18] Todd Veldhuizen. Rapid linear algebra in C++. *Dr. Dobb's Journal*, August 1996.

[19] Gordon W. Zeglinski, Ray S. Han, and Peter Aitchison. Object oriented matrix classes for use in a finite element code using C++. *International Journal for Numerical Methods in Engineering*, 37:3921–3937, 1994.

[20] Olgierd Cecil Zienkiewicz and Robert L. Taylor. *The Finite Element Method*, volume 1. McGraw - Hill Book Company, fourth edition, 1991.