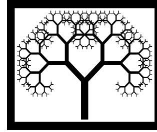
Paper 75



©Civil-Comp Press, 2012
Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

An Advanced Java Approach for the Development of Finite Element Codes

R. Saad[1] and D. Eyheramendy[2]
[1] Centrale Marseille - LMA, CNRS UPR 7051
Aix Marseille University, France
[2] Laboratoire de Mécanique et d'Acoustique, CNRS UPR 7051
Aix Marseille University, France

Abstract

In this paper, an approach to automate the elaboration of numerical tools for the simulation of complex processes by the finite element method is presented. A generic environment is developed to automate discretization schemes in the context of PDEs. The approach is implemented in a consistent object-oriented tool in Java. A symbolic environment to automatically build the symbolic forms of elemental contributions corresponding to a variational formulation is proposed. These contributions are automatically introduced into a simulation tool. The basic object-oriented framework covering the elaboration of the elemental contributions is derived from the weak statement in the context of finite element discretization is described.

Keywords: object-oriented programming, Java, finite elements, symbolic computation, automatic programming.

1 Introduction

New trends in modern software engineering open promising tracks in the design of simulation tools in computational mechanics. In the 90's, the use of the object-oriented programming has probably been the first breakthrough in the design of finite element codes. Since the pioneering works published on the subject (see e.g. Rehak [1], Miller [2], Zimmermann [3] and references therein), every domain in mechanical engineering has been targeted. The main reason to propose new programming paradigms for finite element code was maintainability and extendibility. Since these early contributions, the object-oriented paradigm has been applied in most of the domains in computational mechanics. Recently, a few authors have proposed additional structuring concepts. Among them, Nikishkov [4] has presented the basic concepts of the use of Java as an alternative for finite elements

programming. Similarly, Mackie [5] and Heng [6] have presented similar concepts based on a C# approach. Without entering the details, a few authors have proposed different Java implementation in computational mechanics or close domains: VanderHeyden [7] and Riley [8] in fluid mechanics, Nikishkov [9] [4] in mechanics and fracture mechanics, Eyheramendy [10] in parallel CFD, Eyheramendy [11] for general purpose computational mechanics, Baudel [12] in electromagnetism, Häuser [13] for parallel computational mechanics. Note that many authors have advocated the use of Java and compared the performance of the language to classical ones: Nikishkov [14], Bull [15], Häuser [13], Eyheramendy [10]. In the same time, alternative approaches have been used to fasten the development of finite element codes. Following this track, the algebraic manipulation software has been used for the development of finite element codes. In the pioneering works, Luft [16] and Noor [16] have described a methodology to automatically generate finite element matrices. The basic idea of these approaches was to rapidly develop elemental contributions based on the finite element method allowing the selection of parameter of the finite elements method: type of element, interpolation function, quadrature,… The piece of finite element code is produced for both a given target and a given number of fixed parameters. No additional hand programming is necessary in the development phase. Despite the lack of flexibility of the approach, the development process has been considerably fasten, producing bug free code, from a pure software engineering point of view of course. Ensuing these first approaches, many codes have been proposed for solving different kind of finite element problems based on symbolic computations, e.g. in electromagnetics Yagawa [18] and for solving nonlinear finite element problem Eyheramendy [19] (see Saad [20][21][22] and references therein). From our point of view, the major advance of this kind of approach is the implicit goal of modelling a class of methods in a symbolic framework rather than modelling and implementing a single specific problem. Following this major idea, Korelc [23] has more recently presented a set of tools in Mathematica providing a global framework for deriving FE models. In the same time, Eyheramendy and Zimmerman [24] have proposed a standalone object-oriented framework in Smalltalk to manipulate variational formulations (continuous and discrete) and to automatically generate finite element code for a targeted code (a C++ finite code see Dubois [25]). Similarly, Logg & al. [26] have introduced the project FEniCS with the explicit goal of developing software for the automation of Computational Mathematical Modelling, including an automation of the finite element method. A brick of the global finite element library enables the user to manage the variational statement and its discretization to feed the finite element library. These three approaches have in common to deal with a variational approach and to consider tensor analysis to build finite element contributions. The main difference is the basic framework used to develop the symbolic finite element core. As matter of fact, the way to manipulate the variational form is quite different from an approach to the other one.

The approach we propose belong to the same family. Here, we anticipate on the complete integration of symbolic concepts for finite elements including algorithmic aspects of the method coming from the global solution schemes (time integration, nonlinearities treatment…) that drastically interfere with the finite element model.

The approach we propose is fully integrated within a Java paradigm (see Eyheramendy [10][11] for the basic finite element framework). We relied on classical mathematical formalism to build discrete terms of variational statements.
In section 2, we describe the object-oriented paradigm developed in Java to manage the generic tensor approach we have adopted for describing the finite element scheme. In section 3, we propose a step-by-step treatment of a finite element problem: a 3D Galerkin Least-Squares formulation for a steady-state Navier-Stokes flow.

2 A object-oriented tensor approach for the finite elements discretization

2.1 Mathematical modeling

We consider with a standard variational formulation of a partial differential equation given as follows:

$$\begin{cases} \text{Find } u \in V \text{ such that } \forall v \in \hat{V} \\ a(v,u) = L(v) \end{cases} \tag{1}$$

where:
$a : \hat{V} \times V \rightarrow \mathbb{R}$ is a bilinear form,
$L : \hat{V} \rightarrow \mathbb{R}$ a linear form and
$(\hat{V}, V)$ a pair of suitable function spaces.

The finite element method applied to equation (1) consists in replacing $(\hat{V}, V)$ with a pair of piecewise polynomial discrete function spaces. Considering $\{\hat{\varphi}_i\}_{i=1}^{M}$ as a basis for the test function space $\hat{V}$ and $\{\varphi_i\}_{i=1}^{M}$ a basis for the solution space V, we can express the approximate solution u using the basis functions of V over an element K, $u^K = \sum_{j=1}^{M} \xi_j \varphi_j$, where the $\{\xi_i\}_{i=1}^{M}$ are the degrees of freedom corresponding to the approximate solution U. It can be shown that the discretization of the variational form leads to the following elemental contribution:

$$A_i^K = \int_K \prod_{j=1}^{m} D^j \varphi_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}(x)\, dx \tag{2}$$

with $i = (i_1, \dots, i_r)$ where r represents the number of basis functions, m is the number of factors in the bilinear form a_K, D is the differential operator applied on the field, κ corresponds to the combinations of interpolation functions and degree of freedom, α corresponds to the degree of freedom, β corresponds to the basis coordinate system and γ gives the suitable set of κ, α and δ. D represents the differential operator applied to a field φ. The field φ might be a constant or a field that can be either discretized or not. We pose: $D^j \varphi_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}(x) = H^j_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}$.
The equation (2) becomes:

$$A_i^K = \int_K \prod_{j=1}^{m} H^j_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}\, dx \qquad (3)$$

The central idea of our approach is, first, to symbolically compute $H^j_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}$ and, second, to compute the tensor products (product, contract product,…). The implementation of the symbolic computation of the elemental contribution is directly based on the equation (3). First, each term corresponding to a the discrete form (written under the tensor form $H^j_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}$) is computed, and second the convenient product is used to compute the elemental contribution A_i^K. More details can be found in Saad [20][21][22].

This form is general enough to be applied to any kind of variational statements (multi-linear forms or nonlinear forms). A linearization approach can lead to a similar bilinear form. It is important to note that the formalism is slightly different than the one proposed by Logg [26].

2.2 Object-oriented implementation

In this section, we describe the main objects needed for symbolic manipulations. First of all, we describe the classes needed to represent any kind of expression. Additional classes are needed to represent partial differential operators applied to tensors and partial derivative applied to functions. The class hierarchy is shown in Figure 1.

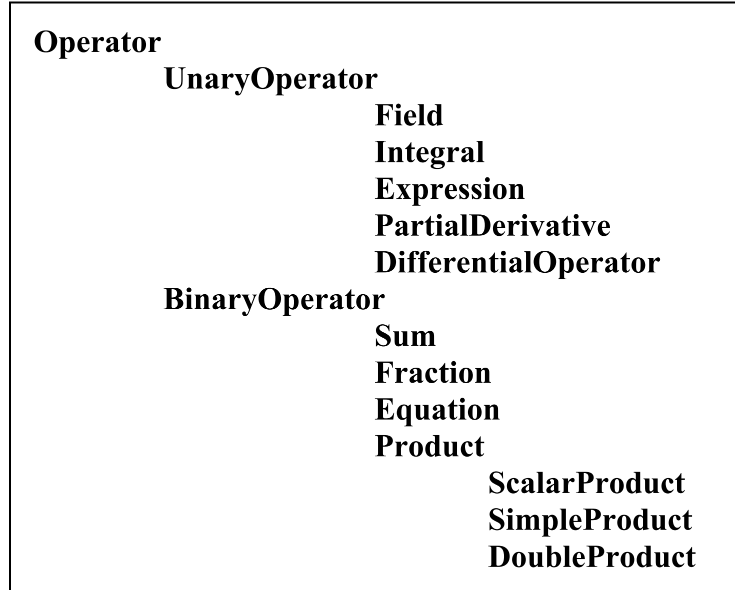


Figure 1: Class hierarchy of operators

The class **Operator** enables the tree structuration of expressions. It manages the various operations. This class is the super class of other classes. It is an abstract class. The methods are implemented in derived classes. An instance of the class **Field** is able to analyze itself, to define the kind of field (scalar, vector or tensor), to give its order and its dimensions. Both classes **PartialDerivative** and **DifferentialOperator** are used to represent respectively the partial derivatives applied to a function and differential operators applied to a field. To define a complex expression, we introduce the class **Expression**. This is to describe and represent any kind of expression (shape function, variational formulation…). The class **BinaryOperator** inherits from **Operator** class. It is used to represent binary operators. A binary operator has two operands a priori non-commutative, a right-hand-side one and a left-hand-side one. This class is an abstract class. The class **Sum** is derived from the **BinaryOperator** class. It represents any kind of sum (between scalars, vectors or tensors). The class **Product** is the generic class that will define all products. In contrast to the sum, products are differentiated according to several types (scalar product, contracted product,…). The class **Product** is abstract, and the methods are implemented in the three subclasses **ScalarProduct**, **SimpleContraction**, **DoubleContraction**. The class **Fraction** is derived from the class **BinaryOperator**. It represents a rational fraction. The class **Integral** is derived from the class **UnaryOperator**. It inherits the attribute **operator** that is the root of an expression. Its second attribute describes the domain of integration. The class **Equation** is derived from the class **BinaryOperator**. It represents the equality between the integral equations (right and left-hand-sides).

At last, two additional classes are needed. The class **Tensor** is the representation of a mathematical tensor. It contains its components that are expressions (binary tree of fields). Basic tensor operations are performed at this level: tensor contraction, tensor product, sum…). In the class **Discretization**, we build the tensor corresponding to the finite element discretization. We based our discretization method on the building of the tensor A_i^K. Note that introducing a new discretization scheme means designing a new discretization class.

3 A step-by-step derivation of a mixed stabilized variational formulation: Application to the incompressible Navier-Stokes equations

3.1 Strong form and variational statement

We consider the formulation of the steady-state 3D Navier-Stokes equations for an incompressible Newtonian fluid. We seek the velocity u and pressure p such that the equilibrium equations, the boundary conditions and incompressibility conditions are met. Using classical notations (see e.g. Tezduyar [27]), the equations of the problem are:

$$\begin{cases} div\,\sigma + f = \rho u \nabla u & \text{on } \Omega \\ div\, u = 0 & \text{on } \Omega \\ \sigma = -pI + 2\mu\varepsilon(u) & \text{on } \Omega \\ \varepsilon(u) = \frac{1}{2}(\nabla u + \nabla^T u) & \text{on } \Omega \\ u = \bar{u} & \text{on } \partial\Omega_1 \\ \sigma.n = F & \text{on } \partial\Omega_2 \end{cases}$$

A mixed Galerkin finite element formulation (velocity-pressure), stabilized by addition of terms like least-squares (see Tezduyar [27]) is adopted to build the numerical model. It is given as follows:

$$\begin{cases} \text{Find } (u^h, p^h) \subset ((\mathcal{S}^h)_n \times (\mathcal{P}^h)_n)\ \forall\, (w^h, q^h) \subset ((\mathcal{W}^h)_n \times (\mathcal{P}^h)_n) \text{ such as :} \\ \int_\Omega \rho[(u^h.\nabla u^h).w^h]\,dv + \int_\Omega 2\mu[\varepsilon(u^h):\varepsilon(w^h)]\,dv + \int_\Omega divu^h q^h\,dv - \int_\Omega p^h divw^h\,dv - \int_\Omega f.w^h\,dv \\ + \sum_{\Omega^e \in \Omega^h} \left[\int_{\Omega^e} [(\rho u^h.\nabla u^h - 2\mu\varepsilon(u^h) + \nabla p^h - f)\tau_{mom}(\rho u^h.\nabla w^h - 2\mu\varepsilon(w^h) + \nabla q^h)]\,dv \right] = 0 \end{cases}$$

where the stabilization parameter is: $\tau_{mom} = \left(\left(\frac{2|u|}{h}\right)^2 + \left(\frac{4\mu|}{h^2}\right)^2 \right)^{1/2}$

and considering the variational spaces:

$(\mathcal{S}^h)_n = \{u^h \epsilon [H^1(Q_n)]^h | u^h = \bar{u}\ sur\ (P_n)_{\bar{u}}\}$

$(\mathcal{W}^h)_n = \{u^h \epsilon [H^1(Q_n)]^h | u^h = 0\ sur\ (P_n)_{\bar{u}}\}$

$(\mathcal{P}^h)_n = \{u^h \epsilon [L_2(Q_n)]^h |\}$

3.2 Definition of the *H* tensor

We can apply the formalism developed in the previous paragraph to this formulation. We consider the term $a_1(u,w) = \int_\Omega \rho[(u.\nabla u).w]\,dv$ (convection term) as illustration. This term leads to the following elemental contribution:

$$A_i^K = \int_K \prod_{j=1}^{m} H^j_{\gamma_j(\alpha,\beta,\kappa_{(i,\beta)})}\,dx$$

For the considered here, the previous equation holds:

$$A_{1,\kappa}^K = \int_K \rho[C_{\alpha_1}][\varphi_{\alpha_1,\kappa_1,\beta_1}][\varphi_{\beta_2,\kappa_2}]\ dx$$

Each term of $A_{1,\kappa}^K$ is explicitly computed in the symbolic environment. Computing the different products, we finally obtain the tensor *H*.

$$\begin{aligned} A_{1,\kappa}^K &= \int_K \rho[C_{\alpha_1}][\varphi_{\alpha_1,\kappa_1,\beta_1}][\varphi_{\beta_2,\kappa_2}]\ dx = \int_K \rho[Q_{\kappa_1,\beta_1}][\varphi_{\beta_2,\kappa_2}]\ dx \\ &= \int_K [H_{\kappa_1,\kappa_2}]\ dx = \int_K [H_{(i_1,\beta_1),(i_2,\beta_2)}]\ dx \\ &= \int_K \left[H_{((N_1,\dots,N_8),(u_1,u_2,,u_3)),((N_1,\dots,N_8),(u_1,u_2,,u_3))}\right]\ dx \end{aligned}$$

The tensor *H* with dimension 24×24 can be expressed as $H_{\kappa_1,\kappa_2} =$

$$
\begin{pmatrix}
\rho N_1\left(u_1\frac{\partial N_1}{\partial x_1}+u_2\frac{\partial N_1}{\partial x_2}\right) & 0 & \cdots\cdots\cdots & \rho N_1\left(u_1\frac{\partial N_4}{\partial x_1}+u_2\frac{\partial N_4}{\partial x_2}\right) & 0 \\
0 & \rho N_1\left(u_1\frac{\partial N_1}{\partial x_1}+u_2\frac{\partial N_1}{\partial x_2}\right) & 0 & 0 & \rho N_1\left(u_1\frac{\partial N_4}{\partial x_1}+u_2\frac{\partial N_4}{\partial x_2}\right) \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\rho N_1\left(u_1\frac{\partial N_2}{\partial x_1}+u_2\frac{\partial N_2}{\partial x_2}\right) & 0 & \ddots & \rho N_4\left(u_1\frac{\partial N_4}{\partial x_1}+u_2\frac{\partial N_4}{\partial x_2}\right) & 0 \\
0 & \rho N_4\left(u_1\frac{\partial N_1}{\partial x_1}+u_2\frac{\partial N_1}{\partial x_2}\right) & \cdots\cdots\cdots & 0 & \rho N_4\left(u_1\frac{\partial N_4}{\partial x_1}+u_2\frac{\partial N_4}{\partial x_2}\right)
\end{pmatrix}
$$

The same developments for terms a_2 up to a_8 can be performed to obtain respectively the tensors $A^K_{2,\kappa}$,…,$A^K_{8,\kappa}$, corresponding to all the elemental contibutions of the problem.

3.3 Step-by-step derivation of the discretization of the variational formulation

We consider the step-by-step implementation the symbolic finite element matrices for the formulation presented in the previous section.

3.3.1 Solution and test function definitions

In Figure 1, 8 interpolation functions corresponding to a trilinear 8-nodes element are defined: N_1 to N_8. The fields u (3D vector, based on the interpolation previously defined), w (3D vector, test function), p (scalar), q (scalar, test function) and f (the body loads) are defined. The constants ρ and μ are then repctively initialed to *200 kg/m*^{3} and 1.0 *N s/m*^{2}. The constitutive tensor is then constructed based on a predefined method called *giveConstitutiveTensor(double,double)*.

3.3.2 Definition of the terms of the variational formulation

In Figure 2, the different terms of the formulation are built: ∇u, $\varepsilon(u)$, div(u),… From a practical point of view, the differential operators are applied to previously defined fields.

3.3.3 Discretization of the variational terms

In Figure 3, we build for each term tensor corresponding to its discretization. This tensors will be used to build the H tensor corresponding. E.g., the elemental contribution corresponding to the advection term described in section 3.2 is built in section A1 in Figure 3.

```
Field N1 = new Field("N1", null) ; //N_1
Field N2 = new Field("N2", null) ; //N_2
Field N3 = new Field("N3", null) ; //N_3
Field N4 = new Field("N4", null) ; //N_4
Field N5 = new Field("N5", null) ; //N_5
Field N6 = new Field("N6", null) ; //N_6
Field N7 = new Field("N7", null) ; //N_7
Field N8 = new Field("N8", null) ; //N_8
Field u = new Field("u", new int[] { 3 }, 3, true,
                 new Operator[]{N1,N2,N3,N4,N5,N6,N7,N8});//u_i
Field w = new Field("w", new int[] {3}, 3, true,null);//w_i
Field p = new Field("p", null, 3, true,
                 new Operator[]{N1,N2,N3,N4,N5,N6,N7,N8});//p
Field q = new Field("q", null, 3, true,null); //q
Field f = new Field("f", new int[] { 3 }, 3); //f
Fraction rho = new Fraction(200.0,1.0) ; //ρ
Fraction mu = new Fraction(1.0,1.0) ; //μ
Tensor C = u.giveConstitutiveTensor(lambda,mu);//C_jilk
```

Figure 1: Fields and constant definitions

```
DifferentialOperator gradu=new DifferentialOperator("grad",u);
DifferentialOperator Eu =new DifferentialOperator("E", u);
DifferentialOperator divu=new DifferentialOperator("div", u);
DifferentialOperator gradp=new DifferentialOperator("grad",p);
DifferentialOperator divw = new DifferentialOperator("div",w);
DifferentialOperator gradw=new DifferentialOperator("grad",w);
DifferentialOperator Ew = new DifferentialOperator("E", w);
DifferentialOperator gradq=new DifferentialOperator("grad",q);
```

Figure 2: Definition of the variational terms

3.4 Generation of the finite element code

The implementation of the formulation is performed through a class called **NavierStokesFormulation**. The class is given in Figure 4. The class is a subclass of class **Formulation** which manages the basic behavior of the integration of the formulation into the finite element code. The class **NavierStokesFormulation** is equipped with a static inner class called **NavierStokes**. The latter is a sublclass of class **Element** which manages the basics actions to manipulate finite elements. The main methods of the class are:

• *initialize ()* which is used to describe the unknown field, pressure and velocity (the pressure is a scalar field and the velocity is a vector field).

• *getElement ()* which enables us to locally instantiate the element called **NavierStokes**. The element is defined from data corresponding to the problem: numerical quadrature, material ...

The class **NavierStokes** contains all the methods needed to compute the contributions related to the finite element formulation. This implementation is based on the concept of inner class allows the programmer to define data at the same level for the global and local problem, and within the same class. This ensures

consistency between the element and the formulation. The elemental matrices are computed in method *computeElementalMatrices(TimeStep ts)*. In this method, as shown in Figure 4, the methods called *compute1()*,*compute2()*,…,*compute8()* permits to compute the 8 elemental contribution of the formulation. The method *compute1()* is shown in Figure 5.

```
Tensor Tq = q.giveDiscretizationTensor();
Tensor Tp = p.giveDiscretizationTensor();
Tensor Tw = w.giveDiscretizationTensor();
Tensor Tgradu = gradu.giveDiscretizationTensor();
Tensor Tgradw = gradw.giveDiscretizationTensor();
Tensor Tdivu = divu.giveDiscretizationTensor();
Tensor Tdivw = divw.giveDiscretizationTensor();
//----------------------------A1----------------------------
Tensor TuTgradu = Tu.simpleContraction(Tgradu);
Tensor TugraduTw = TuTgradu.simpleContraction(Tw);
Tensor rhoTugraduTw = rho.scalarProduct(TugraduTw);
//----------------------------A2----------------------------
Tensor TEu = Eu.giveDiscretizationTensor();
Tensor TEw = Ew.giveDiscretizationTensor();
Tensor TEuTEw = TEu.doubleContraction(TEw);
Fraction deux = new Fraction(2,1);
Fraction deuxmu = deux.multiplication(mu);
Tensor deuxmuTEwTEu=deuxmu.scalarProduct(TEuTEw);
//----------------------------A3----------------------------
Tensor TdivuTq = Tdivu.tensorProduct(Tq);
                                     ⋮
//----------------------------A8---------------------------
Tensor tauxTgradqTgradp = tauxTgradq.simpleContraction(Tgradp);
Tensor unSurRhotauxTgradqTgradp =
                    tauxTgradqTgradp.scalarProduct(unSurRho);
//---------------------Tenseurs-----------------------
Tensor[] TSymbolicLeft = new Tensor[] { rhoTugraduTw,
                         deuxmuTEwTEu, TdivuTq, moinspTdivw};
Tensor[] TSymbolicright = new Tensor[] { moinsfv};
//---------------------Formulation-----------------------
Formulation g = new Formulation("NavierStockes", new
             Equation[]{equation}, new Material (lambda, mu),
           leftSymbolicTensor, rightSymbolicTensor);
```

Figure 3: Partial view of construction of the *H* tensors

3.5 A numerical application: the 3D driven cavity flow

We consider the 3D driven cavity flow problem given in Figure 6. We impose a $1\ m.s^{-1}$velocity on one side of the cavity. . The material characteristics are: volume density $\rho = 200\ Kg/m^3$ and dynamic viscosity $\mu = 1\ Pa$.s. The Reynolds number for the problem is $Re = 200$. In Figure 7, we show the numerical solution of the problem: the velocity field and the pressure field.

```
public class LinearizedStabNSFormulation extends Formulation {
       …
       public void initialize(Domain domain) {
         Field[] fields = new Field[2];
         fields[0] = domain.createAVector2DField(0);
         fields[1] = domain.createAScalarField(1);
         //…
         this.finalizeDomain(domain);
       }

       public static class LinearizedStabilizedNavierStokes extends
            Element { …
            Public Hashtable computeElementalMatrices(TimeStep ts){
            ElementalBlockMatrix stiffness = new
                   ElementalBlockMatrix(Kuu, Kuw, Kwu, Kww,this);
            ElementalBlockMatrix load = new
                           ElementalBlockMatrix(fu, fw, this);
            for(int i = 0;i< quadrature.numberOfGaussPoints();i++){
            //…
            ElementalFullMatrix T1 = compute1(shape1, shapeDer1)
            Kwu.plusSimpleProduct(T1,volume);
            //…
       }
   //…
   }
```

Figure 4: Partial view of class **NavierStokesFormulation**

```
private ElementalFullMatrix compute1(double[] shapeDer,
              double [] constant){
  ElementalFullMatrix answer= new ElementalFullMatrix(8,8);
  double dN1_dx1=shapeDer[0];

  double dN4_dx2=shapeDer[7];
  double lambda = constant[1];
  double mu = constant[2];
  double T_1_1=dN1_dx1*(lambda + 2.0*mu)*dN1_dx1 +
     (0.5*(dN1_dx2)* mu + 0.5*(dN1_dx2)* mu)*0.5*(dN1_dx2) +
     (0.5*(dN1_dx2)* mu + 0.5*(dN1_dx2)* mu)*0.5*(dN1_dx2);

  double T_8_8=(0.5*(dN4_dx1)*mu + 0.5*(dN4_dx1)*mu)*
     0.5*(dN4_dx1)+(0.5*(dN4_dx1)*mu + 0.5*(dN4_dx1)*mu)*
     0.5*(dN4_dx1) + dN4_dx2*( lambda + 2.0*mu)*dN4_dx2;
     answer.atPut(1,2, T_1_1);

     answer.atPut(1,2, T_8_8);
     return answer;}
```

Figure 5: Partial view of method *compute1()*

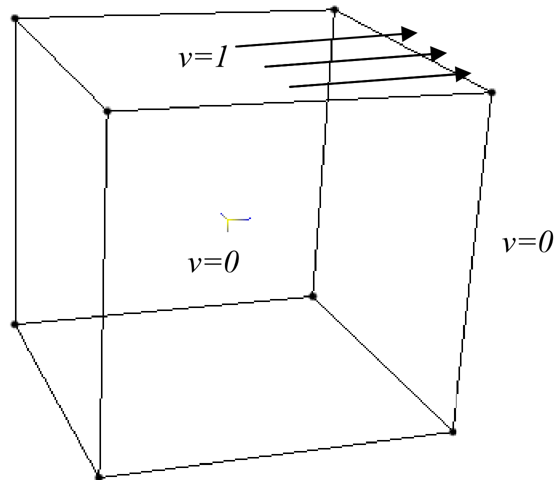


Figure 6: 3D driven cavity flow problem

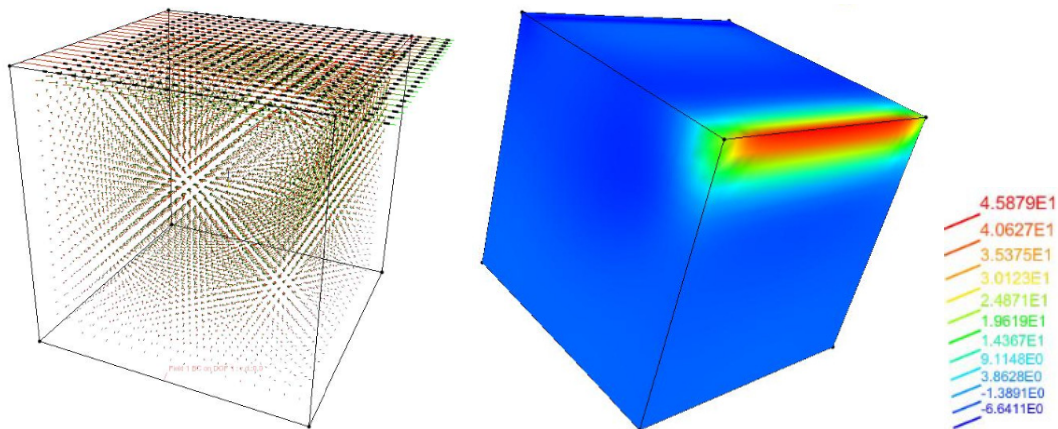


Figure 7: Velocity field and pressure iso-values for the 3D cavity flow problem

4 Conclusion

In this paper, we have first presented the key concepts to model the discretization of the variational terms. The mathematical formalism is based on tensor algebra to describe the discretization of a variational formulation. Secondly, we have proposed an object-oriented integration of this model within a Java finite element code. The generic character of the approach is preserved through this Java object-oriented approach. The symbolic finite element contributions are then added as classical “elements”. To a certain extent, this tool aims at providing a generic model for variational formulations, replacing all the so-called classical “elements”. The advantage of the symbolic approach is that the generic description that can be extended naturally to any discretization model in space or time. We have illustrated

the theoretical developments on a simple nonlinear Galerkin Least-Squares formulation for the steady-state Navier-Stokes equations. This concept is fully validated (see Saad [20]) for simple linear problems (elasticity, heat convection, etc.), for the treatment of mixed variational formulations (thermo-mechanical, Navier-Stokes for incompressible flows...) and for Lagrangian frameworks (elasticity in large transformations, hyperelasticity, etc.) This kind of approach aims at defining high abstraction level concept to automatically produce a computational tool without programming a single line of code. We have restricted our study to the modeling of variational forms, which only enables us to produce the standard element. In a short future, our aim is to go beyond the classical frontiers of finite element codes by including the whole algorithmic framework of a code. This generic approach opens new tracks in the design of new types of code, based on the mathematical model and no more on simple applications.

References

[1] Rehak D.R. and Baugh Jr. J.W., Alternative Programming Techniques for Finite Element Programming Development, Proceedings IABSE Colloquium on Expert Systems in Civil Engineerings, Bergamo, Italy. IABSE, (1989).

[2] G.R Miller., A LISP-Based Object-Oriented approach to structural analysis, Engr. with Comp., vol. 4 (1988) pp. 197-203.

[3] Th. Zimmermann, Y. Dubois-Pèlerin and P. Bomme, Object-oriented finite element programming : I. Governing principles, Comput. Methods Appl. Mech. Engrg., vol. 98 (1992) pp. 291-303.

[4] G.P.Nikishkov, “Object-oriented design of a finite element code in Java”, Computer Modeling in Engng and Sciences, 11, 81-90, 2006.

[5] R.I. Mackie, “Using Objects to handle complexity in Finite Element Software”, Eng. with computers, 13, 99-111, 1997.

[6] B.C.P. Heng and R.I. Mackie, “Design Patterns in Object-Oriented Finite Element Programming”, in Proceedings of the Fifth International Conference on Engineering Computational Technology, B.H.V. Topping, G. Montero and R. Montenegro, (Editors), Civil-Comp Press, United Kingdom, 2006.

[7] W.B. VanderHeyden, E.D. Dendy and N.T. Padial-Collins, “CartaBlanca-a pure-Java, component-based systems simulation tool for coupled nonlinear physics on unstructured grids-an update”, Concurrency and Computation: Practice and Experience, 15, 431-458, 2003.

[8] C.J. Riley, S. Chatterjee and R. Biswas, “High-performance Java codes for computational fluid dynamics”, Concurrency and Computation: Practice and Experience 15, 395-415, 2003.

[9] G.P. Nikishkov and H. Kanda, “The development of a Java engineering application for higher-order asymptotic analysis of crack-tip fields”, Advances in Engineering Software, 30, 469-477, 1999.

[10] D. Eyheramendy, “Object-oriented parallel CFD with JAVA”, 15th International Conference on Parallel Computational Fluid Dynamics, Eds. Chetverushkin, Ecer, Satofuka, Périaux, Fox, Ed. Elsevier, 409-416, 2003.

[11] D. Eyheramendy and D. Guibert, "A Java Approach for Finite Elements Computational Mechanics", ECCOMAS 2004, Jyvaskyla, Finland, July 2004.
[12] L. Baduel, F. Baude, D. Caromel, C. Delbé, N. Gama, S. El Kasmi and S. Lanteri, "A parallel object-oriented application for 3-D electromagnetism", ECCOMAS, Jyväskylä, Finland 2004.
[13] J. Häuser, T. Ludewig, R.D. Williams, R. Winkelmann, T. Gollnick, S. Brunett and J. Muylaert, "A test suite for high-performance parallel Java", Advances in Engineering Software, 31, 687-696, 2000.
[14] G.P. Nikishkov, Y.G Nikishkov and V.V Savchenko, "Comparison of C and Java performance in finite element computations", Computer & Structures, 81, 2401-2408, 2000.
[15] J.M. Bull, L. A. Schmith, L. Pottage and R. Freeman, "Benchmarking Java against C and Fortran for Scientific Applications", Joint ACM JavaGrande – ISCOPE 2001 Conference, Stanford Universtity, June 2-4, 2001.
[16] R.W. Luft, J.M. Roesset and J.J. Connor, Automatic generation of finite element matrices, Struct. Div., Proceedings of ASCE, January 1971, (1971) pp. 349-361.
[17] A.K. Noor, C.M. Andersen, Computerized symbolic manipulation in nonlinear finite element analysis, Computers & Structures vol. 13 (1981) pp. 379-403.
[18] G. Yagawa, G.-W. Ye and S. Yoshimura, A numerical integration scheme for finite element method based on symbolic manipulation, Internat. J. Numer. Methods Engrg., vol. 29 (1990) pp. 1539-1549.
[19] D. Eyheramendy, Th. Zimmermann, The Object-oriented finite elements: II. A symbolic environment for automatic programming, Comput. Methods Appl. Mech. Engrg., vol. 32 (1996) pp. 259-276.
[20] R. Saad, Sur une approche à objets generalise pour la mécanique non linéaire, PhD Thesis report, Université de Provence, 2011.
[21] R. Saad and D. Eyheramendy, Generalized finite element discretization: an object-oriented analysis, ECCM 2010, Paris 21/26-05-2010.
[22] R. Saad and D. Eyheramendy, An object-oriented framework for automated computer-aided finite element derivation, ECT 2010, 14-17 Sept. 2010, Valencia, Spain.
[23] J. Korelc, Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes, Engineering with Computers, vol. 18 (2002) pp. 312-327.
[24] D. Eyheramendy, Th. Zimmermann, "Object-oriented finite elements: III. Theory and application of automatic programming", Comput. Methods Appl. Mech. Engrg., 154, 41- 68, 1998.
[25] Y. Dubois-Pelerin and Th. Zimmermann, "Object-oriented finite element programming: III – An efficient implementation in C++", Computer Methods Appl. Mech. Eng., 108(2), 165-183, 1993.
[26] A. Logg, Automating the Finite Element Method, Arch Comput Methods Eng., vol. 14 (2007).
[27] T.E. Tezduyar, Stabilized finite element formulations for incompressible flow computations, Advances in applied Mechanics, vol. 28 (1992) pp. 1-44