OBJECT-ORIENTED FINITE ELEMENT PROGRAMMING: SYMBOLIC DERIVATIONS AND AUTOMATIC PROGRAMMING

THÈSE N^{o} 1752 (1997)

PRÉSENTÉE AU DÉPARTEMENT DE GÉNIE CIVIL

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Dominique EYHERAMENDY
Ancien élève de l'E.N.S. Cachan, France, et agrégé de mécanique
de nationalité française

acceptée sur proposition du jury:

Dr Th. Zimmermann, directeur de thèse
Prof. E. Anderheggen, corapporteur
Prof. A. Combrescure, corapporteur
Prof. F. Frey, corapporteur
Mr C. Hauviller, corapporteur
Prof. J.-P. Pelle, corapporteur
Dr M. Picasso, corapporteur
Prof. J. Rappaz, corapporteur

Lausanne, EPFL
1997

To my friends ...
To my parents ...

Acknowledgements

I would like to gratefully thank all the people who have permitted me to achieve this work:

- Dr. Thomas Zimmmermann, for his guidance over the years I spent in Lausanne, and for his enriching criticism through my hard days of work and for giving me the opportunity to internationally widen my horizons

- Prof. Francois Frey, Director of the LSC, for making this research possible

- the members of the jury for accepting to examine this work

- Prof. Jean-Pierre Pelle and Prof. Pierre Ladevèze, for their confidence

- Prof. Robert Arrieux and Prof. Max Giordano and all the members of the LMécA, Ahmed Haddad, Eric Pairel, Bernard Riveill-Reydet, Serge Samper, Vincent Simeneau, Laurent Tabourot and Pierre Vacher, at the Ecole Supèrieure d'Ingénieurs d'Annecy, for their advice and help during my activity at the ESIA

- Yves-Dominique Dubois-Pèlerin and Patricia Bomme for introducing me to the object-oriented paradigm and for many interesting discussions

- the members of the LSC, past and present, among them Mohamed Amieur, Wajd Atamaz Sibaï, Ahmadou Barry, David Alvarez-Debrot, Françoise Delaray, José Diaz, Stéphane Commend, César Falla-Luque, Wadah Farra, Richard Frenette, Evelyne Guidetti, Adnan Ibrahimbegovic, Prof. Jaroslav Jirousek, Milan Jirasek, Yijun Li, Krzysztof Podles, Blaise Rébora, Pierrette Rosset, Bernard Rutscho, Jean-Luc Sarf, Birgitt Seem, Malgorzata Stojek, Marc-André Studer, Andrzej Truty, Laurent Vernier, Adam Wroblewski,... for their friendship and for so many fruitful discussions

- Prof. Tayfun Tezduyar and his team, Shahrouz Aliabadi, Marek Behr, Ismail Güler, Andrew Johnson, Vinay Kalro, Steve Ray and Chris Water, at the University of Minnesota, for their welcome in Minneapolis during a cold winter and for numerous fruitful discussions on Computational Fluid Dynamics

- the Department of Mechanics of the Ecole Normale Supérieure de Cachan, led first by Prof. Didier Marquis and then by Prof. Françoise Léné, for the 4th year I was allowed to complete at the LSC, and more especially Danielle Babic for the efficiency of her frequent help

- the Swiss National Science Foundation, for grants N^{0} 21-40290.94 and N^{0} 20-45697.95

Résumé

Les nouvelles technologies dans le domaine de l'informatique appliquée au calcul numérique autorisent aujourd'hui des approches alternatives dans la résolution de problèmes de mécanique par la méthodes des éléments finis (E.F.) . Dans les approches classiques, les développements théoriques conduisent souvent à de lourds calculs pour un problème donné suivi du développement d'un modèle informatique pour la résolution de problèmes pratiques. La première étape du développement est l'analyse du problème physique que l'on souhaite simuler. Ce problème est généralement décrit par un ensemble d'équations, incluant des équations aux dérivées partielles. Ce premier modèle est ensuite remplacé par une suite de modèles équivalents et/ou approchés. Le résultat se réduit généralement à quelques pages décrivant les algorithmes et les matrices élémentaires pour le problème, le tout exprimé dans un langage mathématique "simple" malgré tout. La démarche traditionnelle consiste alors à élaborer un outil informatisé, en général complexe et relativement éloigné des descriptions mathématiques. Le problème de l'architecture du logiciel et du langage utilisé dans les développements se pose alors de manière cruciale. Rompant partiellement avec cette approche, on propose une nouvelle manière de programmer et de développer les formulations E.F. S'appuyant sur une approche hybride symbolique/numérique pour la résolution de problème en mécanique et sur un outil informatique de haut niveau, la programmation orientée objet (O.O.) (ici les langages Smalltalk et C++), l'objectif de ce travail a été de développer un environnement capable d'effectuer d'une part, des manipulations algébriques nécessaires à l'application d'une formulation E.F. à un problème posé sous forme différentielle, et d'autre part, des calculs numériques efficaces. Ainsi, l'environnement créé est capable de gérer tous les concepts nécessaires à la solution de problèmes physiques : manipulations d'équations aux dérivées partielles, formes variationnelles, intégration par parties, formes faibles, approximations E.F... Les concepts manipulés sont très proches des concepts mathématiques. Le résultat de ces opérations algébriques est un ensemble de données élémentaires (matrices de rigidité, de masse, tangentes, ...) à introduire dans un code numérique classique. L'apport du paradigme objet est essentiel à ce travail. Dans le contexte des codes E.F., ce type d'approches a déjà prouvé sa capacité à représenter des structures et phénomènes complexes. Dans l'environnement symbolique pour la dérivation de formulations E.F. , dans lequel des objets tels que l'expression, l'intégrale et la forme variationnelle apparaissent, ceci est confirmé. Le besoin de lien entre le monde numérique (code E.F.) et le monde symbolique a permis d'établir un concept O.O. pour la programmation automatique de formes matricielles symboliques dérivées de la méthode E.F. Le résultat est un environnement global dans lequel le mécanicien est capable d'évoluer naturellement, en utilisant un langage proche de son langage naturel. Le potentiel de l'approche est mis en évidence, d'une part par la variété des problèmes abordés, aussi bien dans le domaine de la mécanique linéaire (élasticité en dynamique 1D et 2D, thermique 2D,...), que non-linéaire (problèmes à convection dominante 1D, écoulement de Navier-Stokes), et d'autre part par le type de formulations manipulées (formulations de Galerkin, formulations de Galerkin espace/temps continues en espace et discontinues en temps, formulations stabilisées de type Galerkin moindres-carrés).

Summary

New technologies in computer science applied to numerical computations open the door to alternative approaches to mechanical problems using the finite element method. In classical approaches, theoretical developments often become cumbersome and the computer model which follows shows resemblance with the initial problem statement. The first step in the development consists usually in the analysis of the physics of the problem to simulate. The problem is generally described by a set of equations including partial differential equations. This first model is then replaced by successive equivalent or approximated models. The final result consists in a mathematical description of elemental matrices and algorithms describing the matrix form of the problem. The traditional approach consists then in constructing a computer model, generally complex and often quite different from the original mathematical description, thus making further corrections difficult. Therefore, the crucial problem of both the software architecture and the choice of the appropriate programming language is raised.

Partially breaking with this approach, we propose a new approach to develop and program finite element formulations. The approach is based on a hybrid symbolic/numerical approach on the one hand, and on a high level software tool, object-oriented programming (supported here by the languages Smalltalk and C++) on the other hand. The aim of this work is to develop an appropriate environment for the algebraic manipulations needed for a finite element formulation applied to an initial boundary value problem, and also to perform efficient numerical computations. The new environment should make it possible to manage all the concepts necessary to solve a physical problem: manipulation of partial differential equations, variational formulations, integration by parts, weak forms, finite element approximations,... The concepts manipulated therefore remain closely related to the original mathematical framework. The result of these symbolic manipulations is a set of elemental data (mass matrix, stiffness matrix, tangent stiffness matrix,...) to be introduced in a classical numerical code. The object-oriented paradigm is essential to the success of the implementation. In the context of the finite element codes, the object-oriented approach has already proved its capacity to represent and handle complex structures and phenomena. This is confirmed here with the symbolic environment for derivation of finite element formulations in which objects such as expression, integral and variational formulation appear. The link between both the numerical world and the symbolic world is based on an object-oriented concept for automatic programmation of matrix forms derived from the finite element method. As a result, a global environment in which the numerician is capable of evolving, using a language close to the natural mathematical one, is achieved. The potential of the approach is further demonstrated, on the one hand, by the wide range of problems solved in linear mechanics (elastodynamics in 1 and 2D, heat diffusion,...) as well as in nonlinear mechanics (advection dominated 1D problem, Navier Stokes problem), and, on the other hand by the diversity of the formulations manipulated (Galerkin formulations, space-time Galerkin formulations continuous in space and discontinuous in time, generalized Galerkin least-squares formulations).

Contents

ACKNOWLEDGEMENTS **5**

RÉSUMÉ **7**

SUMMARY **9**

CONTENTS **11**

CHAPTER 1 INTRODUCTION **15**

1.1 HIGH LEVEL SOFTWARE TOOLS FOR FINITE ELEMENT ANALYSIS 15
1.2 OVERVIEW OF THE USE OF ALGEBRAIC COMPUTATION TOOLS FOR FINITE ELEMENTS 16
1.2.1 Semi-analytical/numerical approaches *16*
1.2.2 Enhancing finite element code performance *17*
1.2.3 Speeding up finite element code development *17*
1.3 OBJECT-ORIENTED FINITE ELEMENT PROGRAMMING 19
1.4 AN OBJECT-ORIENTED HYBRID SYMBOLIC-NUMERICAL APPROACH FOR FINITE ELEMENT ANALYSIS 20
1.5 THE THESIS 21
1.5.1 Overview *21*
1.5.2 How to read the thesis *21*

CHAPTER 2 AN OBJECT-ORIENTED ENVIRONMENT FOR SYMBOLIC DERIVATIONS OF FINITE ELEMENT FORMULATIONS **23**

2.1 GENERAL APPROACH 23
2.2 SYMBOLIC DERIVATION OF MATRIX FORMS FOR A LINEAR INITIAL BOUNDARY VALUE PROBLEM 23
2.2.1 Derivation of the weak and matrix forms for elastodynamics *23*
2.2.2 The main objects of the symbolic environment for elastodynamics *27*
2.2.2.1 Derivation of the weak form 27
2.2.2.2 Derivation of the discretized weak form 30
2.3 THE CLASS HIERARCHY AND THE CLASSES FOR AN INITIAL BOUNDARY VALUE PROBLEM 32
2.3.1 The class hierarchy *32*
2.3.2 Review of the classes *33*
2.3.2.1 The collections 33
2.3.2.1.a The dictionaries of **FEMTheory** 33
2.3.2.1.b Subclasses of FEMTheoryOrderedCollection 35
2.3.2.2 Subclasses of FEMTheory 35
2.3.2.2.a Subclasses of FEMTheoryMathematicalStructures 35
2.3.2.2.b Accessory classes 35
2.3.3 Data transfer in a hierarchical tree of objects dedicated to finite elements *36*
2.4 THE GRAPHICAL ENVIRONMENT FOR THE DERIVATION OF VARIATIONAL STATEMENTS 38
2.4.1 Simple object-oriented concepts for the graphical interface *38*
2.4.2 The interface *39*
2.4.3 Description of the available tools to derive finite elements *41*
2.4.3.1 Tools applicable to variational forms 41
2.4.3.2 Tools applicable to discrete forms 42
2.4.4 About the interface *43*
2.5 A FIRST EXAMPLE OF DERIVATION: A ONE-DIMENSIONAL ELASTIC BAR 43
2.5.1 The problem *43*
2.5.2 Derivation of the matrix form *44*

CHAPTER 3 CONCEPTS FOR AUTOMATIC PROGRAMMING OF FINITE ELEMENTS **49**

3.1 AUTOMATIC GENERATION OF A FINITE ELEMENT CODE IN A SYMBOLIC OBJECT-ORIENTED ENVIRONMENT . 49
3.1.1 Principle of automatic generation of a code in a symbolic object-oriented mathematical environment *49*
3.1.2 Programming in Smalltalk *50*
3.1.3 Finite element automatic programming in the FEM_Theory environment *51*
3.1.4 An example of automatic implementation : a bar in elastodynamics. *60*
3.1.4.1 A new element generated in FEM_Object. 60

3.1.4.2 Test of the new element 61

3.2 CLASSICAL FORMULATIONS IN STRUCTURAL AND FLUID MECHANICS 62

3.2.1 Linear elasticity *62*

3.2.2 A mixed formulation for Stokes flow problem *62*

3.2.2.1 Mathematical formulation 62

3.2.2.2 Derivation in FEM_Theory : 63

3.2.2.3 Numerical results 67

CHAPTER 4 COMPUTER AIDED SOFTWARE ENGINEERING FOR FINITE ELEMENTS DEVELOPMENTS 69

4.1 DIMENSIONAL ANALYSIS IN AN OBJECT-ORIENTED ENVIRONMENT FOR FINITE ELEMENTS 69

4.1.1 The concepts for dimensional analysis *69*

4.1.2 The objects for dimensional analysis *70*

4.1.3 The classes *71*

4.1.4 Strategy for dimensional analysis in FEMTheory *74*

4.1.5 The graphical environment for dimensional analysis *76*

4.1.6 A simple illustration of dimensional analysis in FEM_Theory *77*

4.1.7 Dimension control in finite elements *79*

4.2 CHECKING INDEX WRITING CONSISTENCY 80

4.2.1 Goal *80*

4.2.2 Implementation of writing analysis *80*

4.2.3 Example of analysis *82*

CHAPTER 5 BEYOND A CLASSICAL GALERKIN FORMULATION 85

5.1 TOWARDS FINITE ELEMENT COMPUTATIONS OF INCOMPRESSIBLE FLOWS 85

5.2 GALERKIN LEAST-SQUARES TYPE STABILIZED METHODS 85

5.2.1 Brief review and objective *85*

5.2.2 Integration of stabilized formulations in FEMTheory *87*

5.2.3 A stabilized formulation for the Stokes flow problem *88*

5.2.3.1 Derivation of the stabilized formulation in FEMTheory 90

5.2.3.2 Numerical tests of the stabilized element for Stokes flow 91

5.3 SPACE-TIME FORMULATIONS 92

5.3.1 Discontinuous space-time formulations *92*

5.3.2 Integration of discontinuous space-time formulations concepts in FEMTheory *93*

5.3.2.1 The objects needed for discontinuous space-time formulations 93

5.3.2.1.a The discontinuous space-time formulation for a linear one-dimensional advective equation 93

5.3.2.1.b The objects for the discontinuous space-time formulation 94

5.3.2.2 Class JUMP_TERM 96

5.3.2.3 The graphical environment 97

5.3.3 The linear one-dimensional advection equation in FEM_Theory *98*

5.3.3.1 Derivation of the linear 1-D advection equation 98

5.3.3.2 Code generated automatically into FEM_Object 100

5.3.3.3 Numerical tests 101

CHAPTER 6 A NONLINEAR APPROACH IN FEM_THEORY: APPLICATION TO ADVECTION DOMINATED EQUATION MODELS 103

6.1 OBJECTIVE 103

6.2 A THEORETICAL APPROACH FOR LINEARIZATION 103

6.2.1 Definition of linearization *103*

6.2.2 Directional derivative *104*

6.3 OBJECT-ORIENTED CONCEPTS FOR SYMBOLIC DERIVATIONS OF NONLINEAR PROBLEMS 104

6.3.1 The objects for a consistent linearization scheme in FEM_Theory *104*

6.3.2 Implementation in FEM_Theory *105*

6.3.2.1 The linearization procedure 105

6.3.2.2 Class Increment 108

6.3.3 The graphical environment *109*

6.4 ENHANCING THE AUTOMATIC PROGRAMMING PROCEDURE 109

6.5 APPLICATION TO A NONLINEAR ONE-DIMENSIONAL ADVECTION EQUATION MODEL 110

6.5.1 Strong and weak form of the problem *110*

6.5.2 Derivation of the stabilized space-time formulation for 1-D nonlinear advection in FEM_Theory. *111*

6.5.3 Numerical code generated for nonlinear formulation *112*

6.5.4 Numerical results *114*

6.6 TOWARDS A GENERAL PURPOSE ENVIRONMENT FOR NONLINEAR PROBLEMS 115

CHAPTER 7 APPLICATION TO THE SOLUTION OF THE NAVIER-STOKES PROBLEM 117

7.1 ENHANCEMENT TO SOLVE NAVIER-STOKES EQUATION 117
7.2 A STABILIZED FORMULATION FOR THE STEADY STATE NAVIER-STOKES PROBLEM 118
7.3 A STABILIZED SPACE-TIME FORMULATION DISCONTINUOUS IN TIME FOR NAVIER-STOKES PROBLEM : APPLICATION TO THE DAM BREAK PROBLEM 123
7.3.1 A space-time formulation for the Navier-Stokes problem with moving boundaries 123
7.3.2 Application to the dam break problem 125
7.3.3 Closer to the mechanics 128

CHAPTER 8 CONCLUSION 129

8.1 A BRIEF OVERVIEW 129
8.2 ANALYSIS OF THE GENERICITY 129
8.3 TOWARDS A GENERAL PURPOSE ENVIRONMENT FOR FINITE ELEMENTS DEVELOPMENTS 131

REFERENCES 133

APPENDIX A 141

APPENDIX B - ANALYSIS OF A ONE-DIMENSIONAL DIFFUSION EQUATION 157

GALERKIN FORMULATION 157
GALERKIN LEAST-SQUARES FORMULATION 160
GALERKIN / GRADIENT LEAST SQUARES FORMULATION 161
NUMERICAL RESULTS FOR THE ONE-DIMENSIONAL DIFFUSION EQUATION 163

APPENDIX C - LINEAR ELASTODYNAMICS 165

DERIVATION OF THE FORMULATION AND AUTOMATIC PROGRAMMING 165
TEST OF THE ELEMENT 166
Description of the problem 166
Numerical results 167

CURRICULUM VITAE 169

Chapter 1 Introduction

1.1 High level software tools for finite element analysis

New technologies in computer science applied to mechanical computations open today the door to alternative approaches to the solution of mechanical problems. The usual developments consist in performing a theoretical study of the given problem which normally leads to a tedious procedures done by hand and followed by a computer model implementation. This approach is illustrated in Figure 1 showing the example of the development of a Finite Element code to solve mechanical problems. The first phase of the development is the analysis of the physical problem to solve. The problem is generally described by a set of equations including partial differential equations. This first model is then replaced by successive equivalent and/or approximated models. The last step is the development of the last model which is here the numerical code.

Nowadays, these successive derivations can be made in an easier way, faster and more efficiently through the use of ***high level software tools***, as shown in [FRI 92]. These tools can be grouped into three main classes of ideas. The first one corresponds to the last generation of high level programming tools which can be decomposed into two main classes : the classical procedural languages (FORTRAN 77 and 90, PASCAL, ...) and the object-oriented languages (C++, SMALLTALK, Java, ...). The second one includes algebraic software systems such as Maple, Mathematica, Matlab .... As shown in [FRI 92], it is worth having a third hybrid approach for general mechanical analysis, which means an approach based on mixed symbolic-numerical tools. The objective of this work is to develop such an environment based on high level programming languages, capable of both manipulating algebraic equations and performing efficient numerical computations. It must be widely open to all types of future extensions such as the application to alternative new finite element formulations or new numerical schemes.

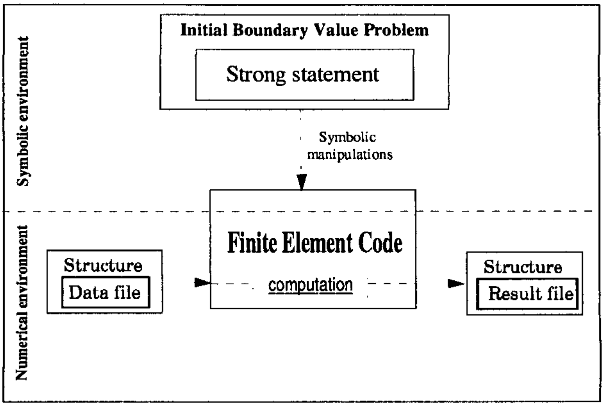


Figure 1 Overall scheme for generating Finite Element Code

1.2 Overview of the use of algebraic computation tools for finite elements

The use of algebraic manipulations software has always been a point of interest for finite elements development. The first related works date from the beginning of the development of the finite element method in the seventies. Among the first related works one finds [LUF 71], in which is described a methodology to automatically generate finite element matrices based on the characteristics of the new element; the approach is restricted so far to a finite number of problems: plane strain, bending, and shallow shells. Since then, a lot of people have used algebraic computation capabilities to assist finite element solution procedures. A few similar works organized in three main categories are related here. Firstly, some people use symbolic computer systems directly applied to finite element analysis, mixing both analytical and numerical approaches. The second class of approaches groups all the works in which the main objective is to improve the efficiency of numerical computations in classical finite element codes. Finally, many authors aimed at accelerating finite element code development using either existing tools or generating them.

1.2.1 Semi-analytical/numerical approaches

In this type of approach, a classical finite element approach is programmed in a symbolic software package. Some variables are kept as symbolic parameters and thus their influence on the computations can be evaluated. Two typical examples are given hereunder.

In [CHO 92] an application to 2D elasticity is developed within the symbolic algebra software Mathematica. The displacement fields for a 2-D body subject to linear temperature distribution is obtained in a semi-analytical form. Two tests are performed : a homogeneous elastic body with rectangular shape and a body containing a circular inhomogeneity. This method renders possible the automation of otherwise tedious code writing and can be useful for sensitivity analysis because all relevant parameters remain in symbolic form. In [IOA 93], the software tool Mathematica is used for the solution of two simple elasticity problems by the finite element method. The principle of the approach consists in keeping a parameter in the symbolic form of the finite element matrices and using Taylor series expansion for approximations. Thus, the objective is to try to optimize the parameter of the computation. It is applied in [IOA 93] to a square plane isotropic elastic medium under symmetric loads, divided in eight triangular elements. The problem is solved using a Gauss-Seidel method which makes it possible to study the influence of Poisson's ratio. The second example consists in the bending analysis of a rectangular isotropic elastic plate with simply supported edges and loaded with a uniformly distributed perpendicular load. Here the influence of the ratio of the dimensions is studied.

This semi-analytical/numerical environment should obviously provide a convenient framework for optimization of parameters, using finite element techniques for the computation. But at the current state of developments in software and hardware, this can only be applied to small problems. Moreover, extension to alternative linear and a fortiori nonlinear problems seems to be difficult.

1.2.2 Enhancing finite element code performance

Another current use of mathematical software tools consists in effectuating some preliminary computations in order to enhance the efficiency of the finite element code.
In [YAN 94], expressions for linear isotropic materials in statics in 2-D and 3-D, are evaluated algebraically, and integration of the stiffness matrix and external forces is performed. Thus the integration scheme is optimized before the code is written.
In [SIL 94], the analytical integration scheme is also optimized, using Maple, and the FORTRAN finite element code is directly generated using a Maple functionality. The code is then applied to solve finite element problems in magnetics. An approach with similar purposes is developed in [YAG 90]. Here REDUCE and Macsyma are employed to optimize a 2D 4-node isoparametric element for elastic analysis and to generate the corresponding code. In [BAR 89], an application of symbolic computing to the hierarchical FEM is shown; in this method the degree p of the approximating polynomial functions tends to infinity which addresses the problem of the accurate computation of the integrals for large values of p. The approach chosen in this paper is to evaluate them in a symbolic way using the package REDUCE. This is illustrated on 2D elasticity and the FORTRAN code is produced automatically, by means of a REDUCE function.
Two important features of using existing mathematical packages are the following. On the one hand, it is possible to use the power and the flexibility of these environments to optimize the expressions needed to evaluate finite element matrices. On the other hand the numerical code can be generated directly within the environment; the advantage is that the code which is generated automatically does not need any debugging.

1.2.3 Speeding up finite element code development

The derivation of finite element matrices generally involves tedious mathematical computations. The idea is to reduce the time spent on these manipulations through the use of a symbolic mathematical environment to determine the matrices of the finite element method and eventually introduce the final elemental forms automatically into an existing numerical code (written in FORTRAN for all the examples of this section). This leads to a systematic development of a finite element code for a given formulation. Some of the works presented below propose programs which generate directly the correct matrices. They are fed with various input parameters such as e.g. number of nodes, number of degree of freedom. Some other works use classical mathematical software to perform the derivations.
An illustration of the first approach is given in [GUN 71]. This paper presents the main features needed to develop finite element stiffness matrices with a computer. An illustration is made for the development of a third order triangular plate bending element. This makes it possible to test, at low cost, new elements for solving a given practical problem. [HOA 80] and [CEC 77] are based on the same approach. Applications in [HOA 80] are shown on a cylindrical shell and on the analysis of a curved beam element, whereas in [CEC 77] simple examples are shown but the method is applied to space-time elements. In [LUF 71], the use of algebraic software is suggested to manipulate polynomials and perform numerical integration for finite element development. This methodology includes the choice of parameters such as number of nodes, number of degrees of freedom per node for each variable, expansion of the polynomial for displacements, geometric and material properties. The user then keeps the main features coming from a finite model under his control in order to obtain correct matrix forms. Many authors have followed the same approach. In [BAR 92], the mathematical package REDUCE is used to automatically produce elemental mass and stiffness matrices

using Hermite polynomials, and then generate the corresponding FORTRAN code for a conventional finite element code. In the same way in [KOR 79], symbolic generation of a finite element stiffness matrix is achieved. Here, the authors have taken advantage of the user-friendly capabilities of MACSYMA: a library option gives access to a set of pre-defined matrices shapes for material properties in linear elasticity. In [NOO 81], the potential of using the symbolic manipulation in the development of nonlinear finite elements is shown. This is the only work that was found relating to the study of nonlinear problems. The development of nonlinear finite elements goes through three steps: generation of the algebraic expressions for the stiffness coefficients of nonlinear analysis, generation of the corresponding FORTRAN code for numerical evaluation of stiffness coefficients, checking the consistency of the FORTRAN code generated by comparing it to the FORTRAN statements for the arrays of coefficients given in the MACSYMA format. Two examples illustrate the approach. A displacement formulation for a 2D shear-flexible, doubly-curved deep shell element, and a mixed formulation for the same model with discontinuous stress-resultant fields at inter-element boundaries. In [CAM 97] the algebraic software Maple is used for multivariate polynomials computation for finite elements models. Polynomials and their derivatives are computed using Horner's method and efficient C and FORTRAN codes are produced. In [LEF 91], a system for the generation of global stiffness matrix is described. An input file for a specific problem is created for a system called SFEAS (Symbolic Finite Element Analysis System) which generates a file in the symbolic mathematical language REDUCE. The result is run and a FORTRAN code is produced and then integrated in the equation solving system. The code produced here is much more efficient than the NASTRAN one, but the preprocessing phase which includes running the REDUCE system is slow. In [WAN 86], a LISP-based system to derive formulas for the finite element method and to generate directly FORTRAN code is described. Efficient techniques for code generation are employed such as automatic labeling of expressions and exploitation of symmetries in expressions. It is the only reference in which the problem of automatic programming is clearly addressed. The package is written in LISP and runs with MACSYMA. The input can be given by the user interactively or introduced via a script file. The different entities needed for finite element formulations can be generated, for example B-matrix (see [HUG 87, p. 87]), jacobian matrix, stiffness matrix, etc... The accent is put on the optimization of code generation, aiming to get an efficient numerical code. The two last examples we give here are probably among the best developed systems. Many other similar applications can be found in [NOO 90] and [NOO 79] and the references therein.

The examples given in this section show the usefulness of high level tools in the development of finite elements. This analysis draws the main lines of the concepts needed for a general purpose environment dedicated to the finite element method. These approaches show the potential of symbolic software tools for enhancing FE techniques in a computerized environment; on the one hand, the domain of application is wide and, on the other hand, various solution schemes have been used. They show that in order to get a general purpose system for fast prototyping of finite elements, several ingredients are necessary: a natural and user-friendly description of the problem, an efficient symbolic computation tool, and finally an efficient link between the symbolic tool and the numerical finite element code. All these systems need a preliminary analysis usually performed manually before the development can be passed over to the computer algebra software and suffer from a lack of generalization capabilities. In fact, all these systems have their drawbacks. The first one is that all the systems still need a preliminary analysis performed manually, which can be rather tedious. The second one is the necessity to use multiple systems which a fortiori requires the developer's knowledge of each of them. For example in [SIL 94, YAG 90, BAR 89] derivations of finite element matrices are obtained using an algebraic system (Maple, REDUCE, Macsyma); the elemental forms are then introduced for a classical finite element code using a classical programming language (in all cases here FORTRAN). Consequently, the user has to do the symbolic computations in one environment and the numerical computations in another one, with the necessity for him to be able to evolve in two different programming environments and to learn both an algebraic software language and a classical programming language. The third one has to do with the computerized symbolic manipulations part. Each one of these systems has been developed to optimize specific features of the finite element approach; for example in [YAG 90], numerical integration is optimized, whereas in [CAM 97] it is the accuracy of the computation of polynomials that is optimized. It means that all these systems are specialized for some given tasks. As matter of fact, the extension of these tools or approaches to new finite element problems can become a tremendous task and can lead to impossibilities in complex nonlinear approaches.

The use of object-oriented techniques should make it possible to overcome these difficulties, while keeping the main advantages of symbolic approaches.

1.3 Object-oriented finite element programming

Many difficulties arise in the development of FE software. This represents the last step in the process of developing simulation tools. At the very beginning of the step lies a given physical problem. This problem is generally modelled by a set of partial differential equations. At this stage assumptions are made on the geometry, the kinematics, the loading, etc... A finite element strategy is then applied to the mathematical model. The result is in general a few pages describing the algorithms and the matrix form of the problem, expressed in a "simple" mathematical language. The traditional approaches lead to the elaboration of the corresponding computational tool, which usually is far from the original form of mathematical algorithms. The problem of both the architecture of the software and the language used in this development is a crucial point evoked e.g. in [BRE 92b, CHA 88]. It is necessary in some sense to get closer to the natural mathematical or mechanical language. Thus, the coupling between conventional procedural approaches (the most popular is FORTRAN) and the developing of high level data abstraction concepts with simple and natural programming rules lead to a new generation of FE codes (see e.g. [VER 88, BRE 92b]). A new approach for the FE code organization advocated in [COL 88, REH 89] corresponds to object-oriented programming. This approach naturally encompasses concepts for high level architecture and evolution towards more natural mathematical languages. For the first time in [REH 89 and MIL 88], object-oriented programming was proposed as a general methodology for Finite Element implementation. Both implementation examples use a LISP based system. One of the key points of the method to get better structured programs is the very high level data abstraction capabilities of the approach. In [REH 89], objects of matrix type appear and in [MIL 88] structural objects such as node, degree of freedom, and element are described. The latter is completed in [MIL 91] where object-oriented languages are discussed. In [FEN 90], the modularity and the reusability of object-oriented finite element codes are put in prominent position and the efficiency in the design and the implementation of FE is emphasized. The same conclusions are drawn in [FOR 90]. Here an interesting comparison is performed between a classical FE code (a C program) and an equivalent object-oriented version (a mixed C - Object Pascal program). The size of the OO code is smaller, probably due to the use of the inheritance. Similar remarks can be found in many papers: [FIL 91, ARO 91, LUC 92, DUB 91, BAU 92, DEV 92a,b , ROS 92a,b,c , MAC 92, SCH 92, NIE 94, ZEG 94, DRO 96, MAC 97]. In [ZIM 92, DUB 92, DUB 93], a complete OO environment for linear FE analysis is thoroughly discussed. A new concept is introduced here as a programming rule, "the non-anticipation rule". By never anticipating the state of the object when sending a message to it, the code becomes much more robust. The extension of the ideas to nonlinear analysis can be found in [DUB 95, DUB 97], with additional interesting concepts such as "unassembled matrix" which allows a more flexible implementation of solution schemes using alternative storage. A complementary approach to the one proposed in [DUB 93] is proposed in [MEN 93] for nonlinear constitutive laws, here J_2 plasticity. Accordingly, in [BES 97, FOE 96], an advanced description of the object "material" is given. The integration of complex constitutive laws in a C++ object-oriented FE code is made easier and more flexible by using C++ programming rules permitting dynamic binding and linking of code. Since then, the Object-Oriented paradigm has been used in many fields of computational mechanics: in parallel implementation of the FE code [ANG 92, BUF 97, HSI 97], in rapid dynamics [POT 97], in multi-domain analysis for metal cutting, mould filling, in composite material forming [WAL 96, GEL 95], in fracture mechanics [KAW 95]. This list is of course non exhaustive but shows that these ideas are now widely spread in the computational mechanics community.

In most of these works it has been shown that the implementation resembles more closely the mathematical developments. Roughly speaking, the algorithms are easier to describe and the definition of basic mathematical entities is natural. The object-oriented paradigm has been shown to be most appropriate to easily describe complex phenomena. But this description is usually limited to the elemental forms and their management whithin complex solution algorithms.

1.4 An object-oriented hybrid symbolic-numerical approach for finite element analysis

Taking into account the features developed in the works on symbolic derivations reported in section 1.2, the idea is now to develop a system dedicated to fast prototyping of finite element formulations, including nonlinear ones.

The need to deal with a large range of problems leads to the creation of an environment capable of managing all the concepts needed to solve physical problems, such as differential equations, variational formulations, integration by parts, weak forms, finite element approximations,... where traditionally manual derivations are replaced by a computer tool. A second important feature is the necessity to keep a traditional numerical code. This is justified for the following reason: complex geometric domains are necessary to test finite element formulations; somehow, tests have to be done on real life problems. The natural integration of both a numerical finite element environment and features for symbolic manipulations (see Figure 1) is achieved through the object-oriented paradigm. In the category of high level languages, object-oriented programming is today getting more and more attention in computational mechanics as shown in section 1.3. In the particular context of finite element software development, this type of approach leads to better structured codes for which maintenance and extendibility are facilitated. These are the capabilities of the approach to represent complex systems which lead us to select it. In a sense, this work can be seen as an extension of previous ideas developed for object-oriented concepts applied to finite elements (see [ZIM 92, DUB 92, DUB 93a]), to the symbolic derivation of the finite elements formulations; it is a new way of programming finite elements. The link between the numerical world and the symbolic world leads to the development of object-oriented concepts for the automatic programming of symbolic elemental matrix forms derived from finite element formulations. The result is a global environment in which the numerician is able to move naturally, always using a language close to his natural one. As a final test of the work presented in this thesis, the proposed environment will have to prove its capabilities in the evaluation of various formulations of the Navier-Stokes equations in the context of a project about debris flows described in [FRE 97].

1.5 The thesis

1.5.1 Overview

In chapter 2, the general approach which leads to the creation of FEM_Theory is presented in the example of elastodynamics. The main objects are then deduced and the hierarchical model is thoroughly discussed. In chapter 3, the object-oriented concepts for automatic programming of finite elements are presented. In chapter 4, an attempt to development assistance tools is made: a simple scheme for dimensional analysis and writing consistency checking is developed. In chapter 5, the approach is extended to stabilized and space-time formulations. A brief review of these methods mainly applied to computational fluid dynamics is given; illustration is made on a 1D advection equation. A nonlinear approach is developed in chapter 6 and is illustrated on a 1D nonlinear pure advection equation; a multi-dimensional example is treated in chapter 7 on an incompressible fluid driven by the Navier-Stokes equations. Conclusions and perspectives for this type of work are drawn in chapter 8.

1.5.2 How to read the thesis

This thesis can be read in two different ways. The reader who wants to have a global overview of the environment can read only chapters 1, 2 and 8, and sections 3.1 and 3.3. He can then have a look at the various examples of formulations given in chapters 5, 6 and 7, and in appendices A, B and C. The reader interested in implementing related ideas will find all the details in chapters 2, 3 and appendix A.

Chapter 2 An object-oriented environment for symbolic derivations of finite element formulations

2.1 General approach

The overall approach for a classical finite element approach is illustrated in Figure 2 by the equations of elastodynamics. Starting from the strong form of the problem statement an equivalent weak form is derived as shown in step 1 ; the matrix form results, from the weak form, by discretization in step 2. Finally, at step 3, the generation of the code depends essentially on the language selected for the code to be generated ; this aspect will be discussed in the next chapter.

As discussed in the introduction, all three steps can be done in a single symbolic object-oriented environment called FEM_Theory. The generated code is implemented in an existing object-oriented finite element code (see [DUB 92] and [DUB 93]). The general features of the environment are described in this chapter. In section 2.2, the main objects needed for the derivations are presented. The hierarchy of classes built in this study, on the one hand, and, the interaction between objects, on the other hand, are presented in section 2.3 with a brief description of the classes. The implementation is done in the Smalltalk environment but could be done in any other object-oriented language. In section 2.4, the concepts of the object-oriented graphical environment of FEM_Theory are described. Finally, an example of derivation is conducted step by step in section 2.5, on the example of a 1D bar in elastodynamics.

2.2 Symbolic derivation of matrix forms for a linear initial boundary value problem

2.2.1 Derivation of the weak and matrix forms for elastodynamics

The proposed approach is best demonstrated with an example; the case of linear elastodynamics is considered here. The problem statement is given on Table 1, as a set of differential equations. The derivation of matrix forms for the implementation into a finite element program requires first a weak form , equivalent to the strong form ; a complete derivation is given on Table 2 for easy reference. This step requires various manipulations such as integration by parts, substitution of expressions using other relations. The selection of an appropriate approximation space leads to a discretized weak form and finally to the derivation of matrix forms written from elemental contributions; it is given on Table 3 and Table 4. A complete treatment can be found in [HUG 87] e.g..

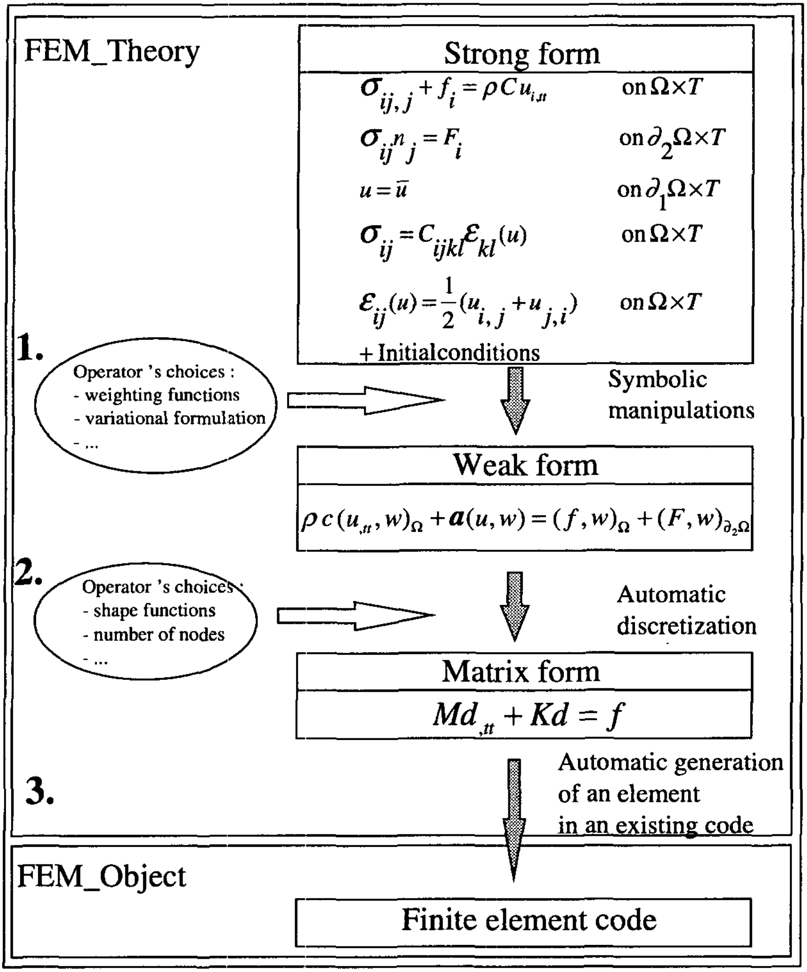

FEM_Theory
Strong form
$\sigma_{ij,j} + f_i = \rho C u_{i,tt}$ on $\Omega \times T$
$\sigma_{ij} n_j = F_i$ on $\partial_2 \Omega \times T$
$u = \bar{u}$ on $\partial_1 \Omega \times T$
$\sigma_{ij} = C_{ijkl} \varepsilon_{kl}(u)$ on $\Omega \times T$
$\varepsilon_{ij}(u) = \frac{1}{2}(u_{i,j} + u_{j,i})$ on $\Omega \times T$
+ Initialconditions
1.
Operator 's choices :
- weighting functions
- variational formulation
- ...
Symbolic manipulations
Weak form
$\rho c (u_{,tt}, w)_\Omega + \boldsymbol{a}(u,w) = (f,w)_\Omega + (F,w)_{\partial_2 \Omega}$
2.
Operator 's choices :
- shape functions
- number of nodes
- ...
Automatic discretization
Matrix form
$Md_{,tt} + Kd = f$
3.
Automatic generation of an element in an existing code
FEM_Object
Finite element code

Figure 2 General variational approach for finite elements

Table 1 Strong statement of linear elastodynamics Initial-Boundary-Value-Problem

Find u with appropriate continuity requirements, such that :

Ω in $R^{n_{sd}}$

n_{sd}: number of the space dimension

The equation of motion :

$$\sigma_{ij,j} + f_i = \rho u_{i,tt} \qquad \text{on } \Omega \times T \tag{1}$$

The boundary conditions :

$$\sigma_{ij} n_j = F_i \qquad \text{on } \partial_2\Omega \times T \tag{2}$$

$$u = \overline{u} \qquad \text{on } \partial_1\Omega \times T \tag{3}$$

with $\partial\Omega = \partial_1\Omega \cup \partial_2\Omega$

The constitutive equation :

$$\sigma_{ij} = C_{ijkl}\varepsilon_{kl} \qquad \text{on } \Omega \times T \tag{4}$$

The initial conditions :

$$u_{,t}(t=0,x) = (u_{,t})_0 \qquad \text{on } \Omega \tag{5}$$

$$u(t=0,x) = u_0 \qquad \text{on } \Omega \tag{6}$$

With the kinematic law :

$$\varepsilon_{kl} = \frac{1}{2}(u_{k,l} + u_{l,k}) \tag{7}$$

Table 2 Derivation of the weak statement of linear elastodynamics

Let $\mathcal{W}$ be the set of the zero kinematically admissible displacement fields for which W is in $\mathcal{W}$, W is regular and satisfies $w = 0$ on the boundary $\partial_1\Omega$.

Let $\mathcal{S}$ be a solution to the problem defined on Table 1, then for every W in $\mathcal{W}$ one can write the variational principle

$$\int_\Omega (\sigma_{ij,j} + f_i)\, w_i\, dv = \int_\Omega \rho\, u_{i,tt}\, w_i\, dv \tag{1}$$

Expanding the left-hand-side :

$$\int_\Omega \sigma_{ij,j} w_i\, dv + \int_\Omega f_i w_i\, dv = \int_\Omega \rho\, u_{i,tt} w_i\, dv \tag{2}$$

Integrating the first integral by parts :

$$\int_{\partial\Omega} \sigma_{ij} n_j w_i\, dv - \int_\Omega \sigma_{ij} w_{i,j}\, dv + \int_\Omega f_i w_i\, dv = \int_\Omega \rho u_{i,tt} w_i\, dv \tag{3}$$

Introducing into (2) and using the boundary condition (see Table 1) , equation (3) becomes :

$$\int_{\partial_2\Omega} F_i w_i\, dS - \int_\Omega \sigma_{ij} w_{i,j}\, dv + \int_\Omega f_i w_i\, dv = \int_\Omega \rho u_{i,tt} w_i\, dv \tag{4}$$

Assuming σ_{ij} symmetric, $\sigma_{ij} w_{i,j} = \sigma_{ij}\varepsilon_{ij}$; introducing the constitutive law (4) becomes :

$$\int_{\partial_2\Omega} F_i w_i\, dS - \int_\Omega C_{ijkl}\,\varepsilon_{kl}(u)\varepsilon_{ij}(w)dv + \int_\Omega f_i w_i\, dv = \int_\Omega \rho u_{i,tt} w_i\, dv \tag{5}$$

And, after rearrangement the final weak formulation results :

$$\int_\Omega \rho u_{i,tt} w_i\, dv + \int_\Omega C_{ijkl}\,\varepsilon_{kl}(u)\varepsilon_{ij}(w)dv = \int_{\partial_2\Omega} F_i w_i\, dS + \int_\Omega f_i w_i\, dv \tag{6}$$

Finally, after an obvious change in notations :

$$(\rho\, u_{,tt}, w)_\Omega + a(u,w) = (f,w)_\Omega + (F,w)_{\partial_2\Omega} \tag{7}$$

Table 3 Derivation of the matrix form

Introducing the approximations into the weak formulation :

$$\sum_{\substack{i=1\\j=1}}^{n_{ed}} \left(\rho u^h_{i,tt}, w^h_j\right)_\Omega + \sum_{\substack{i=1\\j=1}}^{n_{ed}} a\left(u^h_i, w^h_j\right) = \sum_{\substack{i=1\\j=1}}^{n_{ed}} \left(f_i, w^h_j\right)_\Omega + \sum_{\substack{i=1\\j=1}}^{n_{ed}} \left(F_i, w^h_j\right)_{\partial_2\Omega} \qquad (1)$$

After expansion of the equation and assuming that the operators are linear, one obtains :

with : $u^h = \sum_{i=1}^{n_{ed}} \sum_{A\in\eta/\eta_{\bar{u}}} N_A d_{Ai} e_i$ and $w^h = \sum_{j=1}^{n_{ed}} \sum_{A\in\eta/\eta_{\bar{u}}} N_A d^*_{Aj} e_j$ where η is the set of all nodes and $\eta / \eta_{\bar{u}}$: all nodes excluding the kinematically constrained nodes and e_i : the vector basis.

$$\begin{aligned}
&\sum_{A\in\eta/\eta_{\bar{u}}} \sum_{B\in\eta/\eta_{\bar{u}}} \sum_{\substack{i=1\\j=1}}^{n_{ed}} d_{Ai,tt} d^*_{Bj} \left(\rho N_A e_i, N_B e_j\right)_\Omega + \sum_{A\in\eta/\eta_{\bar{u}}} \sum_{B\in\eta/\eta_{\bar{u}}} \sum_{\substack{i=1\\j=1}}^{n_{ed}} d_{Ai} d^*_{Bj} a\left(N_A e_i, N_B e_j\right) \\
&= f_{nod} + \sum_{B\in\eta/\eta_{\bar{u}}} \sum_{\substack{i=1\\j=1}}^{n_{ed}} d^*_{Bi} \left(N_B e_i, f_j\right)_\Omega \\
&+ \sum_{B\in\eta/\eta_{\bar{u}}} \sum_{\substack{i=1\\j=1}}^{n_{ed}} d^*_{Bi} \left(N_B e_i, F_j\right)_{\partial_2\Omega} - \sum_{A\in\eta_{\bar{u}}} \sum_{B\in\eta_{\bar{u}}} \sum_{\substack{i=1\\j=1}}^{n_{ed}} \bar{u}^h_{Ai} d_{Bj} a\left(N_A e_i, N_B e_j\right)
\end{aligned} \qquad (2)$$

Notice that isolated loads are introduced here as nodal loads f_{nod} into equation (2).

Invoking arbitrariness of d^* , and symmetry of M and K equation (3) results :

$$M d'' + K d = f \qquad (3)$$

where :

$$\left|\begin{array}{l} K = A_{e=1}^{n_{el}}\left(k^e\right) \\ M = A_{e=1}^{n_{el}}\left(m^e\right) \\ f = f_{nod} + A_{e=1}^{n_{el}}\left(f^e\right) \end{array}\right| \quad \begin{array}{l} d = A_{e=1}^{n_{el}}\left(d^e\right) \\ d^* = A_{e=1}^{n_{el}}\left(d^{*e}\right) \\ d^e \left| \begin{array}{l} \ldots \\ d_A \left| \begin{array}{l} d_{A1} \\ \ldots \\ d_{An_{ed}} \end{array}\right. \\ \ldots \end{array}\right. \end{array}$$

$$k^e = \left[k^e_{PQ}\right] \quad m^e = \left[m^e_{PQ}\right] \quad f^e = \left[f^e_P\right]$$

$$k^e_{AB} = \int_{\Omega^e} B^t_A D B_B \, dv \quad (4) \quad \text{and} \quad k^e_{PQ} = e^t_i k^e_{AB} e_j \quad (5)$$

$$m^e_{AB} = \int_{\Omega^e} \rho N^t_A N_B \, dv \quad (6) \quad \text{and} \quad m^e_{PQ} = e^t_i m^e_{AB} e_j \quad (7)$$

$$f^e_P = f^e_{Pnod} + \int_{\Omega^e} N_A f_i \, dv + \int_{\partial\Omega^e} N_A F_i \, dS - \sum_{q=1}^{n_{el}} k^e_{pq} \bar{u}^e_q \quad (8)$$

with: $1 \le P,Q \le n_{ee} = n_{ed} n_{en}$
and: $P = n_{ed}(A-1) + i$
$Q = n_{ed}(B-1) + j$

where : A is the assembly operator
n_{ee} is number of element equations
n_{ed} is number of element degree of freedom per node
n_{en} is number of element nodes

Table 4 Elemental contributions

$$k^e = \int_{\Omega^e} B^t D B dv \qquad \text{with :} \quad B = \lfloor B_1 \cdots B_{n_{en}} \rfloor$$

$$m^e = \int_{\Omega^e} \rho N^t N dv \qquad N = \lfloor N_1 \cdots N_{n_{en}} \rfloor$$

D results from the compaction of C_{ijkl}, and N_a and B_a are the nodal contributions of the shape functions and global derivatives of shape functions.

2.2.2 The main objects of the symbolic environment for elastodynamics

The goal consists in building an environment capable of symbolic manipulations and computations for a semi-discrete approach to the linear elastodynamics problem. The classes which are needed will be identified following the steps of the derivation of the weak and matrix forms. The index notation is used throughout these derivations.

2.2.2.1 Derivation of the weak form

From the strong form it appears that a first class of objects is needed to represent terms like σ_{ij}, F_i, u_i, call it class **Term**. The index notation is used here for the reason that the analysis of a string to represent a given term can be simplified using the following convention: an uppercase represents the name of the field, a lowercase represents the indices. Typical members of class **Term** are illustrated in Figure 3 with the specification of their attributes: field name, indices, derivation indices and time derivation indices. All the attributes are strings. Instances of **Term** should be capable of performing tasks like derivation, substitution and adding themselves to other instances of **Term**.

$\sigma_{ij,j}$

Description attributes :

- name : σ
- indices : ij
- derivation indices : j
- time derivation indices : none

$u_{i,tt}$

Description attributes :

- name : u
- indices : i
- derivation indices : none
- time derivation indices : tt

Figure 3 Description of typical members of the class **Term**

An instance of **Term** is capable of analyzing itself, i.e. to define the nature (scalar or vector, unknown or known, virtual or solution) of its own field ; capable also of identifying which operator is applied to it. The tasks assigned to an instance of **Term** are some manipulations : substitution, addition and derivation. An instance of **Term** is built in fact from elementary strings, and all operations performed on it are therefore nothing more than alphanumeric list manipulations. The capacities of an instance of **Term** of analyzing itself indicate however an elementary mathematical structure which could be extended. At this stage, a special type of term is needed to represent special operators, such as operator ε_{ij} the symmetric part of the gradient of vector field. The corresponding class is called **Operator**. The structure and the behavior of this class are the same as the one of class **Term**. But one attribute is added: the function on which the operator is applied.

With instances of **Term**, instances of **Expression** can be built, which are algebraic sums of products of instances of **Term** or **Expression** ; an instance of class **Expression** is illustrated in Figure 4.

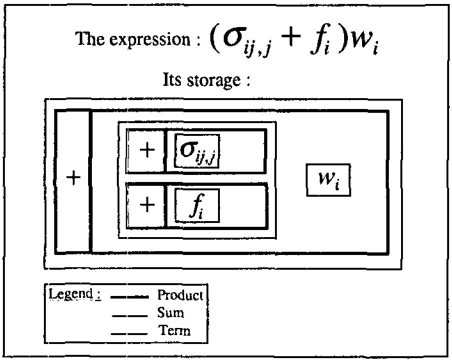


Figure 4 Structure of an instance of **Expression**

Again some basic operations can be performed by the instances of **Expression** which are essentially : expansion, substitution. These methods are discussed next.

Instances of **Expression** can be viewed as lists of lists, representing sums of products. All tasks consist in manipulating these lists. Expansion consists in reorganizing the expression in order to obtain simpler expressions, i.e. products containing only instances of class **Term** (no instances of class **Expression** as Figure 4 shows). The underlying mathematical operation is the application of the distributivity, e.g. $((a+b)(c+d)) = (ac+bc+ad+bd)$. The corresponding methods are : *addDist:* and *multDist:*

Substitution consists in removing some instances of **Term** in products and adding an instance of **Expression** to replace it.

The **Expression** class has shown the necessity to manipulate algebraic sums and products as shown in Figure 4. Two classes, **SumList** for sums and, **ProdList** for products model this behavior.

At this point the integrand of the variational form can be stated and manipulated; a class **Integral** is now needed. This class can be described as shown in Figure 5.

Class **Integral**

Example : $\int_{\Omega} \sigma_{ij,j} w_i dv$

main attributes:

- the integrand : $\sigma_{ij,j} w_i$

- the domain : Ω

main tasks:

- integration by parts, i.e. : $\int_{\Omega} \sigma_{ij,j} w_i dv \rightarrow \int_{\partial\Omega} \sigma_{ij} w_i n_j dv - \int_{\Omega} \sigma_{ij} w_{i,j} dv$

Figure 5 Typical instance of class **Integral**

Integration by parts generates a sum of integrals and the new integrands are generated by the first one, which is an instance of class expression. This shows that the expression must be capable of identifying which term carries the space derivative, and of activating the proper substitution.

The notion of functional is introduced next. Class **Functional** incorporates algebraic sums of instances of **Integral**. Here classes **SumList** and **ProdList** are also used, as shown in Figure 6.

Class **Functional**

Example : $\int_{\Omega} (\sigma_{ij,j} + f_i - \rho u_{i,tt}) w_i dv$

main attributes :
- composing integrals (sum of products of integral) **sumList**
main tasks :

- expansion : $\int_{\Omega} (f + g) dv = \int_{\Omega} f dv + \int_{\Omega} g dv$

- integration by parts of selected terms :

$$\int_{\Omega} (\sigma_{ij,j} + f_i - \rho u_{i,tt}) w_i dv \rightarrow \int_{\partial\Omega} \sigma_{ij} w_i n_j dv - \int_{\Omega} \sigma_{ij} w_{i,j} dv + \int_{\Omega} f_i w_i dv - \int_{\Omega} \rho u_{i,tt} w_i dv$$

Figure 6 Typical instance of class **Functional**

The variational principle is an instance of class **IntEquation**, expressing the equality of two functionals, one on the left-hand-side, one on the right-hand-side. It is described in Figure 6.

Class **IntEquation**

Example : $\int_\Omega (\sigma_{ij,j} + f_i) w_i dv = \int_\Omega \rho u_{i,tt} w_i dv$

main attributes :
- left-hand side **lhs**
- right-hand side **rhs**

main tasks :
- manipulations on lhs and rhs

Figure 7 Typical instance of class **IntEquation**

An accessory class not directly associated with our reference problem can be added here, the class **System**. This object is the representation of a set of equations (see Figure 8).

Class **System**

Example : $\begin{cases} \int_\Omega (\sigma_{ij,j} + f_i) w_i dv = \int_\Omega \rho u_{i,tt} w_i dv \\ \int_\Omega u_{i,i} q dv = 0 \end{cases}$

main attributes :
- a set of equations **equationsCollection**

main tasks :
- manipulations performed on its equation

Figure 8 Typical instance of class **System**

Notice that this class has a general construction. The collection of equations can contain all types of objects, not only **IntEquation** objects type.

At this stage all the key classes needed to generate weak forms are available and one can proceed with the discretization.

2.2.2.2 Derivation of the discretized weak form

The operations associated with discretization are described on Table 3 and Table 4. The first one consists in the selection of proper local approximation functions for u and w , e.g. :

$$u^h = \sum_{a=1}^{n_{en}} N_a d_a$$

Replacing u and w, in the weak form, by u^h and w^h and associated derivatives consists in instantiating **DiscretizedExpression**. Each instance of **Term** instantiates an expression which corresponds to its discretization. The smallest entity of the result is the representation of elemental contributions, an instance of **DiscretizationMatrix**. Then the result is arranged in order to yield an instance of **DiscretizedEquation**. As the class **DiscretizationMatrix** is the equivalent of the class **Term** for the discrete form, classes **DiscretizedExpression** and **DiscretizedEquation** are the equivalent of **Expression** and **IntEquation**. Indeed, discretizing an equation requires discretization of the left and right-hand sides, which are functionals composed of integrals. Consider the discretization of the following integral :

$$a(u,w) = \int C_{ijkl}\, u_{(i,j)} w_{(k,l)} dv$$

where u is a kinematically admissible field and w a zero kinematically admissible field and $u_{(i,j)} = \varepsilon_{ij}(u) = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right)$ is the symmetric part of the operator gradient applied to a vector field.

The discretization of this integral is sketched in Figure 9; this requires new classes for the discretized expressions which are **DiscretizedExpression** and **DiscretizedEquation**. The same algebraic sums and products, i.e. classes **SumList** and **ProdList**, are used here. The discretization procedure is decomposed into 3 steps and shows the need for new structures. The integral (as example here $\int_\Omega C_{ijkl} u_{(k,l)} w_{(i,j)} dv$) asks its integrand to give its discrete form. The terms are included into intermediate structures, instances of subclasses of **FEMTheoryDiscretizationStructure** (one subclass for unknowns and one for given data). This object asks its terms to determine the differential operator applied to its own field, which operator answers the corresponding discrete form; the different discrete forms are then combined to give the final answer $d^t K d^*$. The discretization of a term contained in an intermediate structure is described in Figure 10. The procedure is here completed by the introduction of a new intermediate structure in which the elementary discrete form corresponding to the differential operator applied to the field is embedded, subclasses of **FEMTheorySpatialDifferentialOperators** (new differential operators are introduced here). The main objects needed for the derivation of a finite elements model for classical elastodynamics have been introduced here. It is worth noting the generality of the proposed approach. First, new types of operators can be introduced easily into the objects. Second, the different steps needed for the approximation of the weak form and discretization of the domain have been clearly separated in the study of section 2.2.1.

Remark: the class **System** defined in the previous section can also be used to represent collections of discrete equations.

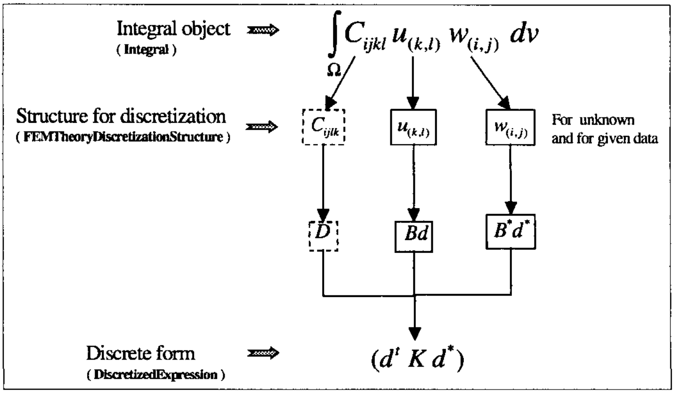

Integral object
(Integral)
$\int_\Omega C_{ijkl}\, u_{(k,l)}\, w_{(i,j)}\, dv$
Structure for discretization
(FEMTheoryDiscretizationStructure)
C_{ijlk}
$u_{(k,l)}$
$w_{(i,j)}$
For unknown
and for given data
D
Bd
B^*d^*
Discrete form
(DiscretizedExpression)
$(d^t K d^*)$

Figure 9 Discretization of an integral

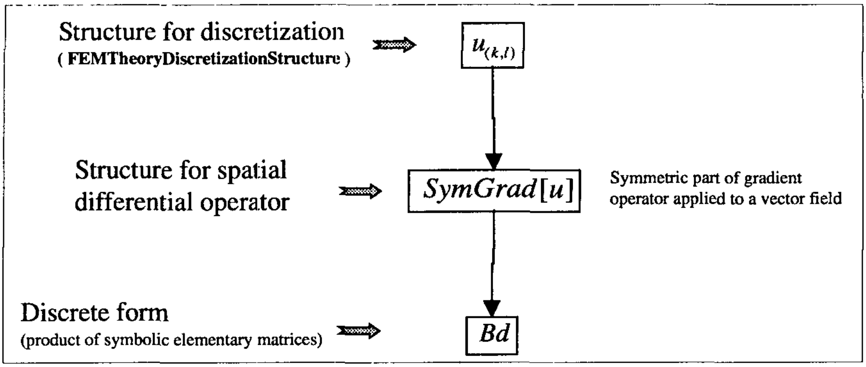

Structure for discretization
(FEMTheoryDiscretizationStructure)
$u_{(k,l)}$
Structure for spatial differential operator
$SymGrad[u]$
Symmetric part of gradient operator applied to a vector field
Discrete form
(product of symbolic elementary matrices)
Bd

Figure 10 Discretization of a term included in an intermediate structure

2.3 The class hierarchy and the classes for an initial boundary value problem

At this stage, the main objects needed for the derivation of a finite element problem by a variational approach have been detected. The intuitive approach chosen here to identify the objects needed for the proposed problem leads to a given number of structures for which the main behaviors are determined. The following step is the analysis the corresponding classes in order to isolate common attributes and methods. Then, they can be included into the hierarchy of the Smalltalk environment. The aim of this section is not to give the details of the implementation of the classes, but only an overview of the environment. All details about the classes and their methods can be found in Appendix A.

2.3.1 The class hierarchy

The complete hierarchy corresponding to the symbolic environment is grouped under classes **FEMTheory** and **FEMTheoryOrderedCollection**, and illustrated in Figure 2. Observe that the set of classes for **FEMTheory** lies next to the classes of **FEMObject** (in bold italic characters in the hierarchy shown in Figure 2) -the object-oriented finite element code (see Chapter 3) without any barrier between symbolic and numerical environments. In the next section, the classes of the hierarchy are briefly discussed following the hierarchical order.

2.3.2 Review of the classes

This subsection may viewed as a quick reference to the hierarchy of classes.

2.3.2.1 The collections

All the classes implemented in this part of the hierarchy are a specialization of existing classes of the Smalltalk environment. In the following section, classes **Dictionary** and **OrderedCollection** are discussed (see [VIS 95a] and [VIS 95b]).

*2.3.2.1.a The dictionaries of **FEMTheory***

Class **Dictionary** implements the behavior of the object dictionary. A dictionary represents a collection of objects, in which elements are accessed by keywords. Two types of behavior specialization have been implemented.

In the symbolic environment dedicated to the finite element method, for some specific features, it is necessary to have a large number of possibilities that can be included into lists. For example, in order to offer the user a friendly environment, several solved problems, i.e. a set of partial differential equations, are available and the list can of course be enriched easily. An elegant way to implement it is to use the existing Smalltalk class **Dictionary**. This object is used to store solved problems. It is also sometimes necessary for a typical collection of objects to specialize the access to the objects or to classify them; this is the case for class **FEMTheoryShapeFunctionsDictionary** which represents a dictionary of shape functions. Class **FEMTheoryGeneralDictionaryOfUnits** regroups all combined units constructed from fundamental units; this dictionary is used for dimensional analysis, its special role is to recognize the units through their definition.

Remark 1: a tool of the Smalltalk environment stores the objects (instances of classes) on disk, and recovers them (see class **ObjectFiler** [VIS 95b]); this makes it possible to manage the persistency of dictionaries in a natural way, as well as the elements contained therein.

Another way to specialize class **Dictionary** is to use this class for special purpose objects. Class **Dimension** for example, implements composed units, which result from combinations of fundamental units. The problem of dimensional analysis is addressed in chapter 4.

Remark 2: Classes equivalent to class **Dictionary** exist in other object-oriented languages (e.g. C++) or can easily be built (see e.g. [DUB 93]).

```
Object
...
    Collection
        IndexedCollection
            OrderedCollection
                HashedCollection
                    Dictionary
                        FEMTheoryDictionaries
                            Dimension
                            FEMTheoryGeneralDictionaryOfUnits
                            FEMTheoryShapeFunctionsDictionary
                ...
                FEMTheoryOrderedCollection
                    EquationsCollection
                    ExpressionLists
                        Prodlist
                        SumList
                    FEMTheoryCollectionStructure
                        FEMTheoryDiscretizationStructure
                            GivenDatasDiscretizationStructure
                            UnknownDiscretizationStructure
                        FEMTheoryProductStructure
                        FEMTheorySumStructure
...
    FEMObject
...
    FEMTheory
        FEMTheoryMathematicalStructures
            StructureWithDimension
                Expression
                    DiscretizedExpression
                Functional
                Integral
                IntEquation
                    DiscretizedEquation
                        VariationalPrinciple
                        DiscretizedVariationalPrinciple
                Term
                    DiscretizationMatrix
                    Increment
                    Operator
                    SpecialTerms
                        JUMP_TERM
            System
        FEMTheorySpatialDifferentialOperators
            SCALAR
                DFGradientF
                FgradientDF
                Gradient
                Scalar
                SecondDerivative1D
            TENSOR
            VECTOR
                Div
                DivSymmGrad
                DUGradU
                Grad
                SymmGrad
                UgradDU
                Vector
        GeometryReference
            NpbDim1
                Q2
            NpbDim2
                Q4
        Unit
...
    String
```

N.B. : underlined classes names correspond to the native environment

Figure 11 A symbolic environment for the Finite Element Method

2.3.2.1.b Subclasses of FEMTheoryOrderedCollection

In the Smalltalk environment, ordered collections (class **OrderedCollection**) are collections of objects in which the elements are stored in a prescribed order, the one in which objects are included into the collection. This object is the basic structure for various objects of **FEMTheory** regrouped under generic class **FEMTheoryOrderedCollection**. Class **EquationsCollection** implements the behavior needed for collections of equations for systems (see at the end of section 2.2.2.1). Under class **ExpressionLists**, are classes **ProdList** and **SumList**. These classes represent the products and sums needed to model and manipulate expressions (for the continuum problem or the discrete one). Special collections are grouped under class **FEMTheoryCollectionStructure**: a) classes used for discretization processes, for unknowns and given data (class **FEMTheoryDiscretizationStructure**), b) special sums and products needed for automatic generation of the code (classes **FEMTheoryProductStructure** and **FEMTheorySumStructure**)

2.3.2.2 Subclasses of FEMTheory

2.3.2.2.a Subclasses of FEMTheoryMathematicalStructures

In class **FEMTheoryMathematicalStructures**, all the classes needed to represent the different steps of the derivation are grouped. All its subclasses inherit its attribute **hierarchicParent** which makes it possible to take advantage of the hierarchical structure of the manipulated objects, such as expressions, equations, ... This feature is discussed in section 2.3.3. Class **StructureWithDimension** inherits behavior corresponding to dimensional analysis by its subclasses (see chapter 5). Classes **Expression**, **Functional**, **Integral**, **IntEquation**, **VariationalPrinciple**, **Term**, **Operator** and **SpecialTerms** and subclasses permit to represent continuum problems in strong and variational forms. Classes **DiscretizedEquation**, **DiscretizedVariationalPrinciple** and **DiscretizationMatrix** make it possible to deal with discrete forms of the problem. Class **System** makes it possible to manage sets of equations.

2.3.2.2.b Accessory classes

Class **FEMTheorySpaceOperator** and subclasses implement the basic matrix structures corresponding to the spatial and time differential operators

Class **GeometryReference** and subclasses implement behavior for a reference element, i.e. data describing the kinematics of the element in a natural coordinate system. Elements are classified by dimensions. More complex kinematics for beam or shell elements can be added here.

Class **Unit** is a structure representing a unit, e.g. the *Newton N*. The characteristic of the unit is its dimension (object dimension is a dictionary, see Chapter 5 and section 2.3.2.1.a) , e.g. $N = kg.m.s^{-2}$; the unit has access to the dictionary of units.

2.3.3 Data transfer in a hierarchical tree of objects dedicated to finite elements

Structure

The problem addressed in this section is the storage and the transfer of data in complex objects. To give an illustration consider the object instance of **IntEquation** used to represent a variational form in Figure 13. The instance of **IntEquation** has a left hand side and a right hand side. Consider the left hand side. It is a functional which has a sum containing products. In these products, one can find an integral. The integrand is an instance of **Expression**, which is a sum of products of terms or other expressions. The problem is that the same data can be required at different levels of the tree in order to perform different tasks. It is necessary first to store a piece of data once, and second to be sure to have access to it when necessary. The natural structure makes it possible to pass messages down the roots of the tree. A link to the father, represented with dashed-dotted arrows in Figure 13, makes it possible to pass messages upwards. This is done in the hierarchy (Figure 11) by giving every mathematical structure an attribute called **hierarchicParent**. In fact, this attribute is given to class **FEMTheoryMathematicalStructures** for main mathematical objects, and to **ExpressionLists** for sum and product lists. Subclasses inherit the attribute and corresponding behavior from their respective superclasses. The illustration of the contents of the attribute **hierarchicParent** for the classes is given in Figure 12.

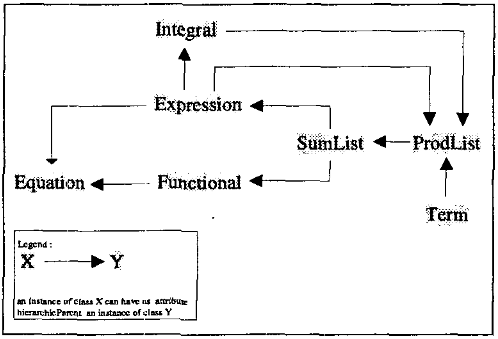

Integral
Expression
SumList
ProdList
Equation
Functional
Term
Legend :
X
Y
an instance of class X can have us attribute hierarchicParent an instance of class Y

Figure 12 Tree built with the attribute **hierarchicParent**

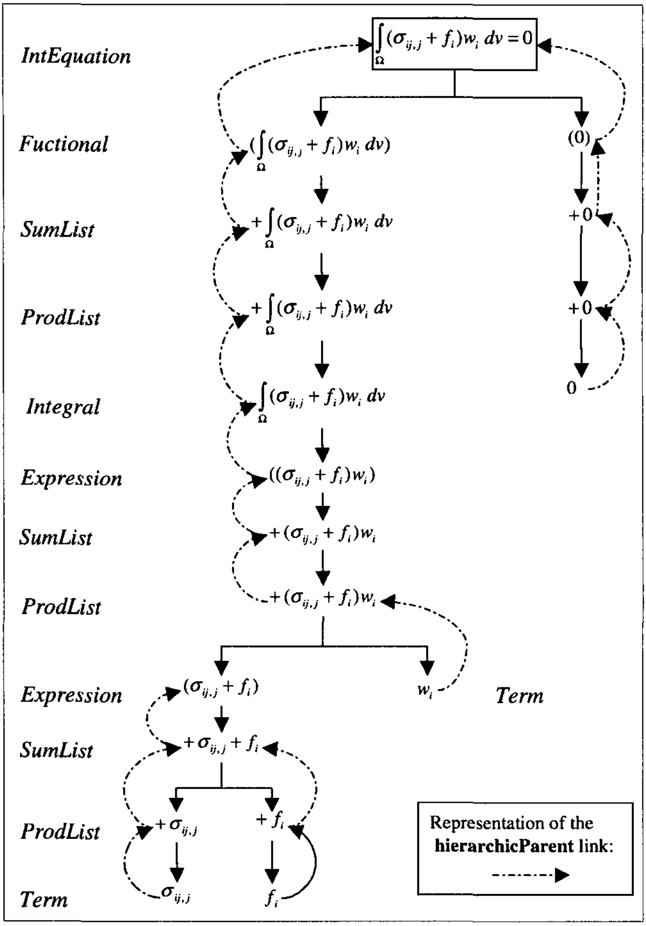

IntEquation
$\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv=0$
Fuctional
$(\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv)$
(0)
SumList
$+\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
$+0$
ProdList
$+\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
$+0$
Integral
$\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
0
Expression
$((\sigma_{ij,j}+f_i)w_i)$
SumList
$+(\sigma_{ij,j}+f_i)w_i$
ProdList
$+(\sigma_{ij,j}+f_i)w_i$
Expression
$(\sigma_{ij,j}+f_i)$
w_i
Term
SumList
$+\sigma_{ij,j}+f_i$
ProdList
$+\sigma_{ij,j}$
$+f_i$
Term
$\sigma_{ij,j}$
f_i
Representation of the hierarchicParent link:

Figure 13 Tree representation of an instance of **IntEquation**

Message passing procedure

Within the tree structure, messages can go down the roots in a natural way, the equation sends a message to its functional located in the left hand side, a message to be forwarded to sum and product, and so on to the target object which can perform the task or return an answer. With the proposed frame, messages can also propagate the opposite way. Consider the following simple example of the space dimension in the context of Figure 13. The object representing the variational form $\int_\Omega (\sigma_{ij,j} + f_i)w_i\, dv = 0$, which is an instance of **IntEquation**, knows the dimension of the space, attribute *spaceDimension* of class **IntEquation**. Suppose the term

$\sigma_{ij,j}$ needs this information (frequently needed) which is stored at the level of the object $\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv = 0$. The procedure by which the term gets the space dimension is programmed by means of two methods shown in Figure 14 and Figure 15. The method implemented in class **IntEquation** is natural and respects the non-anticipation rule adopted as a programming principle (see [DUB 92]). The object 'equation' supplies the number if it exists, otherwise it will try to get it by an alternative way (here by asking the user). The other objects of the structure (terms, sums, products, integrals, functionals) inherits the method Figure 15 from **FEMTheoryMathematicalStructures** or **ExpressionLists**. Each one of them successively requests the space dimension from its hierarchic parent until the equation which can answer the question.

```
giveSpaceDimension

spaceDimension isNil
        ifTrue:[ spaceDimension := self getSpaceDimension].
^spaceDimension
```

Figure 14 Method *giveSpaceDimension* of class **IntEquation**

```
giveSpaceDimension

    hierarchicParent notNil
     ifTrue:[^ hierarchicParent giveSpaceDimension ].
    ^nil
```

Figure 15 Method *giveSpaceDimension* of class **FEMTheoryMathmaticalStructure**

2.4 The graphical environment for the derivation of variational statements

At this stage, the mathematical environment has been described. Now, the derivation of a new problem consists in writing first a script file to describe the formulation and second to act on it. In order to simplify the management of the formulations, a simple user-friendly graphical interface has been created. The concepts described in section 2.4.1 have been established with easy and fast extendibility in mind which is an essential aspect of the global prototyping approach. A brief description on the use of the interface is given in sections 2.4.2 and 2.4.3.

2.4.1 Simple object-oriented concepts for the graphical interface

The main graphical window gives a view of the problem and is used to send a message to the object on which an operation needs to be performed.

View of the object

Every object is represented by a character string; strings can be obtained for every object by sending to it the message *printString*. Examples of views of different objects are given on Table 5. Remember that here the convention adopted is the following: upper cases represent names of fields and constants, and lower cases represent indices (the index notation is adopted throughout FEMTheory). Note that every object can be identified from its string, and can also be extracted from a global structure. For example, consider the string representing a variational formulation: «INT{(N,xW) // D}+INT{(RW) // D} = INT{DAU,ttW)». The object **Integral** represented by the string «INT{(N,xW) // D}» can be extracted from the equation and sent a specific message. In order to obtain the representation of this object, the message *printString* goes down the tree illustrated in Figure 13, and the resulting string is constructed by concatenating the contributions of each object.

Table 5 View of objects in FEMTheory

Class	Usual mathematical notation (example)	Corresponding representation in FEMTheory
Term	$\frac{\partial^2 U_i}{\partial t^2}$	Ui,tt
Expression	$N_{,x} - R$	(N,x-R)
Integral	$\int_D N_{,x} w dv$	INT{(N,xW) // D}
Functional	$\int_D N_{,x} wdv + \int_D Rwdv$	INT{(N,xW) // D}+INT{(RW) // D}
Equation	$\int_D N_{,x} wdv + \int_D Rw dv = \int_D DAu_{,tt} wdv$	INT{(N,xW) // D}+INT{(RW) // D} = INT{DAU,ttW)

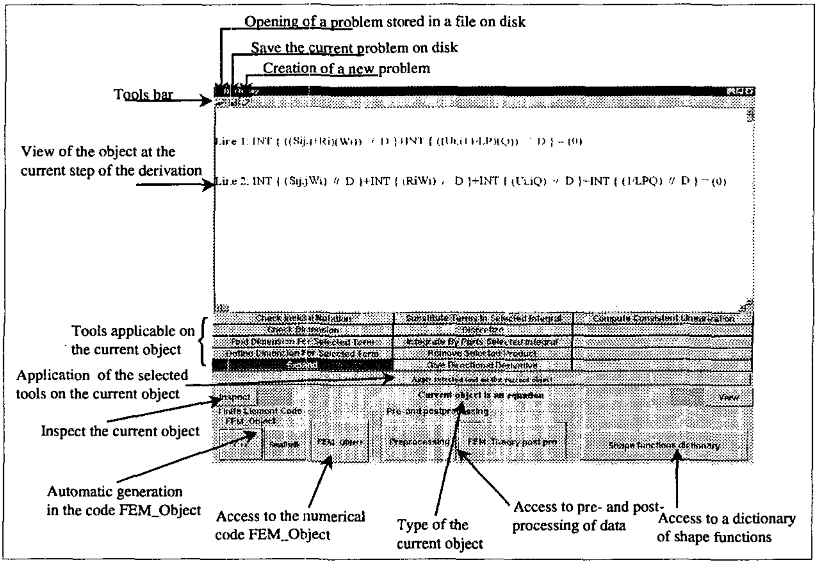
Derivation of a problem

During the derivation of a finite element formulation, the nature of the problem changes. First, an object **IntEquation** represents the successive variational forms; then an object **DiscretizedEquation** makes it possible to model the discrete forms, and finally an object **System** regroups the set of discrete equations. With the principles of representation given in the previous section, all derivations can be made by simple manipulations on these main objects, performed within a Windows graphical environment.

2.4.2 The interface

The manipulations of the current problem are made within a graphical window (Figure 2). This window is built by using a tool called WindowBuilder, which is completely integrated within the Smalltalk environment (for details about the environment see [WIN96]). As a matter of fact, the window object is an instance of a class called **FEMTheoryMainView**. The object representing the problem (either an instance of **IntEquation**, or **DiscretizedEquation**, or **System**) is an attribute of this window. A push-button system permits to send it messages. The different steps of the derivation appear in the text pane of the main window and are referred by line number; each step corresponds to the view (string) of the object manipulated. A set of push-buttons permits to select a tool and to apply it using the button *'Apply selected*

tool on the current object'. The set of tools available in the graphical environment, for the current object, is obtained by sending it the message *giveArrayOfTools*. The set of tools is then filled with the tools. Each tool corresponds to the selector of a method of the window which is automatically built and dynamically made available. The current object is then sent a message corresponding to the invoked action. Take the example in Figure 16 line 1. The object is an equation (instance of **IntEquation**). Consider the selected tool called *'Expand'*. Pushing the button *'Apply selected tool on the current object'* sends the message *applySelectedTool* of the window object. The corresponding method is presented in Figure 17. The string 'applyExpand' is built, and the code 'self applyExpand' is dynamically executed (the keyword 'self' corresponds here to the window instance of **FEMTheoryMainView**). The method *applyExpand* is presented in Figure 18. Here message *expand* is sent to the current object; in this case, the method *expand* is implemented in class **IntEquation**. So, adding a new tool is made in two steps, in a matter of minutes :
(a) implementing a method in class **FEMTheoryMainView** for which the name is the string 'apply' completed by the name of the push-button
(b) adding this new tool name in the list of the object being manipulated.



Line 1: INT { ((Sij,j+Ri)(Wi)) // D }+INT { ((Ui,i+LP)(Q)) // D } = (0)

Line 2: INT { (Sij,jWi) // D }+INT { (RiWi) // D }+INT { (Ui,iQ) // D }+INT { (1/LPQ) // D } = (0)

Opening of a problem stored in a file on disk
Save the current problem on disk
Creation of a new problem
Tools bar
View of the object at the current step of the derivation
Tools applicable on the current object
Application of the selected tools on the current object
Inspect the current object
Automatic generation in the code FEM_Object
Access to the numerical code FEM_Object
Type of the current object
Access to pre- and post-processing of data
Access to a dictionary of shape functions
Check Indicial Notation
Check Dimension
Find Dimension For Selected Term
Define Dimension For Selected Term
Expand
Substitute Terms In Selected Integral
Discretize
Integrate By Parts Selected Integral
Remove Selected Product
Give Directional Derivative
Compute Consistent Linearization
Apply selected tool on the current object
Inspect
Current object is an equation
View
Finite Element Code FEM_Object
C++
Smalltalk
FEM_Object
Pre- and postprocessing
Preprocessing
FEM_Theory post pro
Shape functions dictionary

Figure 16 Screen of FEM_Theory

```
applySelectedTool                                               Selector of the method

  "Callback for the #clicked event in a Button (contents is 'Apply ').   Comments
   (Generated by WindowBuilder)"

      | methodName arguments |                                  Temporary variables

  methodName := (self paneNamed: #Tools) selectedItem.          Ask the push-buttons to answer the name of the selected tool
  methodName notNil                                             If a tool is selected...
    ifTrue:[
      self changeToBusy.                                        Activity flag of the window is changed to  busy state
      methodName := 'apply' , (methodName without:$ ).          The selector of method is built for current tool (string)
      self perform: ( methodName asSymbol ).                    The code generated at previous line is dynamically executed
      self changeToNormal.                                      Activity flag of the window is changed to free state
          ]
```

Figure 17 Method *applySelectedTool* of class **FEMTheoryMainView**

```
applyExpand                                                              Selector of method

  ((self giveObject isIntEquation) or: [self giveObject isDiscretisedEquation]) Check the nature of the object
    ifTrue: [  self object: (self giveObject expand)]                    Send the message 'expand' to the current object;
                                                                         the result becomes the new current object
    ifFalse: [Prompter prompt: (self giveObject printString)
                        default: 'Not IntEquation or DiscretizedEquation']. Prompt an error message if not allowed
  self printObjectOnBlackboard.                                          Print the new current object on the pane of the window
```

Figure 18 Method *applyExpand* of class **FEMTheoryMainView**

2.4.3 Description of the available tools to derive finite elements

In the previous section the functioning principle for the interface was addressed. In this section, the main available tools are described.

2.4.3.1 Tools applicable to variational forms

The tools given on Table 6 are applicable to equations, i.e. instances of **IntEquation**, or to selected objects composing them.

Table 6 List of tools available for variational formulations of a continuum

"Expand"	The current equation is asked to expand both sides. The properties of distributivity and the linearity of the integral are applied.
"Integrate By Parts Selected Integral"	Integration by parts applies to an integral selected in the line representing the current object, and this integral is integrated by parts. The divergence theorem is applied after integration.
"Substitute Terms In Selected Integral"	Substitution applies to an integral selected in the last line ; an edition window permits substitution of one term or more in the integrand by another expression.
"Discretize"	The weak form is sent the message to get its approximation and the discretization of the different fields on the element. The user is then requested to specify the discretization, the space dimension, etc...; all this information is requested by the object which needs it.
"Check Indicial Notation"	Check the coherence of the notation introduced by the user
"Check Dimension"	Perform a dimensional analysis of the current object; check the coherence of the units (see chapter 4)
"Find Dimension For Selected Term"	Find the dimension of the selected term by analyzing the current object (can only handle simple situations, see chapter 4)
"Define Dimension For Selected Term"	Associate a dimension with the selected term
"Remove Selected Product"	Remove selected product of the current object
"Give Directional Derivative"	Compute the directional derivative of the current object using theory presented in [HUG 78]
"Compute Consistent Linearization"	Compute a consistent linearization of the current object (see [HUG 78] and chapter 6 for more details)

2.4.3.2 Tools applicable to discrete forms

The tools of Table 7 are applicable to instances of **DiscretizedEquation**.

Table 7 List of tools available for discrete variational formulations

"Invoke Linear Independence"	This button invokes the arbitrariness of the weighting functions chosen for the current problem (linear independence of the equation of the linear system)
"Transpose"	Each component of the current object, a discretized equation, is a matrix. This tool permits the user to transpose each side of the equation, and the corresponding matrices.
"Shape Functions Replacing"	This button permits the replacement of shape functions by their expressions. It gives access to a dictionary of predefined shape functions, which can of course be enriched easily.
"Rename"	Permits renaming the selected term in the current object
"Remove Selected Product"	Remove selected product of the current object
"Add A Perturbation Term"	Add a term to the current discrete object (see discussion on stabilized methods in chapter 5)
"Add Methods"	Automatic generation of C++ code corresponding to the selected elemental form

The tools applicable to systems of discrete equations are similar to the one presented on Table 7.

The last tools applicable to discrete forms are the ones needed for automatic finite element coding for the code **FEMObject** (see Chapter 2 and [ZIM 92][DUB 92][DUB 93]. Push-buttons to generate either Smalltalk or C++ code are part of the main window of **FEMTheory** (see Figure 16).

2.4.4 About the interface

The simple and truly object-oriented features applied above to the interface can be generalized and applied to different situations. Here objects are represented only by strings. The same scheme could be applied to views which could be closer to natural mathematical notations, using e.g. a bitmap system.

2.5 A first example of derivation: a one-dimensional elastic bar

The easiest way to get a feeling of the proposed approach is to follow a demonstration.

2.5.1 The problem

The case of an elastic uniaxial truss in dynamics is illustrated hereafter. The problem statement is given on Table 8; this corresponds to a particular case of the problem stated on Table 1

Table 8 One-dimensional elastic bar problem in dynamics

Find u displacement with appropriate continuity such that :

$N_{,x} + R = \rho A u_{,tt} \text{on} \, \Omega \times T \qquad \Omega \subset \Re$

R are the body loads

A is the area of the section

Boundary conditions :

$N = F \quad \text{on} \, \partial_2 \Omega \times T$

$u = \bar{u} \quad \text{on} \, \partial_1 \Omega \times T$

Constitutive equation :

$N = EA\varepsilon$

Initial conditions :

$u_{,t}(t = 0, x) = (u_{,t})_0 \quad \text{on} \, \Omega$

$u(t = 0, x) = u_0 \quad \text{on} \, \Omega$

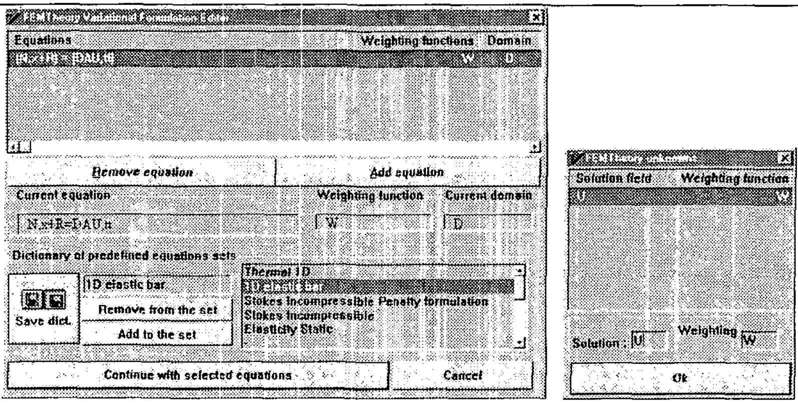
Kinematics law :

$\varepsilon = u_{,x}$

2.5.2 Derivation of the matrix form

The activation of a new problem gives access to two successive editors (see Figure 19) where the differential equations, trial solution and weighting functions can be defined using the notations defined above. The first window give access to a dictionary of a predefined set of equations corresponding to various problems. This dictionary can of course be enriched.

The resulting variational statement is posted on the screen of Figure 20, on Line 1.


FEMTheory Variational Formulation Editor

Equations	Weighting functions	Domain
N.x+R=DAU.tt	W	D

Current equation: N.x+R=DAU.tt

Weighting function: W

Current domain: D

Dictionary of predefined equations sets: 1D elastic bar

- Thermal 1D
- 1D elastic bar
- Stokes Incompressible Penalty formulation
- Stokes Incompressible
- Elasticity Static

FEMTheory unknowns

Solution field	Weighting function
U	W

Solution : U

Weighting W

where :

$N_{,x}$ is the first derivative of the traction stress

R represents the body loads

u is the axial displacement (solution)

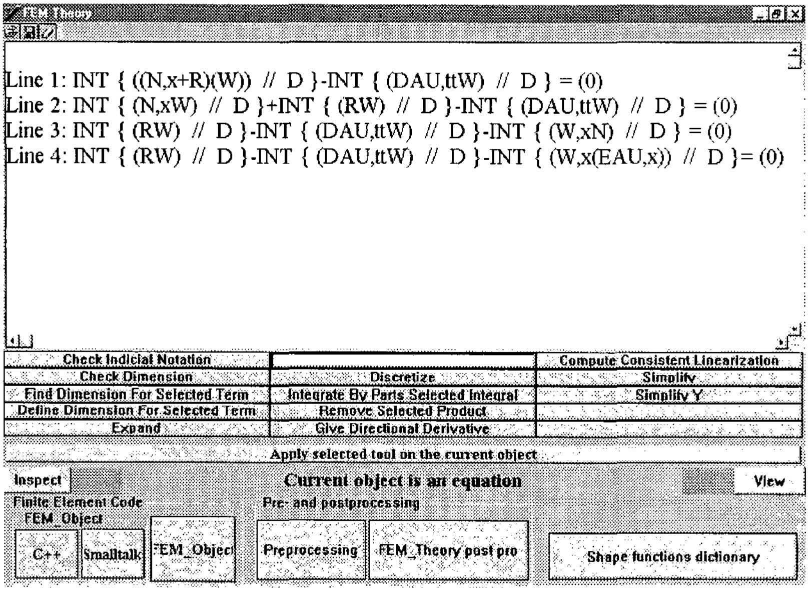
w is the weighting function for the axial displacement

D is the density

A is the area

$u_{,tt}$ is the second derivative of the axial displacement (acceleration)

Figure 19 Definition of the initial statement in FEMTheory



```
Line 1: INT { ((N,x+R)(W)) // D }-INT { (DAU,ttW) // D } = (0)
Line 2: INT { (N,xW) // D }+INT { (RW) // D }-INT { (DAU,ttW) // D } = (0)
Line 3: INT { (RW) // D }-INT { (DAU,ttW) // D }-INT { (W,xN) // D } = (0)
Line 4: INT { (RW) // D }-INT { (DAU,ttW) // D }-INT { (W,x(EAU,x)) // D }= (0)
```

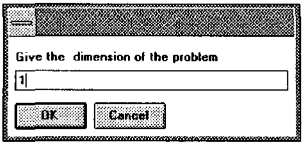
Check Indicial Notation
Check Dimension
Find Dimension For Selected Term
Define Dimension For Selected Term
Expand
Discretize
Integrate By Parts Selected Integral
Remove Selected Product
Give Directional Derivative
Compute Consistent Linearization
Simplify
Simplify Y
Apply selected tool on the current object
Inspect
Current object is an equation
View
Finite Element Code
FEM_Object
C++
Smalltalk
FEM_Object
Pre- and postprocessing
Preprocessing
FEM_Theory post pro
Shape functions dictionary

Figure 20 Derivation of continuum problem in the screen of **FEMTheory**

The expansion of selected expressions results from manipulations on lists activated by pushing the proper button. Notice that both terms of the instance of **IntEquation** are expanded. The result is shown on line 2, Figure 20.

Integration by parts requires the identification of the integral to work on. The integral is selected on Line 2, and the button *"Integrate by parts"* is pushed. The result appears on line 3 Figure 20. Notice that no boundary conditions and no nodal loads are taken into account ; these will be considered in the numerical part of the solution. They do not appear explicitly here, but should be kept in mind.

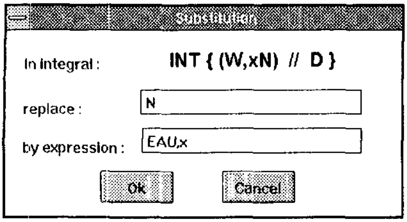
Additional information, such as the problem dimension is simultaneously requested from the user, using prompters like the one shown in **Error! Reference source not found.**. The constitutive law can now be introduced to replace *N* by *EAU,x* (see **Error! Reference source not found.**). This concerns the third integral on Line 3. The result of the substitution appears on line 4. of Figure 20.



Give the dimension of the problem

1

Figure 21 An example of prompter



In integral: INT { (W,xN) // D }

replace: N

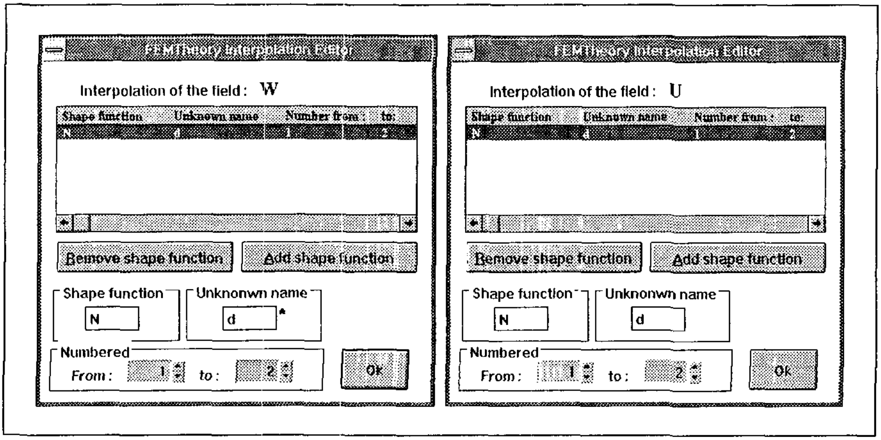
by expression: EAU,x

Figure 22 Prompter for substitution

Notice that substitution replaces instances of class **Term** by an instance of **Expression** without performing any mathematical operation. The introduction of the Neumann boundary condition also corresponds to a substitution which is purely formal, the associated boundary domain being unknown at this stage.

Discretization and interpolation requires the introduction of locally based approximations for u and w such that e.g. $u = \sum_{i=1}^{2} N_i d_i$ and $w = \sum_{i=1}^{2} N_i d_i^*$. The windows used to introduce them are shown in Figure 23.

A matrix form of the constitutive equation is introduced next. The result is shown on Line 5, in Figure 24.



Interpolation of the field: W

Shape function	Unknown name	Number from:	to:
N	d	1	2

Shape function: N

Unknonwn name: d*

Numbered From: 1 to: 2

Interpolation of the field: U

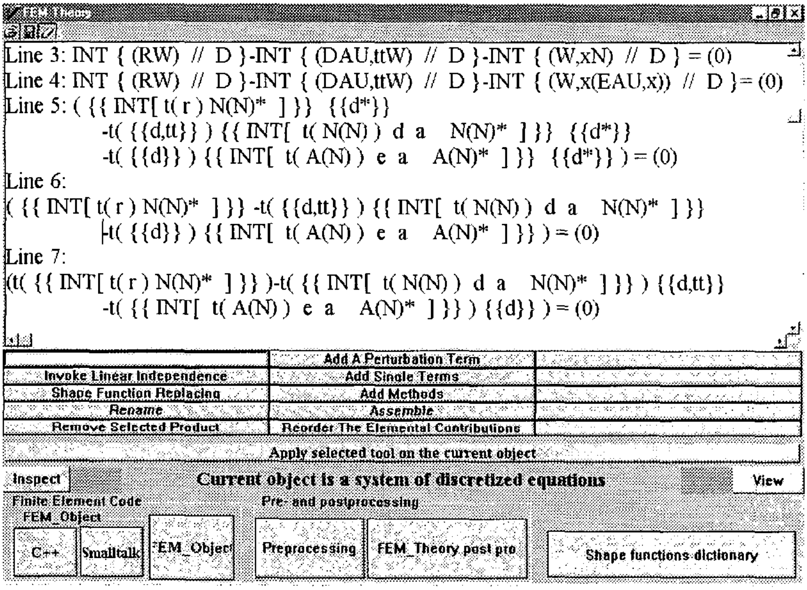
Shape function	Unknown name	Number from:	to:
N	d	1	2

Shape function: N

Unknonwn name: d

Numbered From: 1 to: 2

Figure 23 Interpolation choice



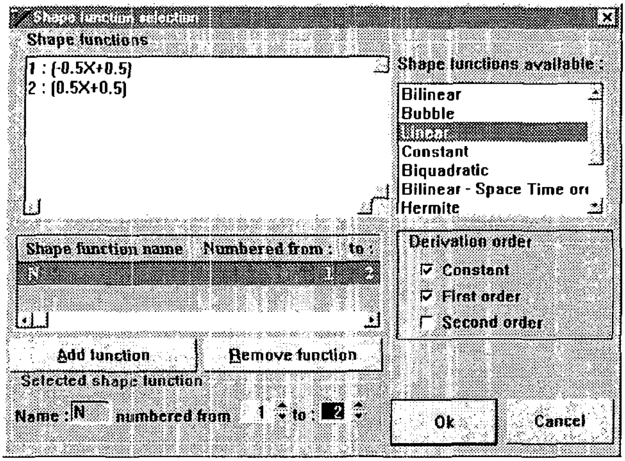
```
Line 3: INT { (RW) // D }-INT { (DAU,ttW) // D }-INT { (W,xN) // D } = (0)
Line 4: INT { (RW) // D }-INT { (DAU,ttW) // D }-INT { (W,x(EAU,x)) // D }= (0)
Line 5: ( {{ INT[ t( r ) N(N)* ] }} {{d*}}
          -t( {{d,tt}} ) {{ INT[ t( N(N) ) d a  N(N)* ] }} {{d*}}
          -t( {{d}} ) {{ INT[ t( A(N) ) e a  A(N)* ] }} {{d*}} ) = (0)
Line 6:
( {{ INT[ t( r ) N(N)* ] }} -t( {{d,tt}} ) {{ INT[ t( N(N) ) d a  N(N)* ] }}
          -t( {{d}} ) {{ INT[ t( A(N) ) e a  A(N)* ] }} ) = (0)
Line 7:
(t( {{ INT[ t( r ) N(N)* ] }} )-t( {{ INT[ t( N(N) ) d a  N(N)* ] }} ) {{d,tt}}
          -t( {{ INT[ t( A(N) ) e a  A(N)* ] }} ) {{d}} ) = (0)
```

Current object is a system of discretized equations

Figure 24 Derivation of semi-discrete form on the screen of FEMTheory

Arbitrariness of the virtual field d^* is then invoked; the result appears on Line 6 Figure 24 (the star indicates a virtual field). The equation is finally transposed, the resulting matrix formulation of the problem is shown on Line 7 Figure 24; it corresponds to the usual matrix form of the problem, i.e. $Md'' + Kd = f$.

The actual choice of the shape functions occurs at this stage along with the integration scheme. The corresponding editor proposes a selection of shape functions, as illustrated in Figure 25. This editor gives access to a dictionary of shape function, object instance of **FEMTheoryShapeFunctionsDictionary** (see section 2.3.2.1.a and Annex A). Now, the integrands have taken a form which is ready for coding, e.g. the element (1,1) of the elemental matrix {{INT[t(A) t(E) t(B(N)) B(N)*]]}} corresponding to the stiffness matrix is given by (using the proposed notation) : INT { (((((1/((X1(-0.5))+(X2(0.5))))(-0.5)))((E)(A)))(((1/((X1(-0.5))+(X2(0.5))))(-0.5)))) // D }. This string of characters can now be interpreted to generate the code in an appropriate language, e.g. Smalltalk or C++. These aspects are discussed at length in the next chapter.



Shape function selection

Shape functions

1 : (-0.5X+0.5)
2 : (0.5X+0.5)

Shape functions available :

Bilinear
Bubble
Linear
Constant
Biquadratic
Bilinear - Space Time ori
Hermite

Shape function name	Numbered from :	to :
N	1	2

Derivation order

☑ Constant
☑ First order
☐ Second order

Selected shape function

Name : N numbered from 1 to : 2

Figure 25 Shape functions selection

Chapter 3 Concepts for automatic programming of finite elements

The use of symbolic software such as Mathematica or Macsyma can be really helpful either for directly solving finite element problems or for simplifying expressions in numerical computations of finite element problems. This is illustrated for example in [YAG 90, LEF 91, IOA 92, CHO 92]. Automatic generation of finite element elemental contributions and their implementation into a finite element code can be found in [CEC 77, KOR 79, WAN 86]. All these papers show the power of symbolic computation tools, and show their usefulness to generate numerical code. In chapter 2 and in [ZIM 96, EYH 96a], the basis of an object-oriented environment for finite element code generation is presented. The aim of this chapter is to give the principles of automatic generation of finite element code in this object-oriented environment. The description is given in the context of a finite element code developed in Smalltalk language (see [ZIM 92b]; extension into a C++ finite element code (see [DUB 92b]) is also possible, in order to achieve efficiency. A first application is made in the example of elastic bar seen in the previous chapter. These principles are illustrated on a classical formulation of linear elasticity in dynamics and on a penalty formulation for Stokes flow problem.

3.1 Automatic generation of a finite element code in a symbolic object-oriented environment

In this section, the principles of automatic programming are described, and code is generated into the object-oriented Finite Element code FEM_Object, Smalltalk version (see [ZIM 92b]). The reader should refer to chapter 2 for a complete description of the structure of the symbolic environment. In first subsection, the principles of automatic implementing an object-oriented code are given; in the second subsection the way of adding dynamically classes and methods in the Smalltalk class hierarchy is described, and is used in the third subsection for finite element code generation.

3.1.1 Principle of automatic generation of a code in a symbolic object-oriented mathematical environment

The object oriented data organization adopted here, leads to an easy and natural code generation. Most of the information needed to develop part of the code corresponding to the new theory are contained in the object characterizing the problem, i.e. an instance of class **System**. The reader should refer to the previous chapter for details about the different structures representing the problem. Roughly speaking, only the code corresponding to elemental matrix contributions needs to be generated and added to an existing finite element code, using the proper language. In the following, the automatic generation of a new element in the object-oriented code FEM_Object, written in Smalltalk, is discussed. Code generation in C++ for FEM_Object in C++ version, for example was also done and follows the same principles.

The key point in this part is, on the one hand, to reuse the code corresponding to data management (nodes, degree of freedom, ...), for the solution of the global linear system resulting from the discretization and, on the other hand to particularize the behavior corresponding to the computation of the elemental vectors and matrices. The data organization of FEM_Object (see [ZIM 92a] and [DUB 92a]) leads to the creation of a new subclass of the class **Element**. This class will be given the behavior needed to compute the elemental contributions corresponding to the formulation. The particularity of the Smalltalk environment is that the creation of the code is made dynamically, i.e. when Smalltalk and FEMTheory are running.

As already mentioned, the result of the symbolic derivation is an instance of class **System**. This instance is sent the message to create the code. The message goes over all the objects composing the instance, each of them realizing a particular task using its encapsulated information. Our aim in this chapter is to describe all the tasks performed by the instances forming the system.

The problem consists now in creating a new subclass of the class **Element** into which the elemental matrices are built. All the information needed to achieve this is encapsulated in the objects participating in the instance of class **System**, representing the last step of the symbolic derivation. Each of them has to bring its contribution to the generated code and so as naturally as possible. As usual in object-oriented approaches, it is better for the reusability of the code to decentralize the operations in order to be as general as possible. First it is necessary to see how it is possible to add a subclass to a class , and how to add instance and class methods in a given class. Then, each object embedded in the system will have a particular task in the generation of the element for the finite element code. It is interesting to note that this scheme is quite general for the creation of a code in any language, only the operation to be performed would be different, since obviously there is no creation of class in Pascal or FORTRAN.

3.1.2 Programming in Smalltalk

In this part the key points of the implementation in Smalltalk language are given. Details of the language can be found in [VIS 95b].

The power of the Smalltalk environment is that, as already mentioned, the addition of a code can be done dynamically. This feature is worth detailing. Programming in Smalltalk means enriching the native hierarchy, this operation can be made during the execution itself. The advantage is that it is possible to add a code and to use it as soon as created.

There are three main types of additions to an existing hierarchy : the first is adding a class, the second adding an instance method and the third a class method. Details about these techniques can be found in [EYH 96a].

3.1.3 Finite element automatic programming in the FEM_Theory environment

After this description of implementation in Smalltalk in a general way, let us look at the different subclasses of FEM_Theory to describe the generation of the element.

The result of the symbolic derivations is a system of discretized equations; in elastodynamics it is for example $\boldsymbol{Md}_{,tt} + \boldsymbol{Kd} = \boldsymbol{f}$. It is important to know the structure of this symbolic object to understand the process of the code generation. Illustration is given in chapter 2, in Figure 3. The principle of message passing is the same as for other operations like transposition, discretization, etc.... It is explained in section 2.3.3 and in [EYH 94]. The message goes over from one object of the system to the next, following the **hierarchicParent** tree. Figure 26 illustrates it in the example of the discrete system for the elastic bar in dynamics derived in the previous chapter. The system (instance of class **System**) has one equation, which has a left-hand side and a right-hand side. The left-hand side is a sum of a product of matrices (instance of **DiscretizationMatrix**). The structure of this object is illustrated on the stiffness matrix K. This object is characterized by an elemental matrix, here a two-by-two matrix. Each coefficient of the matrix is an integral, for which the integrand is an expression. Messages for code creation follow this tree.

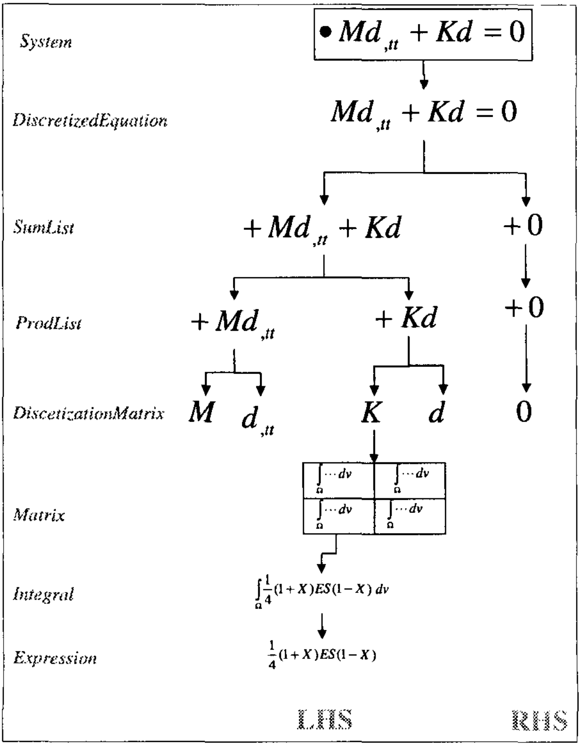


Figure 26 Hierarchical tree for a system of discrete equations

Class System

This object is the one manipulated by the user in the graphical interface. The command to create a new element in FEM_Object is sent to the environment by the user through a push button (see section 3.4.2). The message sent is *aSystem createNewElement*. Through this message a few tasks are activated (see Figure 27). As the class **Element** encapsulates the behavior associated with a specific theory in FEM_Object (e.g. elastodynamics), likewise the system possesses all the knowledge to build this element, which knowledge is encapsulated in several objets. Each of them brings its own contribution to the new element.

```
createNewElement
        " Create a new element in class Element of FEMObject. "
   | newElernent   aBag  result  string1   equation |

"1 - CREATION OF A NEW SUBCLASS OF CLASS Element IN FEMObject"
newElement := self askTheUserTheNewElementName.
Element subclass: (newElement asSymbol )
            instanceVariableNames: ''
            classVariableNames: ''
            poolDictionaries: ''.
self createMethodsIn: (newElement asSymbol).
self createInstanciationMethodIn: (newElement asSymbol).


"2  -  CREATION   OF   THE   METHODS   MANAGING   THE   ATTRIBUTE
jacobianMatrix
 IN THE NEW ELEMENT IF IT DOES EXIST "
(newElement asSymbol asClass) hasAnAttributeNamed: 'jacobianMatrix'
     ifFalse: [
            Element subclass: (newElement asSymbol )
                        instanceVariableNames: ' jacobianMatrix  '
                        classVariableNames: ''
                        poolDictionaries: ''.
            (self              giveJacobianMatrix)              createMethodWithArgument:
#giveJacobianMatrixAt:
                                                                   inElement:   (newElement
asSymbol).
                ].


"3  -  CREATION   OF   THE   METHODS   MANAGING   THE   ATTRIBUTE
jacobianMatrix
 IN THE NEW ELEMENT IF IT DOES EXIST "
equation  createMethodsInElement: (newElement asSymbol) .
```

Figure 27 Method of class **System** to create a new element

Firstly, the most important task that the system has to perform is to create the subclass of class **Element**. The user is asked the name of the new element with a prompter message. Then the new element is given four class methods. The first one is the instanciation method *new*. The method to be implemented is shown in Figure 28. The problem consists in constructing the string corresponding to this method and to compile it in the element environment. The method for doing this is *createInstanciationMethodIn: newElement*. This method uses the code creation described in section 3.1.2. At this point, the system is asked the number of nodes of the element needed to create the method *new*.

```
new

            ^super new setNumberOfNodesTo: 2
```

Figure 28 Example of method **new** to instanciate new elements

The other three class methods added to the new class, are the ones which return the number of nodes, return the number of geometric nodes, and the space dimension, all this information coming from the system's arguments ; the method for doing this is *createMethodsIn: newElement.*

Secondly, the system creates the code corresponding to the management of the numerical jacobian matrix (see class **Integral** for example) ; this comes from the fact that the class system possesses all the characteristics for the change of variable from local to global coordinate axes embedded in attribute **geometryReference**.

Thirdly, the element must be equipped with methods in order to be capable of performing tasks to compute elemental contributions .The last part of the method *createNewElement*, is to send the message *createMethodsInElement: newElement* to the instances of class **DiscretizedEquation**, which embodies the initial boundary value problem. As the system has already created all the things it could, it now requests its equations to perform their tasks.

Class DiscretizedEquation

The only method (see Figure 29) which has to do with the creation of a new element is *createMethodsInElement: newElement.* In this method, all the terms are put on the left hand side of the equation which is sent the message *createMethodsInElement: newElement* ; the left-hand-side is an instance of **DiscretizedExpression**.

```
createMethodInElement: newElement

    self putAllLhs.
    self giveLhs createMethodsInElement: newElement
```

Figure 29 Method of class **DiscretizedEquation** to create new methods

Class DiscretizedExpression

In the method *createMethodsInElement: newElement* , the argument **sumList** is asked to create its own methods (see Figure 30).

```
createMethodsInElement: newElement

self giveSumList createMethodsInElement: (newElement asSymbol).
```

Figure 30 Method of class **DiscretizedExpression** to create new methods

Class SumList

In the method *createMethodsInElement: newElement* , all the components of the instance, here instances of **ProdList**, are asked to create their own methods (see Figure 31)

```
createMethodsInElement: newElement

self do:[:pList| pList createMethodsInElement: newElement ].
```

Figure 31 Method of class **SumList** to create new methods

Class ProdList

The tasks assigned to this class in code generation (see Figure 32) are more complex than those seen above. It is important to recall that the code is introduced in an existing code capable of dealing with first and second order time derivative semi-discrete problems. At this stage, the instance of **ProdList** only contains instances of class **DiscretizationMatrix**. These are sent the message to create a code with the chosen selector of method.

Consider the products *Kd* and $Md_{,tt}$. For the first one the matrix *K* corresponds to a stiffness matrix and *M* to a mass matrix. The parameter which distinguishes them is one nodal unknown vector, *d* or $d_{,tt}$. This check is made here. Then the instance of **DiscretizationMatrix** is asked to create a method named *computeStiffnessMatrix* or *computeMassMatrix*. Notice that if the product has only one factor, the matrix is asked to create methods to compute load vectors.

```
createMethodsInElement: newElement

(self size =1)
        ifTrue:[( self at: 1) createLoadMethodsInElement: (newElement asSymbol) ].

(self size = 2)
ifTrue:[
        ((self at: 2) isTimeConstant)
                ifTrue:[(self at: 1) createMethod: #computeStiffnessMatrix
                                        inElement: (newElement asSymbol)].
        ((self at: 2) isTimeSecondDerived)
                ifTrue:[(self at: 1) createMethod: #computeMassMatrix
                                        inElement: (newElement asSymbol)].
        ].
```

Figure 32 Method of class **ProdList** to create new methods

Class DiscretizationMatrix

Ensuing from the preceding paragraph (see Figure 32) , this class has two behaviors for code generation.

The first one (see Figure 33) is the creation of a method with a given selector ; the attribute **elementaryMatrix** is sent the message *createMethod: aSelector inElement: newElement.*

```
createMethod: aSelector inElement: newElement

   self giveElementaryMatrix createMethod: aSelector
                             inElement: newElement.
```

Figure 33 Method of class **DiscretizationMatrix** to create new methods

The second method (see Figure 34) concerns the loads methods. The matrix checks if it is defined on the domain or on its boundary, and according to the nature, applies the above method with the selector *computeSurfaceLoadVector* or *computeBodyLoadVector.*

```
createLoadMethodsInElement: newElement

(self isBodyLoadMatrix)
     ifTrue:[ self createMethod: #computeBodyLoadVector
                    inElement: newElement].
(self isSurfaceLoadMatrix)
     ifTrue:[ self createMethod: #computeSurfaceLoadVector
                    inElement: newElement].
```

Figure 34 Method of class **DiscretizationMatrix** to create methods to compute loads

Class Matrix

The attribute **elementaryMatrix** is an instance of class **Matrix**. The class Matrix has two different methods of code creation. The first one creates a method to get the numerical matrix, the second one creates the same method with the capability of passing an argument. For example, an element may be sent the message *self computeMassMatrix* . For the creation of this method the symbolic matrix *aMatrix* is sent *aMatrix createMethod: #computeMassMatrix inElement: newElement.* In the same way, an element may be sent the message *self computeJacobianMatrixAt: gaussPoint2.* Here, the symbolic matrix is sent *aMatrix createMethodWithArgument: #computeJacobianMatrixAt: inElement: newElement.*

Both methods are built on the same principle (see Figure 35 for the first one); a first part creates the method which can return the numerical form of the matrix (method *computeMassMatrix* for example); a second part creates the methods which compute each component of the matrix.

This second part of the method consists in sending to each component of the symbolic matrix the message to create a method to compute itself with the correct selector. Thanks to polymorphism, whatever the class of the component may be, the right methods will be created. There is no anticipation of the nature of the components of the matrix (given in a symbolic way). Each of them may be an instance of **Integral, Matrix** and **Expression**. So all of these classes have both methods :

createMethod: newMethod inElement: newElement
createMethodWithArgument: newMethod inElement: newElement

Suppose the mass matrix is a 2-2 matrix; method *computeMassMatrix* is illustrated in Figure 36.

```
createMethod: newMethod  inElement: newElement

        | string1 dim result newselect |
dim := self dimensions.

"1 - CREATION OF THE METHOD OF CLASS newElement WHICH
REPLIES THE NUMERICAL MATRIX"
 string1 := (newMethod asString) , ' | dim aselector k |
        dim := ' ,(dim asString), '. k := Matrix new: dim.'.
1 to: (dim x)   do:[:i|
        1 to: (dim y) do:[:j|
string1 := string1 , 'k at:(' , (i asString) , '@' , (j asString) ,') put: ( self  ', (newMethod asString) , (i asString) , (j asString) ,
'.'.
                        ].
                    ].
string1 := string1 , ' ^k ' .

result := ((Smalltalk at: newElement asSymbol) compile: string1).
(Smalltalk at: newElement asSymbol) addSelector: (newMethod asSymbol)
                                    withMethod: ( result value).
Smalltalk logSource: string1
          forSelector: (result key)
             inClass: (Smalltalk at: (newElement  asSymbol ) ).


"2 - CREATION OF THE METHODS OF CLASS newElement WHICH REPLIES
THE (i@j) COMPONENT OF THE NUMERICAL MATRIX"

1 to: (dim x) do:[:i |
      1 to: (dim y) do:[:j |
             newselect :=( newMethod asSymbol) , (i asString) ,(j asString).
             (self at: (i@j)) createMethod: (newselect asSymbol)
                               inElement: newElement .
                    ].
             ].
```

Figure 35 Method of class **Matrix** to generate methods

```
computeMassMatrix
        | dim aselector k |
dim := 2@2.
k := Matrix new: dim.

k at: (1@1) put: (self  computeMassMatrix11).
k at: (1@2) put: (self  computeMassMatrix12).
k at: (2@1) put: (self  computeMassMatrix21).
k at: (2@2) put: (self  computeMassMatrix22).

^massMatrix := k
```

Figure 36 Example of method generated to compute a matrix

Remark 1 : in Smalltalk, a string representing a source code can be dynamically executed. The code shown in Figure 36 can be replaced by the one shown in Figure 37. The principle is here to create the message dynamically to compute component i-j. This is an easy way to generate a short code automatically. This short cut cannot be implemented into a non-interpreted language (C++ for example) and so, the code will be similar, in the structure, to the one of Figure 36.

```
computeMassMatrix
            | dim aselector k |
dim := 2@2.
k := Matrix new: dim.
1 to: (dim x) do:[:i|
            1 to: (dim y) do:[:j|
                        aselector := 'computeMassMatrix',(i asString),(j asString).
                        aselector := aselector asSymbol.
                        k at: (i@j) put: (self perform: aselector).
                                                   ].
                                        ].
^ k
```

Figure 37 Example of simplified method to compute a matrix

Remark 2 : this principle of the automatic generation of a method passing no or one argument could easily be extended to the passing of multiple arguments.

Remark 3 : the same applies to the method *createMethodWithArgument: newMethod inElement: newElement*. In the following paragraphs, as both methods are based on the same principles, only one is discussed.

Class Integral

The method *createMethod: newMethod inElement: newElement* (see Figure 38) depends on the numerical integration scheme chosen by the user. At this stage of the development only the gauss integration is available, and used here as an illustration. Notice that any kind of integration scheme could easily be added here.

```
createMethod: newMethod inElement: newClass
      | integrationScheme |

integrationScheme := self giveNumericalIntegrationScheme.

(((integrationScheme  = 'gauss1pt')
      or:[integrationScheme = 'gauss2pts'])
                  or:[integrationScheme = 'gauss3pts'])
      ifTrue:[self createMethodGaussianIntegration: newMethod inElement: newClass ]
```

Figure 38 Method to switch to user defined numerical integration

The type of numerical integration scheme is known by the system, instance of class **System**, if the same numerical scheme has been chosen by the user for all the integrals.

The following tasks are performed by the method *createMethodGaussianIntegration: #newMethod inElement: newElement* (see Figure 39).

Another aspect is the change of variable because the integrands are expressed in local coordinate axes (from a mathematical point of view $\int_{\Omega^e} f(X)dX = \int_{\Omega^e} f(X(x))J(x)dx$ where $J(x)$ is the Jacobian determinant and x the variable in the local coordinate axes).

The process of creation of methods may be split into three actions :

- a method which returns the gauss points for the numerical scheme is created
- a method which manages and computes the integral is created
- the last part consisting in sending to the integrand the message *createMethod: #newMethodFunctionAt: inElement: newElement* in order to generate the code to compute it.

Remark : the selector of the method used for computing the integrand at a given point is constructed by adding *functionAt:* to the one permitting to compute the integral.

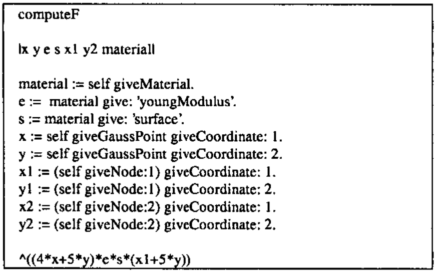
```
 createMethodGaussianIntegration: newMethod inElement: newElement
        | string1  result  aBag |

 "1 - CHECK IF THE ATTRIBUTE CORRESPONDING TO GAUSS COORDINATES EXISTS IN THE
NEW ELEMENT, IF NOT CREATE IT"
 (anElementClass asSymbol asClass hasAnAttributeNamed:((self giveNumericalIntegrationScheme) ,'Array'))
        ifFalse: [self createGaussPointsMethodsIn: newElement] .


 "2 - CREATION OF THE METHOD OF THE CLASS WHICH REPLIES THE NUMERICAL
COMPUTATION OF THE INTEGRAL"
   string1 := (newMethod asString) , '
                | f jacobian weight reply  |
   reply := 0.
   self give' , (self giveNumericalIntegrationScheme ), 'Array do:[: gp |
                f := self ',( newMethod),'functionAt: ( gp giveCoordinateArray) .
                jacobian := (self giveJacobianMatrixAt: ( gp giveCoordinateArray)) determinate.
                weight := gp giveWeight .
                reply :=  f *  jacobian * weight + reply.
                                ].
    ^ reply'.

     result := ((Smalltalk at: newElement asSymbol)  compile: string1).
    (Smalltalk at: newElernent asSymbol)   addSelector: (newMethod asSymbol)
                                            withMethod: ( result value).
       Smalltalk logSource: string1
              forSelector: (result key)
                  inClass: (Smalltalk at: (newElement asSymbol )).


 "3 - SENDING OF THE MESSAGE TO THE INTEGRANT TO CREATE THE METHODS TO
COMPUTE NUMERICALLY THE INTEGRAND" "
   self giveIntegrand createMethodWithArgument: (  (newMethod),'functionAt:' )
                                          inElement: newElement.
```

Figure 39 Method of class **Integral** to generate a numerical Gaussian quadrature

Class Expression

The process to create a method to evaluate numerically a symbolic expression is based on the following principle.

Take an expression (4.x+5.y).e.s.(x1+5.y). The method to be created to compute this expression is shown in Figure 40. The first part of this method, the assignation of variables, depends on the language of implementation and on the preexisting Finite Element code. The last line is the result of the expression method *printString*. The source code corresponding to the declaration of the variable is the same for every expression (in a given context); the last line is added by each expression. The string which has been built may then be compiled into the new element.



```
computeF

|x y e s x1 y2 material|

material := self giveMaterial.
e := material give: 'youngModulus'.
s := material give: 'surface'.
x := self giveGaussPoint giveCoordinate: 1.
y := self giveGaussPoint giveCoordinate: 2.
x1 := (self giveNode:1) giveCoordinate: 1.
y1 := (self giveNode:1) giveCoordinate: 2.
x2 := (self giveNode:2) giveCoordinate: 1.
y2 := (self giveNode:2) giveCoordinate: 2.

^((4*x+5*y)*e*s*(x1+5*y))
```

Figure 40 Example of method generated to compute an expression

Remark 1 : this section describes code generation in a F.E. code in Smalltalk. This can be extended to any finite element code written in any language. At this stage of development, code generation exists for the object-oriented finite element code FEM_Object written in Smalltalk and in C++ (see [DUB 92] and [DUB 93]). The principles of automatic generation in the symbolic environment are the same. An additional difficulty in C++ is that the language is strongly typed. So during the generation of code for matrices, in particular, it is necessary to take into account the type of the object contained in the structure, generally double type or **FloatArray** type (see [DUB 93]). In a sense, the non-anticipation principle is violated to generate the source file.

Remark 2 : all the processes described here can easily be extended to mixed problems. Take the example of the following system :

$$\begin{cases} Md_{,tt} + Kd + G_1 p = f \\ G_2 d = 0 \end{cases} \qquad \text{(a)}$$

An assemblage procedure leads easily to the following form : $Au_{,tt} + Bu = F$ (b)

where : $A = \begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix}$, $B = \begin{bmatrix} K & G_1 \\ G_2 & 0 \end{bmatrix}$, $u = \begin{bmatrix} d \\ p \end{bmatrix}$ and $F = \begin{bmatrix} f \\ 0 \end{bmatrix}$.

The form (b) of the system can be treated as described in this section to generate automatically finite element codes. The supplementary structures added at this stage are a special type of sums and products called **FEMTheorySumStructure** and **FEMTheoryProductStructure** (see Appendix A for details of implementation). This scheme is illustrated on examples of Stokes flow in the next section.

Remark 3: the principles of code generation have been shown for matrices which do not need passing of arguments. For problems in which elemental matrices computation depends on time for example, it is necessary to pass arguments, such as the object **TimeStep** in FEM_Object (see [DUB 92] and [DUB 93]). This scheme is easily enhanced for it, without any changes in the general structure. The automatic generation of code for the multiple passing of arguments can easily be obtained from the scheme presented here.

3.1.4 An example of automatic implementation : a bar in elastodynamics.

3.1.4.1 A new element generated in FEM_Object.

The symbolic derivation of an elastic bar in dynamics is detailed in section 2.1.3. The derivation leads to a system of the form $Md_{,tt} + Kd = f$, where M is the mass matrix, K the stiffness matrix and f the load vector (M and K are 2*2 matrices, f is 2*1 matrix). The elemental forms of this system can be introduced in the finite element code FEM_Object. The new element called **NewElement** appears in the view of partial FEMObject class hierarchy in Figure 41 (see [ZIM 92a] for more details about the FEM_Object class hierarchy). As described before, the new class is a subclass of class **Element**. The methods given to the element **NewElement** are shown in Figure 42. They allow the computation of the stiffness matrix, the mass matrix, the body loads and the Jacobian matrix, the latter being used in some methods of computation of the integrals. Moreover, the new element is given the capabilities to create itself and to manage the Gauss points.

```
FEMObject
    FEMComponent
        Element
            PlaneStrain
            Truss2D
            NewElement
        Load
            BoundaryCondition
            BodyLoad
                DeadWeight
            NodalLoad
            SurfaceLoad
        LoadTimeFunction
            ConstantLTFunction
            PeakLTFunction
        Material
        Node
        TimeStep
    Dof
    Domain
    LinearSystem
    Tool
        GaussPoint
        Polynomial
```

Figure 41 Partial class hierarchy of **FEMObject**

```
computeBodyLoadVectorAt:
computeBodyLoadVector
computeBodyLoadVector11
computeBodyLoadVector11functionAt:
computeBodyLoadVector21
computeBodyLoadVector21functionAt:
computegauss2pts
givegauss2ptsArray
computeMassMatrix
computeMassMatrix11
computeMassMatrix11functionAt:
computeMassMatrix21
computeMassMatrix21functionAt:
computeMassMatrix12
computeMassMatrix12functionAt:
computeMassMatrix22
computeMassMatrix22functionAt:
computeStiffnessMatrix
computeStiffnessMatrix11
computeStiffnessMatrix11functionAt:
computeStiffnessMatrix21
computeStiffnessMatrix21functionAt:
computeStiffnessMatrix12
computeStiffnessMatrix12functionAt:
computeStiffnessMatrix22
computeStiffnessMatrix22functionAt:
giveJacobianMatrixAt11:
giveJacobianMatrixAt:
new
```

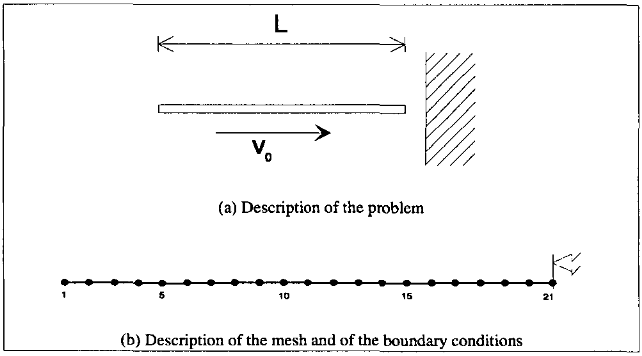
Figure 42 List of the methods created in the new element class

Remark 1: Figure 41 and Figure 42 show the class and the list of methods created in the environment of the finite element code FEM_Object in Smalltalk version. The class and the methods created in FEM_Object version C++ are exactly the same; their construction is done on the same principles.

Remark 2: The creation of the new element is done in a matter of minutes without any debugging steps for testing the element.

3.1.4.2 Test of the new element

A numerical problem is proposed to test the new element. The impact of a bar on a rigid surface is analyzed. The problem is described in Figure 43, where Figure 43a represents the situation, Figure 43b the mesh and the boundary conditions.



(a) Description of the problem

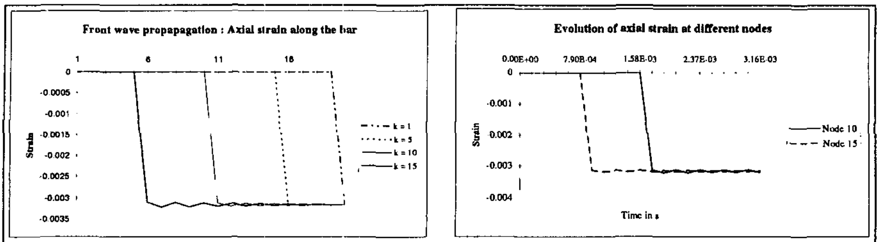
(b) Description of the mesh and of the boundary conditions

Figure 43 Description of the numerical test

The data used for the numerical example is :
Density : $D = 0.01$ kg.m^{-3}
Young modulus : $E = 1000$ N.m^{-2}
Length : $L = 1$ m

An explicit Newmark algorithm is used for time integration. The parameters are (see [HUG 87]): $\gamma = 0.5$ and $\beta = 0$. The time step chosen is $\Delta t = 1.58\ 10^{-4}$ s. The initial velocity of the bar is $v_0 = 1$ m.s^{-1}.

The exact solution is characterized by a wave emanating from the impact. The wave speed is $c = \left(\frac{E}{D}\right)^{\frac{1}{2}}$. The time step Δt corresponds here to the time required for the wave front to cross one element. The result is shown in Figure 44: Figure 44a shows the propagation of the wave along the bar at a different time step, and Figure 44b shows the characteristic time evolution of the strain at a given point on the bar. They are in agreement with the theory.



(a) Propagation of the wave front at a different time step (t = k Δt)

(b) Strain time-history at node 10 and 15

Figure 44 Numerical results for elastic bar

3.2 Classical formulations in structural and fluid mechanics

In this section, two classical formulations are presented to demonstrate the fast development capabilities of the FEM_Theory environment. One is taken from structural mechanics, and the other one from fluid mechanics.

3.2.1 Linear elasticity

Two different derivations have been conducted within the FEM_Theory environment. No fundamental novelty appears in the first derivation; the reference problem is the one of chapter 2. This can be found, completed with numerical tests in [EYH 96b] and in Appendix C. Notice that in the derivation, a boundary term appears due to integration by parts.

For the second derivation, the point of departure is the expression of the potential energy; the originality of this formulation is the introduction of the concept of functional. A new object **VariationalPrinciple** is then created; its behavior is linked to the minimization of the functional. This is described in [EYH 97b].

3.2.2 A mixed formulation for Stokes flow problem

3.2.2.1 Mathematical formulation

The present formulation is a mixed formulation of compressible media capable of representing the incompressible limit. Theoretical problems attached to this formulation and notation definitions can be found in [HUG 87].

The equations chosen for the Stokes flow problem are :

Given *f*, *g* and *F* , find (*u* , *p*), the velocity and the pressure, with appropriate conditions of continuity on domain Ω ($\Omega \subset \Re^{n_{sd}}$, where n_{sd} is the space dimension) such that :

$$\sigma_{ij,j}+f_i=0 \quad \text{in } \Omega$$

$$u_{i,i}+\frac{p}{\lambda}=0 \quad \text{in } \Omega$$

with the following boundary conditions :

$$\sigma_{ij}n_j=F_i \quad \text{on } \Gamma_{F_i}$$

$$u_i=g_i \quad \text{on } \Gamma_{g_i}$$

and a constitutive law :

$$\sigma_{ij}=-p\delta_{ij}+2\mu\varepsilon_{kl}(u)$$

In the nearly incompressible case, λ is taken large with respect to μ ($10^7 \leq \frac{\lambda}{\mu} \leq 10^9$ which is a good approximation with a computer precision of 10^{-15}).

Let us define $\mathcal{W}$ and $\mathcal{S}$, respectively as the spaces of weighting and trial solution for velocity, and $\mathcal{P}$ as a space of pressure (functions in $\mathcal{W}$ are H_1 and zero on the boundary of the domain, functions in $\mathcal{S}$ are H_1, functions in $\mathcal{P}$ are L_2).

A variational formulation equivalent to the preceding strong form is :

Given *f*, find $(u,p)\in\mathcal{S}\times\mathcal{P}$ such that for all $(w,q)\in\mathcal{W}\times\mathcal{P}$:

$$\int_\Omega(\sigma_{ij,j}+f_i)w_i dv+\int_\Omega(u_{i,i}+\frac{p}{\lambda})qdv=0$$

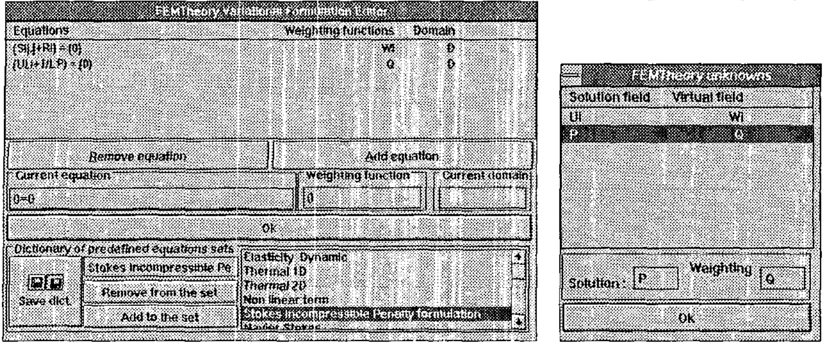
where $\sigma_{ij}=-p\delta_{ij}+2\mu\varepsilon_{kl}(u)$

The derivation of the finite element formulation is classical; the operations are described step by step in the FEM_Theory environment.

Remark: the same formulation is used in structural mechanics to model a linear elastic incompressible media.

3.2.2.2 Derivation in FEM_Theory :

The variational formulation defined above is introduced through window (a) shown in Figure 45: «Sij,j» is the divergence operator applied to the stress tensor, «Ri» represents the body loads, «Ui,i» is the divergence operator applied to velocity vector «Ui», «L» (lambda) is a constitutive parameter, «P» is the pressure field, «Wi» is the virtual velocity, «Q» is the virtual pressure. Window (a) gives access to a dictionary of predefined sets of equations. On window (b) the solution and corresponding weighting fields are defined. The variational principle is then posted on line 1 of the FEM_Theory window shown in Figure 46.



Equations	Weighting functions	Domain
(Sij,j+Ri) = (0)	Wi	D
(Ui,i+1/L P) = (0)	Q	D

Current equation: 0=0

Weighting function: 0

Current domain:

Dictionary of predefined equations sets: Stokes incompressible Pe

Elasticity Dynamic
Thermal 1D
Thermal 2D
Non linear term
Stokes incompressible Penalty formulation
Navier Stokes

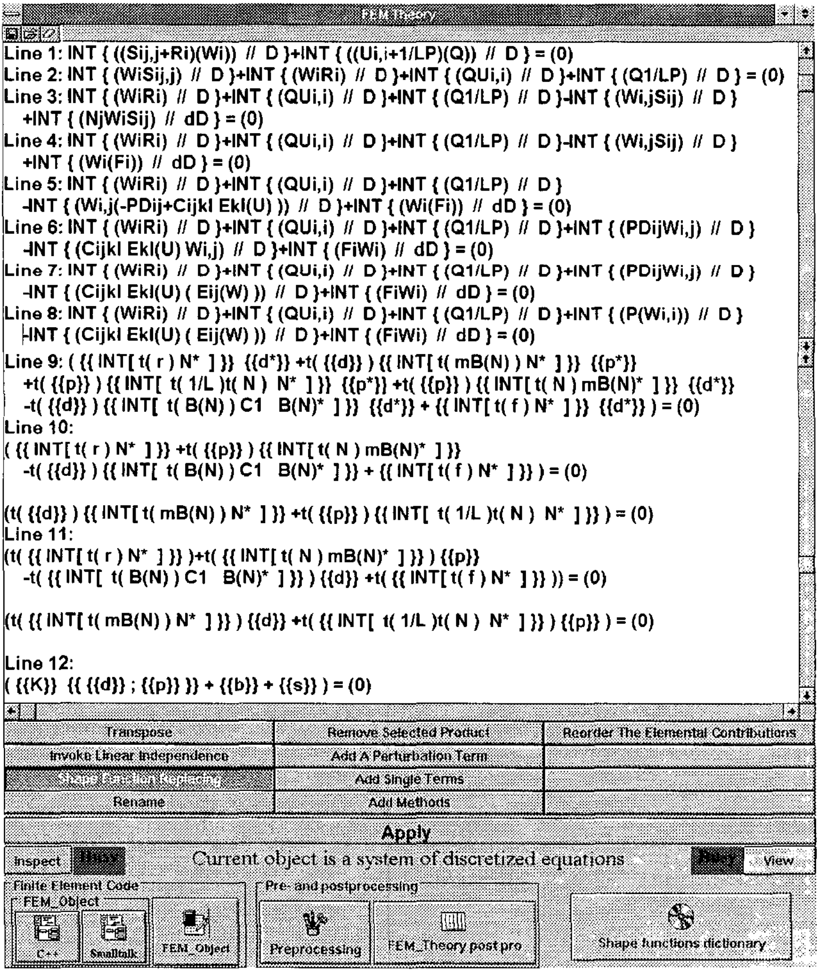
Solution field	Virtual field
Ui	Wi
P	Q

Solution : P Weighting Q

(a) Definition of the variational formulation (b) Definition of the unknowns

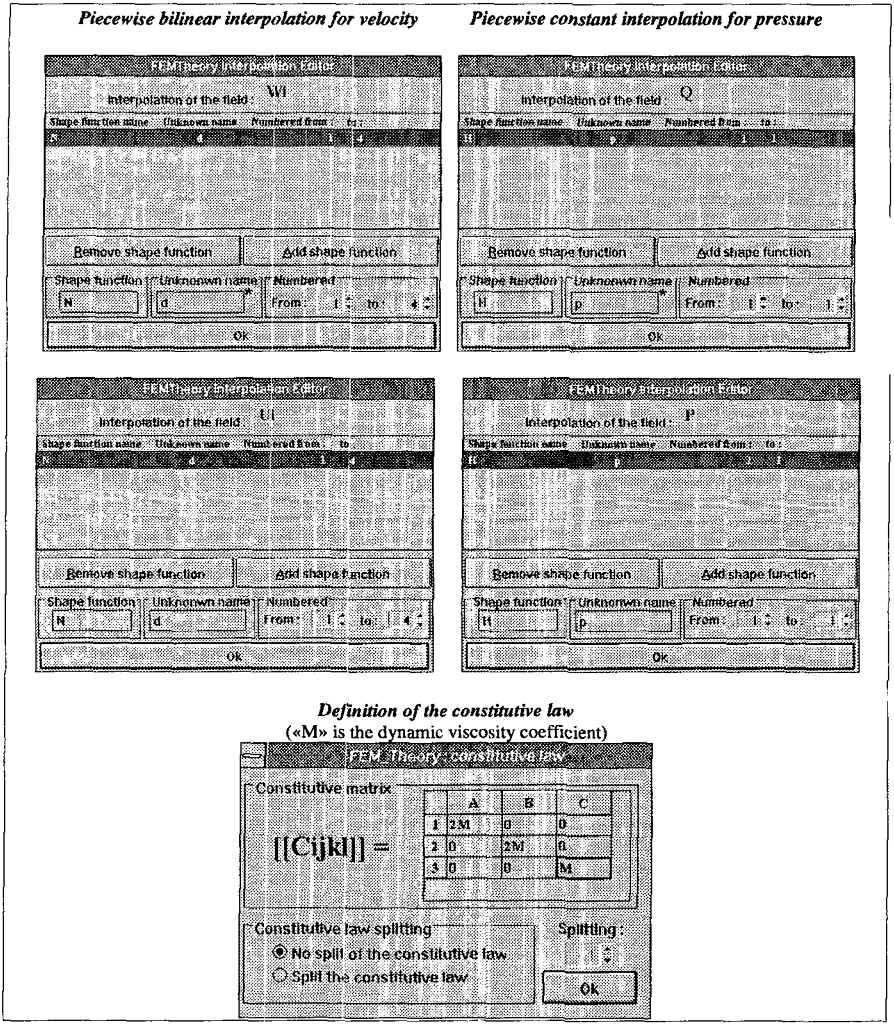
Figure 45 Windows permitting to introduce the Stokes problem in FEMTheory

On line 2, the variational formulation of line 1 is expanded. On line 3, the integral «INT{(WiSij,j) // D }» is integrated by parts. On line 4, the boundary term «SijNj» has been replaced by «Fi» (Neumann boundary condition). On line 5, the constitutive law is used to replace the stress tensor «Sij» with «-PDij+CijklEkl(U)» ; «Dij» represents the Kroneker symbol δ_{ij}, «Cijkl» is the constitutive law, «Ekl(U)» is the symetric part of the gradient tensor of vector «U». On line 6, the preceding line is expanded. As «CijklEkl(U)» is symmetric, «Wi,j» is replaced in line 7 with the symmetric part of gradient tensor applied to «W», «Eij(W)». On line 8, the indices of the product «DijWi,j» are contracted in «Wi,i». This is the weak form of the problem. This form is approximated through the use of a finite elements technique. Information to obtain the discretized form of the weak from is asked of the user. They are grouped in Figure 47. A bilinear interpolation for velocity and a constant interpolation for pressure are chosen on a quad element (Q1/Q0 element), «M» is the dynamic viscosity coefficient. Notice that the space dimension is 2. As the equation on line 9 is true for every «d*» and for every «p*», the two equations of line 10 can be deduced, the result is a system of discretized equations. The equations of this system are then transposed, the result is shown on line 11. On line 12, the system is arranged to get a system of one equation; the change of notation is obvious (see remark 2 in section 3.1.3, in class **Expression**) and is shown in Figure 48. At this point, some information about the numerical integration scheme has to be given for each elemental matrix of the formulation : a two by two points Gauss integration quadrature is chosen for all matrices. The theoretical justification can be found for example in [MAL 78]. Note that information about the numerical integration is used only during the generation of the code. The shape functions are then defined, in the window shown in Figure 49: functions «N» numbered from 1 to 4 are replaced with classical bilinear shape functions, function «H» (only one item) is replaced with a constant shape function. The code corresponding to this formulation is then introduced in the C++ version of numerical code FEM_Object; this is necessary to get enough numerical efficiency for subsequent tests. During the code generation some additional information is requested from the user, such as the constants used in the problem (here «L» a constitutive parameter and «M» the dynamic viscosity, the names "lambda" and "mu" allow instantiation of the constants in the data file). The numerical test is briefly discussed bellow.



```
Line 1: INT { ((Sij,j+Ri)(Wi)) // D }+INT { ((Ui,i+1/LP)(Q)) // D } = (0)
Line 2: INT { (WiSij,j) // D }+INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D } = (0)
Line 3: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }-INT { (Wi,jSij) // D }
  +INT { (NjWiSij) // dD } = (0)
Line 4: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }-INT { (Wi,jSij) // D }
  +INT { (Wi(Fi)) // dD } = (0)
Line 5: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }
  -INT { (Wi,j(-PDij+Cijkl Ekl(U) )) // D }+INT { (Wi(Fi)) // dD } = (0)
Line 6: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }+INT { (PDijWi,j) // D }
  -INT { (Cijkl Ekl(U) Wi,j) // D }+INT { (FiWi) // dD } = (0)
Line 7: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }+INT { (PDijWi,j) // D }
  -INT { (Cijkl Ekl(U) ( Eij(W) )) // D }+INT { (FiWi) // dD } = (0)
Line 8: INT { (WiRi) // D }+INT { (QUi,i) // D }+INT { (Q1/LP) // D }+INT { (P(Wi,i)) // D }
  -INT { (Cijkl Ekl(U) ( Eij(W) )) // D }+INT { (FiWi) // dD } = (0)
Line 9: ( {{ INT[ t( r ) N*  ] }}  {{d*}} +t( {{d}} ) {{ INT[ t( mB(N) ) N*  ] }}  {{p*}}
  +t( {{p}} ) {{ INT[  t( 1/L )t( N )  N*  ] }}  {{p*}} +t( {{p}} ) {{ INT[ t( N ) mB(N)*  ] }}  {{d*}}
  -t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ] }}  {{d*}} + {{ INT[ t( f ) N*  ] }}  {{d*}} ) = (0)
Line 10:
( {{ INT[ t( r ) N*  ] }} +t( {{p}} ) {{ INT[ t( N ) mB(N)*  ] }}
  -t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ] }} + {{ INT[ t( f ) N*  ] }} ) = (0)

(t( {{d}} ) {{ INT[ t( mB(N) ) N*  ] }} +t( {{p}} ) {{ INT[  t( 1/L )t( N )  N*  ] }} ) = (0)
Line 11:
(t( {{ INT[ t( r ) N*  ] }} )+t( {{ INT[ t( N ) mB(N)*  ] }} ) {{p}}
  -t( {{ INT[  t( B(N) ) C1   B(N)*  ] }} ) {{d}} +t( {{ INT[ t( f ) N*  ] }} )) = (0)

(t( {{ INT[ t( mB(N) ) N*  ] }} ) {{d}} +t( {{ INT[  t( 1/L )t( N )  N*  ] }} ) {{p}} ) = (0)

Line 12:
( {{K}}  {{ {{d}} ; {{p}} }} + {{b}} + {{s}} ) = (0)
```

Figure 46 Derivation of the Stokes flow problem



Piecewise bilinear interpolation for velocity

FEMTheory Interpolation Editor

Interpolation of the field : Wi

Shape function name	Unknown name	Numbered from :	to :
N	d	1	4

Shape function: N
Unknonwn name: d
Numbered From: 1 to: 4

FEMTheory Interpolation Editor

Interpolation of the field : Ui

Shape function name	Unknown name	Numbered from :	to :
N	d	1	4

Shape function: N
Unknonwn name: d
Numbered From: 1 to: 4

Piecewise constant interpolation for pressure

FEMTheory Interpolation Editor

Interpolation of the field : Q

Shape function name	Unknown name	Numbered from :	to :
H	p	1	1

Shape function: H
Unknonwn name: p
Numbered From: 1 to: 1

FEMTheory Interpolation Editor

Interpolation of the field : P

Shape function name	Unknown name	Numbered from :	to :
H	p	1	1

Shape function: H
Unknonwn name: p
Numbered From: 1 to: 1

Definition of the constitutive law

(«M» is the dynamic viscosity coefficient)

FEM Theory - constitutive law

Constitutive matrix

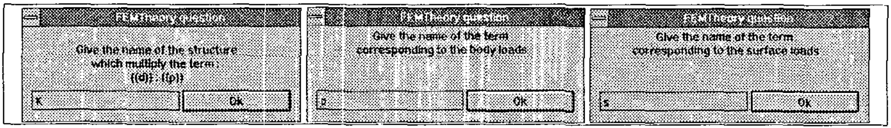
[[Cijkl]] =

	A	B	C
1	2M	0	0
2	0	2M	0
3	0	0	M

Constitutive law splitting
● No split of the constitutive law
○ Split the constitutive law

Splitting : 1

Figure 47 Approximation of velocity and pressure for Stokes flow



FEMTheory question

Give the name of the structure which multiply the term :
{(d)} ; {(p)}

K

FEMTheory question

Give the name of the term corresponding to the body loads

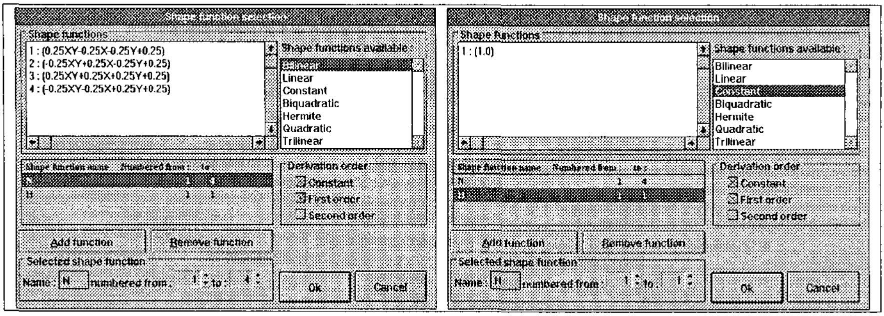
b

FEMTheory question

Give the name of the term corresponding to the surface loads

s

Figure 48 Change of notation in the assemblage procedure for Stokes flow



Shape function selection

Shape functions
1 : (0.25XY-0.25X-0.25Y+0.25)
2 : (-0.25XY+0.25X-0.25Y+0.25)
3 : (0.25XY+0.25X+0.25Y+0.25)
4 : (-0.25XY-0.25X+0.25Y+0.25)

Shape functions available : Bilinear, Linear, Constant, Biquadratic, Hermite, Quadratic, Trilinear

Shape function name	Numbered from :	to :
N	1	4
H	1	1

Derivation order
☒ Constant
☒ First order
□ Second order

Selected shape function
Name : N numbered from : 1 to : 4

Shape function selection

Shape functions
1 : (1.0)

Shape functions available : Bilinear, Linear, Constant, Biquadratic, Hermite, Quadratic, Trilinear

Shape function name	Numbered from :	to :
N	1	4
H	1	1

Derivation order
☒ Constant
☒ First order
□ Second order

Selected shape function
Name : H numbered from : 1 to : 1

Figure 49 Selection of the shape functions for Stokes flow

3.2.2.3 Numerical results

The new element is tested on the cavity flow problem. The description of the problem is given in Figure 50.

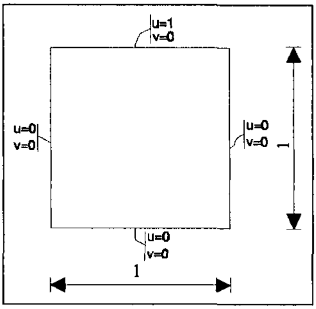


Figure 50 Description of the cavity flow problem

The constitutive parameters taken are : $\mu = 1$ and $\lambda = 10^7$.

This Q1/Q0 element is tested on a 12*12 mesh. The velocity field and pressure contours are shown in Figure 51. The pressure is post-processed to draw the contour without using smoothing techniques. As known, this element gives good results for velocity, but has poor performances for pressure evaluation, as the pressure solution shows a checkerboard phenomenon.

This formulation is to be compared with another one proposed by Franca in [FRA 87] and derived in Chapter 5. This type of comparison illustrate the fast prototyping capability of the environment and its usefulness.

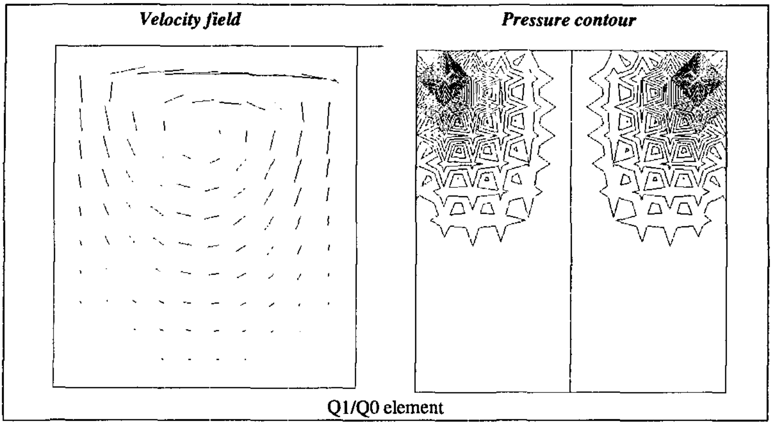


Figure 51 Numerical results for a penalty formulation of Stokes flow for the cavity problem

Chapter 4 Computer Aided Software Engineering for finite elements developments

4.1 Dimensional analysis in an object-oriented environment for finite elements

4.1.1 The concepts for dimensional analysis

The point of departure of this study is the international system of units (ISO). Take the example of [NF X 02-051] (French norms AFNOR) or [SNV 0121100] (Swiss normalization SNV). From the normalization (see Table 10) it can be deduced that each magnitude has a unit that can be expressed by means of 7 basis units shown in Table 9. The definition of the unit can be completed by the use of a factor and a prefix, e.g. 1 ft=0.3048 m where the factor is 0.3048, and 1 kN=10^3N (prefix k). All the units can be expressed in this way. The aim of this part is to build structures to represent the units, capable of conversion and analysis.

Table 9 Basis units of International System

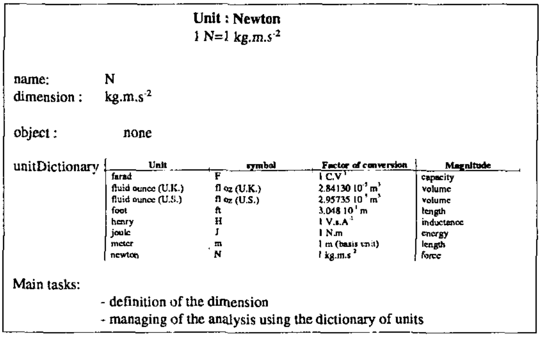
Magnitude		Unit basis	
name	symbol	name	symbol
Length	l	meter	m
Mass	m	kilogram	kg
Time	t	second	s
Intensity of current	I	ampere	A
Thermodynamic temperature	T	kelvin	K
Quantity of material	n	mole	mol
Luminosity intensity	I_v	cadela	cd

Table 10 Example of dimension of units (from NF X 02-051)

Unit	symbol	Factor of conversion	Magnitude
farad	F	1 C.V^{-1}	capacity
fluid ounce (U.K.)	fl oz (U.K.)	2.84130 10^{-5} m^3	volume
fluid ounce (U.S.)	fl oz (U.S.)	2.95735 10^{-5} m^3	volume
foot	ft	3.048 10^{-1} m	length
henry	H	1 V.s.A^{-1}	inductance
joule	J	1 N.m	energy
meter	m	1 m (basis unit)	length
newton	N	1 kg.m.s^{-2}	force

4.1.2 The objects for dimensional analysis

In FEM_Theory, the unit, the basic object to be associated to a magnitude, and the behavior for dimensional analysis are inherited from class **StructureWithDimension** (see chapter 2), e.g. for terms, expressions, integrals,... This object is illustrated in Figure 52 on the example of Newton. The object has a name, a dimension, and its definition can be completed by a prefix and a factor of conversion. For the sake of simplicity the factor and prefix are not taken into account. The unit can have access to a data base where it could find all the data needed for conversion (similar to Table 10). At least, the unit can be associated to an object. So, the class **Unit** is defined.



Unit : Newton

1 N=1 kg.m.s^{-2}

name: N

dimension : kg.m.s^{-2}

object : none

unitDictionary

Unit	symbol	Factor of conversion	Magnitude
farad	F	1 C.V^{-1}	capacity
fluid ounce (U.K.)	fl oz (U.K.)	2.84130 10^{-5} m^{3}	volume
fluid ounce (U.S.)	fl oz (U.S.)	2.95735 10^{-5} m^{3}	volume
foot	ft	3.048 10^{-1} m	length
henry	H	1 V.s.A^{-1}	inductence
joule	J	1 N.m	energy
meter	m	1 m (basis unit)	length
newton	N	1 kg.m.s^{-2}	force

Main tasks:

- definition of the dimension
- managing of the analysis using the dictionary of units

Figure 52 Definition of the object unit on the example of Newton

The main component of the object unit is its dimension (instance of class **Dimension**). The goal is to build a structure capable of giving a representation of the dimension based on Table 9. This is illustrated in Figure 53. The idea is to use an existing structure of Smalltalk, the dictionary (see [SMA 93] and chapter 3). The key to get access to the data stored in the dictionary is a symbol corresponding to the name of the magnitude. The data stored at the corresponding key is a signed integer which gives the power of the basis unit. In Figure 53, the power corresponding to the length (l) is 1, to the mass (m) is 1, to the time (t) is –2; the others are 0. The result for Newton is kg.m.s^{-2}. The main tasks of this object is to define itself, by asking information to the user for example, and to effectuate combinations of dimensions in products. The best is to illustrate it in an example. Consider the product $P = m \cdot g$ where m is the mass expressed in kg (kilogram), g the acceleration of gravity expressed in m.s^{-2} and P the weight. The result P is then expressed in kg.m.s^{-2}. This is found as follows. Represent the dimension of a magnitude by the notation [ ]. So, the dimension of P, [P] is obtained by multiplying the dimension of m by the one of g : $[P] = [m] * [g]$. The "product" which makes it possible to obtain this dimension is sketched in Figure 54. The final dimension of P is obtained by simply adding the indices corresponding to the basis units. Thus, this object has the possibility to be multiplied or divided by another object dimension. A basic algebra is defined at the level of the object dimension.

The last object needed for dimensional analysis is a dictionary to store the units that can be seen e.g. in Table 10. A simple dictionary object in Smalltalk could be used here, but a specialization scheme is needed to look up units in the dictionary. This object is an instance of **FEMTheoryGeneralDictionaryOfUnits**.

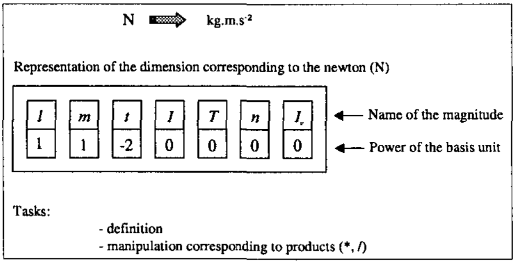

N
kg.m.s-2
Representation of the dimension corresponding to the newton (N)
l
m
t
I
T
n
I_v
1
1
-2
0
0
0
0
Name of the magnitude
Power of the basis unit
Tasks:
- definition
- manipulation corresponding to products (*, /)

Figure 53 Example of dimension

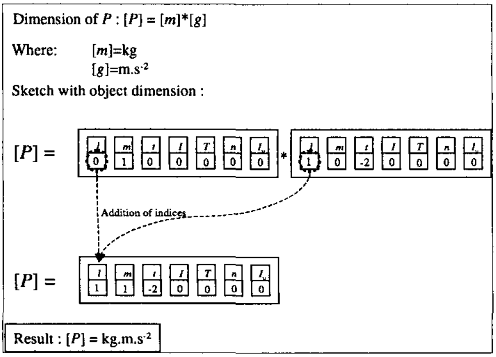


Figure 54 Sketch for the dimension of $P = m \cdot g$

4.1.3 The classes

Class **Dimension**

The class **Dimension** presented on Table 11 is a dictionary of size 7, which corresponds to the number of basic units, i.e. length (symbol l), mass (symbol m), time (symbol t), intensity of current (symbol I), thermodynamic temperature (symbol T), material quantity (symbol n) and luminosity intensity (symbol Iv) –see [AFNOR X02-051] or [SNV 012100] for more details–. Each of these symbols gives access in the dictionary to an integer which represents the power of the corresponding basis unit. The behavior of the classes consists first in defining the dimension, and second in effectuating basic algebra manipulations on it.

Table 11 Class **Dimension**

Class Dimension **Inherits from : Dictionary, FEMTheoryDictionaries,..., Object**		
Inherited tasks	**Inherited attributes**	**Inherited methods**
-	-	*- all the methods for dictionaries*
Tasks	**Attributes**	**Methods**
1) Definition	-	answerYourselfFor: anObj asArray atAllPut: anInteger define defineFor: obj getBasicUnits giveBasicUnitsArray isDefined isNotDefined
2) Algebra		* aDimension / aDimension inverse power: anInt = aDict

Class **Unit**

The class **Unit** (see Table 12) has five attributes. The first four make it possible to define the unit: the attribute **dimension** which is an instance of **Dimension** is completed by the attribute **name**, instance of **Symbol**. The unit can be the unit of a data, a term, a product,... ; it is the attribute **object**. At least, the unit may need information to complete its definition from data stored in the dictionary of units, attribute **unitsDictionary**.

The first part of the behavior is linked to the complete definition of the unit, i.e. its symbol, attribute **name**, its dimension, and perhaps an object (term, expression,...) to which the unit is associated. The second part of the behavior is the management of the data contained in the dictionary of units.

Remark 1: The definition of the dimension by the user is decentralized to the attribute *dimension* itself (access to the prompter Figure 58).

Remark 2: To give a complete definition of all types of units two attributes could be added here. The first one is needed to represent the prefix of the unit (see [NF 02-051]), e.g. prefix 'k' for 'kilo' corresponding to 10^3. This new attribute *prefix* could be an instance of a new class **Prefix** similar to the class **Unit**, but managing the prefixes. A second attribute, call it *factor*, a float instance of **Float**, is needed to complete the conversion between the units of the international system and the others (e.g. the darcy, the gallon, the foot ...)

Table 12 Class **Unit**

Class Unit **Inherits from : FEMTheory, Object**		
Inherited tasks	**Inherited attributes**	**Inherited methods**
-	-	-
Tasks	**Attributes**	**Methods**
1) Definition	- dimension - name - object	define defineDimension dimension: aDim isDefined isUnit name: aSymbol object: anObj prefix: aSymbol
2) Manipulations	- unitDictionary	getBasicUnits getDimension giveDimension giveName giveObject givePrefix initUnitDictionary

Class FEMTheoryGeneralDictionaryOfUnits

This class presented on Table 13 is used only to store the different units that can be used in **FEMTheory** and behaves as a classical Smalltalk dictionary (see [VIS 95b]). Only one instance appears during execution; this instance is stored on disk, and recovered whenever needed, using the tool class **ObjectFiler** (see [VIS 95b]).

The key used to store the objects of type **Unit** is a symbol that is the name of the unit. For example the unit 'Joule' corresponding to work or energy, has as symbol *J* and dimension $kg.m^2.s^{-2}$. All the behavior of the class is inherited from **Dictionary**. Only one special method is added to get a unit from the definition of its dimension. This method make it possible to make a loop on the values of the dictionary to get the key, i.e. from the definition of the dimension $kg.m^2.s^{-2}$, to get the unit *J*.

Table 13 Class **FEMTheoryGeneralDictionaryOfUnits**

Class FEMTheoryGeneralDictionaryOfUnits **Inherits from : FEMTheoryDictionaries, Dictionary, ..., Object**		
Inherited tasks	**Inherited attributes**	**Inherited methods**
-	-	-
Tasks	**Attributes**	**Methods**
Manipulations		findUnitOfDimension: aDim

Class **StructureWithDimension** and subclasses

The class **StructureWithDimension** (see Table 14) regroups the behavior common to subclasses needed for representing the variational formulation (see the general hierarchy of classes of FEM_Theory, chapter 2). The only attribute of the class is called **unit** and becomes an instance of class **Unit**. The only class level behavior is linked to the management of the unit, i.e. its definition and the procedure to check the consistency of units in a variational formulation. This scheme is described in the next section.

Table 14 Class **StructureWithDimension**

Class StructureWithDimension		
Inherits from : FEMTheory, FEMTheoryMathematicalStructures, Object		
Inherited tasks	**Inherited attributes**	**Inherited methods**
-	- hierarchicParent	- "management of the attribute *hierarchicParent*"
Tasks	**Attributes**	**Methods**
Managing of the unit	unit	addDimensionCharacteristicsTo: col forObject: anObj deduceDimension findDimensionBackwards giveUnit updateDimensionForTerm: aTerm

4.1.4 Strategy for dimensional analysis in FEMTheory

The scheme for dimensional analysis is based on the data structure presented in chapter 2. The simple algorithm described here implicates the class **StructureWithDimension** and subclasses (**Term**, **Expression**, **IntEquation**, ...). The problem is to deduce the dimension of an object within a complex expression, just by giving the dimension of some terms. The purpose is not to give a general algorithm for the problem, but rather to give an overview of the possibilities of such a tool. The principle of the algorithm is sketched in Figure 55 on the equation $\int_{\Omega} (\sigma_{ij,j} + f_i)\ w_i\ dv = 0$ taken from the linear elasticity (see Appendix B). This scheme is described to find the dimension of the object integral $\int_{\Omega} (\sigma_{ij,j} + f_i)\ w_i\ dv$, but could be applied to any objects :

- on the screen, the object integral is selected
- the tool '*Find Dimension For Term*' is selected and applied; consequently, the message *deduceDimensionSelection: integralString* is sent to the object **IntEquation**
- the message goes down the roots of the tree following the doted arrows in Figure 55, until the selected integral is recognized (highlighted in gray in the figure)
- when the integral is found, the message *deduceDimension* is sent to the integral itself
- in the method *deduceDimension*, a first search is done while descending the roots, sending successive messages *findDimensionForward*, shown by plain line arrows in the figure; the goal is to try to deduce the dimension of the object, just by deducing the dimension of the object composing it; this scheme is successful when one branch of a root at a sum level is defined
- in the method *deduceDimension*, if dimension is not found with a process descending the roots, an ascending process is started by the message *findDimensionBackward* (dashed line arrow in the figure), which has the task of sending the message

deduceDimensionForObject: integral; in this method either message *findDimensionForward* or *findDimensionBackward* or both are sent to try to deduce the dimension at the current node (recursive message passing)
- so, the messages goes down and up at each node, i.e. each object composing the tree of the equation; the process stops either when the dimension asked by the user is found, or when all the nodes of the tree have been tested

The methods enumerated here will be implemented differently for each object, but the scheme presented here for object 'integral' is the same for all objects. Notice that this scheme cannot solve all the situations. It is based on the assumption that each node can be solved locally. This is true for the most common situations, but the scheme fails when reasoning concepts at a global level are necessary. An example where local reasoning (at the object level) is enough is the index writing consistency check presented in section 4.2.

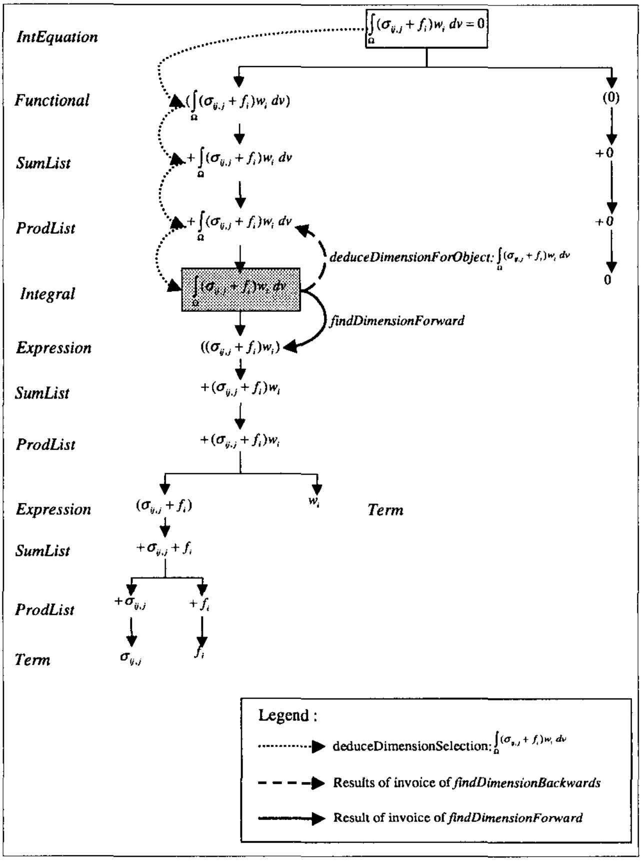
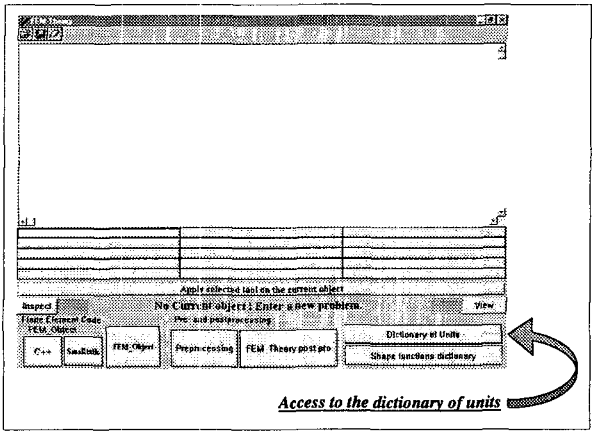

IntEquation
$\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv=0$
Functional
$(\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv)$
(0)
SumList
$+\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
+0
ProdList
$+\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
+0
deduceDimensionForObject: $\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
0
Integral
$\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
findDimensionForward
Expression
$((\sigma_{ij,j}+f_i)w_i)$
SumList
$+(\sigma_{ij,j}+f_i)w_i$
ProdList
$+(\sigma_{ij,j}+f_i)w_i$
Expression
$(\sigma_{ij,j}+f_i)$
w_i
Term
SumList
$+\sigma_{ij,j}+f_i$
ProdList
$+\sigma_{ij,j}$
$+f_i$
Term
$\sigma_{ij,j}$
f_i
Legend :
deduceDimensionSelection: $\int_\Omega (\sigma_{ij,j}+f_i)w_i\,dv$
Results of invoice of *findDimensionBackwards*
Result of invoice of *findDimensionForward*

Figure 55 Sketch for illustrating dimensional analysis strategy

4.1.5 The graphical environment for dimensional analysis

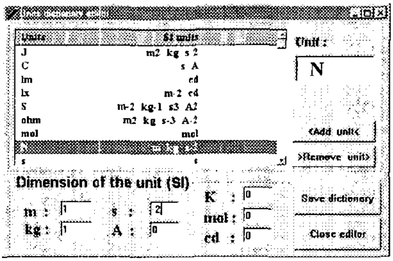
The object presented in the above sections can be visualized within the graphical environment presented in chapter 2. In the graphical environment of FEM_Theory a push-button is added (see in Figure 56) which launches an editor that can be seen in Figure 57. In this editor, the units contained in the dictionary, instance of **FEMTheoryGeneralDictionaryOfUnits** and stored on disk in file named 'fem.dct', can be viewed and new units can be added. The units are described in this editor by their name, and their dimension is given; e.g. the Newton (symbol N) is highlighted and its dimension is kg.m.s^{-2}. During a derivation, the dimension of a term can be defined by the user; this is done using the editor of Figure 58. The dimension of every object can also be visualized through the use of the prompter of Figure 59. The list of tools for the instances of class **IntEquation** is enriched with new tools: *'Define Dimension For Selected Term'*, *'Find Dimension For Selected Term'*, *'Check Dimension'*.

Remark: the dimension can be defined in terms of any units, including derived units.



Access to the dictionary of units

Figure 56 Main window of FEM_Theory with the management of the dictionary of units



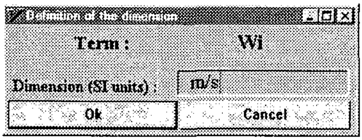
Units	SI units
J	m2 kg s 2
C	s A
lm	cd
lx	m-2 cd
S	m-2 kg-1 s3 A2
ohm	m2 kg s-3 A-2
mol	mol
N	[illegible]
s	s

Unit : N

Dimension of the unit (SI)

m : 1
kg : 1
s : 2
A : 0
K : 0
mol : 0
cd : 0

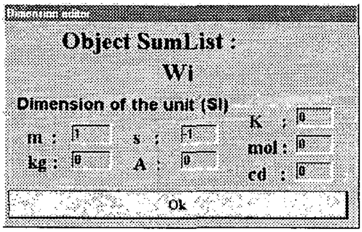
Figure 57 Units dictionary editor



Term : Wi

Dimension (SI units) : m/s

Figure 58 Prompter to define a dimension



Object SumList :
Wi

Dimension of the unit (SI)

m : 1 s : -1 K : 0
kg : 0 A : 0 mol : 0
cd : 0

Figure 59 Prompter to visualize a dimension

4.1.6 A simple illustration of dimensional analysis in FEM_Theory

The goal of this section is to give a trivial example of the usefulness of a dimensional analysis scheme in the symbolic environment. Take the example of the penalty formulation for Stokes flow used in chapter 3. The formulation is posted onto the screen of FEMTheory in Figure 60, line 1. The problem and the notations are defined in chapter 3. Let us define in the formulation on line 1 the dimensions of the terms that are obvious. This is done by selecting the term on the screen line 1, and applying to it the new tool *'Define Dimension For Selected Term'*. This tool gives access to the prompter shown in Figure 61. Here are the definitions of the following terms:

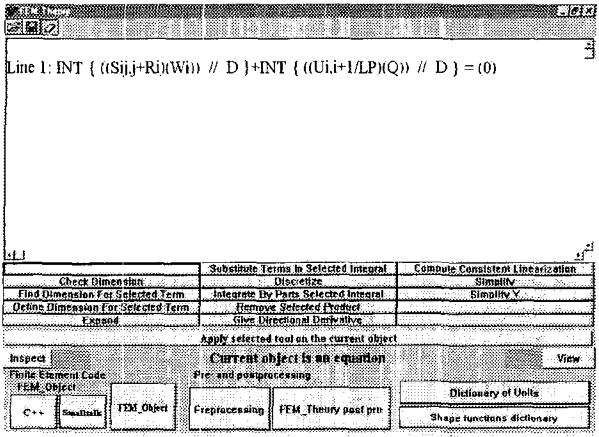
the weighting velocity w_i: $[w_i] = m.s^{-1}$

the pressure P: $[P] = N.m^{-2}$

the weighting pressure Q: $[Q] = N.m^{-2}$

the body loads (dimension given using its expression, i.e. the product between the density and the acceleration of gravity) R: $[R] = [\rho]\cdot[g] = (kg.m^{-3})\cdot(m.s^{-2})$

The dimension of all the entities composing this equation are now defined, and their dimension can be retrieved through the tool *'Find Dimension For Selected Term'*. The result is posted in prompters such as the ones of Figure 62, e.g. the dimension of the term $\frac{1}{\lambda}$ is: $\left[\frac{1}{\lambda}\right] = m.kg^{-1}.s$. The dimension of the various object of the equation are shown in Figure 62.

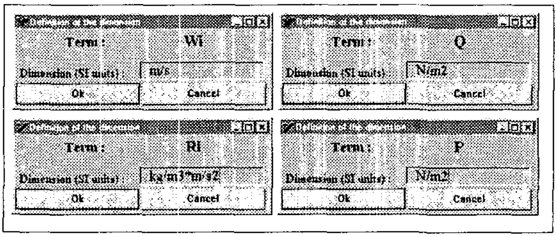


```
Line 1: INT { ((Sij,j+Ri)(Wi)) // D }+INT { ((Ui,i+1/LP)(Q)) // D } = (0)
```

Current object is an equation

Check Dimension
Find Dimension For Selected Term
Define Dimension For Selected Term
Expand
Substitute Terms In Selected Integral
Discretize
Integrate By Parts Selected Integral
Remove Selected Product
Give Directional Derivative
Compute Consistent Linearization
Simplify
Simplify Y
Apply selected tool on the current object
Inspect
View
Finite Element Code
FEM_Object
C++
Smalltalk
FEM_Object
Pre- and postprocessing
Preprocessing
FEM_Theory post pro
Dictionary of Units
Shape functions dictionary

Figure 60 Dimensional analysis of the penalty formulation for 2D Stokes problem



Term : Wi

Dimension (SI units) : m/s

Term : Q

Dimension (SI units) : N/m2

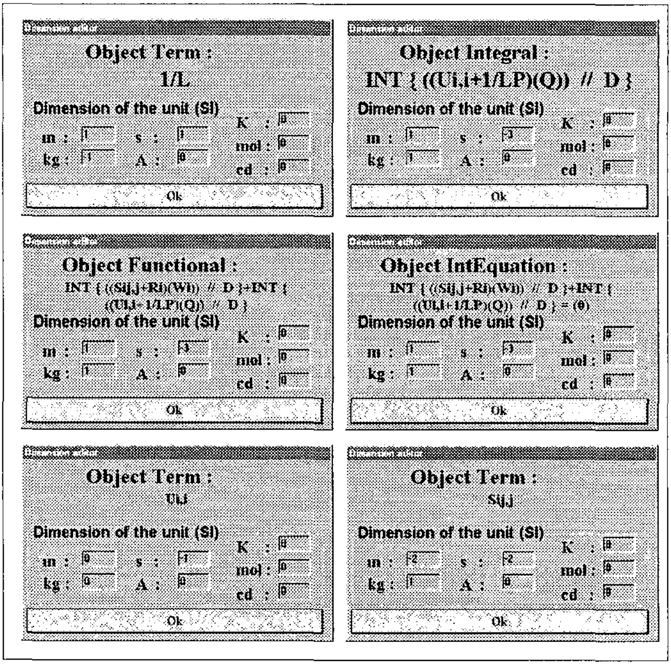
Term : Ri

Dimension (SI units) : kg/m3*m/s2

Term : P

Dimension (SI units) : N/m2

Figure 61 Definition of the dimensions for selected terms


Object Term :
1/L
Dimension of the unit (SI)
m : 1 s : 1 K : 0
kg : -1 A : 0 mol : 0
cd : 0

Object Integral :
INT { ((Ui,i+1/LP)(Q)) // D }
Dimension of the unit (SI)
m : 1 s : -3 K : 0
kg : 1 A : 0 mol : 0
cd : 0

Object Functional :
INT { ((Sij,j+Ri)(Wi)) // D }+INT {
((Ui,i+1/LP)(Q)) // D }
Dimension of the unit (SI)
m : 1 s : -3 K : 0
kg : 1 A : 0 mol : 0
cd : 0

Object IntEquation :
INT { ((Sij,j+Ri)(Wi)) // D }+INT {
((Ui,i+1/LP)(Q)) // D } = (0)
Dimension of the unit (SI)
m : 1 s : -3 K : 0
kg : 1 A : 0 mol : 0
cd : 0

Object Term :
Ui,i
Dimension of the unit (SI)
m : 0 s : -1 K : 0
kg : 0 A : 0 mol : 0
cd : 0

Object Term :
Sij,j
Dimension of the unit (SI)
m : -2 s : -2 K : 0
kg : 1 A : 0 mol : 0
cd : 0

Figure 62 Dimensional analysis of various objects

4.1.7 Dimension control in finite elements

The dimensional analysis process has been applied here in the context of the symbolic development of finite elements for trivial purposes. This ensues from the wish to develop concepts for finite elements with a high level of abstraction in the finite element derivation. The next step would be to use all the theoretical concepts developed and used during the symbolic derivation in the numerical computation. The control of the data introduced for a computation by the user is a crucial problem in numerical computation. The proposed approach could be extended to solving this problem. First, the three structures proposed in the previous sections can be used in any context, i.e. not only in a symbolic environment. Any type of structure can be given a characteristic 'unit'. From here, the control of dimensions could easily be done even during a numerical computation, by using a similar approach as the one presented in the previous section. A second extension would be to pass information about dimensional analysis from the symbolic environment to the numerical one, in which it could be used to check dimensions.

4.2 Checking index writing consistency

4.2.1 Goal

In FEM_Theory, the writing of the formulation is based on index notation. This notation is used for its general aspect. But mistakes in the notation are easy to make and can have disastrous consequences on the discretization process. The idea is to introduce a checking process for the consistency of the writing. This new tool does not need any new object, but only an enhancement of the classes involved in the representation of the variational formulation of the continuum problem. The process is described next.

4.2.2 Implementation of writing analysis

Contrary to the dimensional analysis process described in the previous section, the checking of the writing can be made at the local level, i.e. at the level of each object (see all the objects involved in the process in Figure 65). Thus, each object is able to recover the contracted indices characterizing itself. The implementation ensues naturally. Each object has a method called *checkIndicialNotation*. This method returns a string representing the indices of the receiver contracted, e.g. the object $\sigma_{ij,j}$ returns the string 'i' which is the contraction of the indices 'ijj' (rules for classical index notation). For all the objects, the structure of the method is the same :

(a) ask the objects composing it to check their index notation (message *checkIndicialNotation*); they return a string representing the indices contracted

(b) check the coherence of the indices at its level if necessary

(c) return the string representing the indices (contracted)

Two examples of implementation of this method are given in Figure 64 and in Figure 63, for objects integral (class **Integral**) and sum (class **SumList**); they respect the three points given previously. The message for checking the notation goes down the tree as illustrated in Figure 65. The process ends when each node of the tree has made this check.

```
checkIndicialNotation
   "Check if the indicial notation of the receiver is correct"
     | reply str |                                        Definition of the local variables

 reply := (self at:1) checkIndicialNotation.              Initialization of the string  with the reference of the first product
 self do:[:p|                                             Loop over the products composing the sum.
   str:= p checkIndicialNotation.                         (a) Ask its products to check their notation (result stored in str)
   (reply isAnAnagramOf: str)                             (b) Check if the notation of the current product is the same as the
                                                          one of the reference
       ifFalse:[
       FEMTheoryMessage openOnMessage:                    ... if not answer an error message
       ('Error in the index notation of :' ,(self printString)).
       ^nil
       ].
 ].

^reply                                                    (c) Returns the string representing the indices of the sum, which is
                                                          the same as each product composing it.
```

Figure 63 Method *checkIndicialNotation* in class **SumList**

```
checkIndicialNotation
   "Check if the notation of the receiver is coherent"

   ^ self giveIntegrand checkIndicialNotation        (a) Ask its integrand to check its notation and (c) return the
                                                     indices contracted.
                                                     Note that no part (b) for coherence control is needed here
```

Figure 64 Method *checkIndicialNotation* in class **Integral**

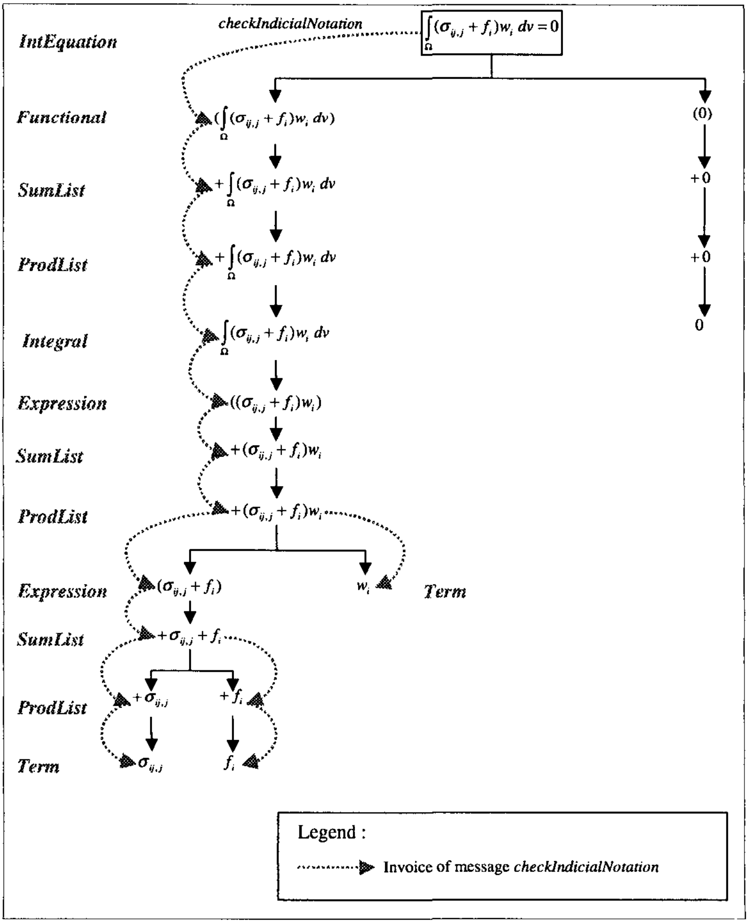
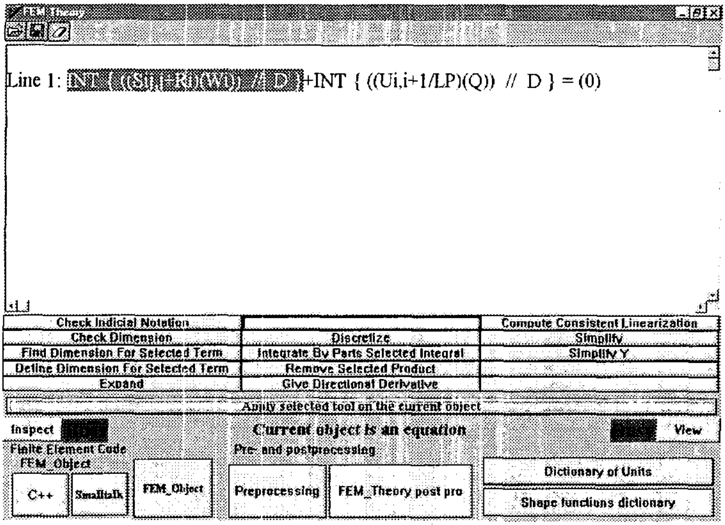

IntEquation
checkIndicialNotation
$\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv = 0$
Functional
$(\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv)$
(0)
SumList
$+\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv$
+0
ProdList
$+\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv$
+0
Integral
$\int_\Omega (\sigma_{ij,j} + f_i) w_i \, dv$
0
Expression
$((\sigma_{ij,j} + f_i) w_i)$
SumList
$+(\sigma_{ij,j} + f_i) w_i$
ProdList
$+(\sigma_{ij,j} + f_i) w_i$
Expression
$(\sigma_{ij,j} + f_i)$
w_i
Term
SumList
$+\sigma_{ij,j} + f_i$
ProdList
$+\sigma_{ij,j}$
$+f_i$
Term
$\sigma_{ij,j}$
f_i
Legend :
Invoice of message checkIndicialNotation

Figure 65 Sketch for the checking of the index notation

4.2.3 Example of analysis

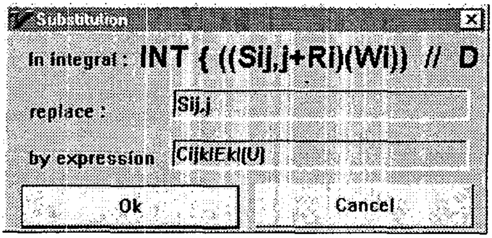
An illustration of the use of this scheme in FEM_Theory is shown in Figure 66; on line 1, the penalty formulation for Stokes of chapter 3 is posted. In the integral selected on line 1 (highlighted object on the screen), the prompter of Figure 67 makes it possible to replace the term $\sigma_{ij,j}$ by the expression $C_{ijkl}\varepsilon_{kl}(u)$, and instead of $C_{ijkl}\varepsilon_{kl,j}(u)$ as it should be done. But an error is introduced in the prompter (the index 'j' is left out). Then, the prompter of Figure 68 appears indicating that the expressions introduced are not correct.



Line 1: INT { ((Sij,j+Ri)(Wi)) // D }+INT { ((Ui,i+1/LP)(Q)) // D } = (0)

FEM_Theory
Check Indicial Notation
Check Dimension
Find Dimension For Selected Term
Define Dimension For Selected Term
Expand
Discretize
Integrate By Parts Selected Integral
Remove Selected Product
Give Directional Derivative
Compute Consistent Linearization
Simplify
Simplify Y
Apply selected tool on the current object
Inspect
Current object is an equation
View
Finite Element Code
FEM_Object
C++
Smalltalk
FEM_Object
Pre- and postprocessing
Preprocessing
FEM_Theory post pro
Dictionary of Units
Shape functions dictionary

Figure 66 Illustration of the writing consistency on a penalty formulation of the Stokes problem

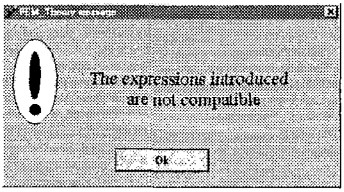


In integral : INT { ((Sij,j+Ri)(Wi)) // D

replace : Sij,j

by expression CijklEkl(U)

Figure 67 Prompter for replacing an expression with a notation error



The expressions introduced
are not compatible

Figure 68 Notification of the error in the notation

Chapter 5 Beyond a classical Galerkin formulation

5.1 Towards finite element computations of incompressible flows

In the previous chapters, an environment to handle basic linear variational formulations was presented. In this chapter, our aim consists in showing the extendibility capabilities of the environment to "real life" problems, i.e. formulations subject to active research. We will focus our attention on modern approaches to solve numerically the incompressible Navier-Stokes equations, which still remain today a challenging problem. Interesting issues about different formulations of incompressible flows are discussed in [GRE 91, FLE 91a and b]. We will focus here on velocity-pressure formulations. Two major issues for the solution of this type of problem will be investigated.

First, as it is well known, velocity-pressure Galerkin formulations for Navier-Stokes equations exhibit two types of instability. One comes from the presence of the advective term and becomes dominant at high Reynolds. The other source of instability is due to the mixed character of the velocity-pressure formulation, i.e. more precisely to inappropriate combinations of interpolation functions for the representation of velocity and pressure fields. To remedy these spurious oscillations, we will investigate the Galerkin Least-squares type stabilized formulations, such as the well known SUPG presented in [BRO 82] and the various stabilization schemes studied in [FRA 89, FRA 92b, TEZ 92d].

Second, another challenging problem occurring in computational fluid dynamics is the computation of interfaces and moving boundaries. In [TEZ 92b, TEZ 92c, HAN 92a and HAN 92b], an original application of space-time formulations is done in the computation of moving boundaries.

Keeping in mind our first aim which is the solution of incompressible flows, the enhancement of the environment FEM_Theory to handle Galerkin Least-squares type stabilized formulations is discussed in section 5.2. In section 5.3, the enhancement of FEM_Theory to space-time formulations is discussed. In both cases, the development is restricted to linear formulations. The extension to nonlinear formulations will be done in Chapter 6.

5.2 Galerkin Least-Squares type stabilized methods

5.2.1 Brief review and objective

The mixed formulation for Stokes problem presented in Chapter 3 exhibits spurious oscillations in the pressure field (Q1/Q0 element). In the particular numerical computation of the cavity flow problem, these oscillations are not transmitted to the velocity field, which is not always the case. These oscillations are the consequence of an inappropriate combination of interpolation functions for velocity and pressure fields. Galerkin Least-Squares type methods are widely used in the Computational Fluid Dynamics community to cure this problem. These methods have also the advantage of curing oscillations emanating from the advection terms present in the Navier-Stokes formulation. In this context, the SUPG (Streamline-upwind/Petrov-Galerkin) method, initially formalized in [BRO 82], was the first to be intensively used in both incompressible and later compressible flows. This method consists in adding artificial numerical diffusion in the direction of the streamlines. The SUPG formulation is obtained by adding to the classical Galerkin formulation, least-squares type terms defined over each element. The terms added here correspond to the product of the residual of the momentum equation by the advective operator acting on the weighting function. A generalization of this method, called GLS (Galerkin Least-squares) was introduced first for the Stokes problem in [HUG 86, FRA 87, FRA 88] and in [HUG 89] for advective diffusive systems. In this method the terms added to the original Galerkin formulation are built by integrating over each element the product of the residual of the momentum equation by the corresponding differential operator acting on the weighting functions. Since then, many authors have presented slightly different forms of this formulation. A few methods are reviewed in [FRA 92a, FRA 92b and FRA 93b]. Among them, note a method attributed to Douglas and Wang in [FRA 92b] in which the Galerkin least-squares terms added to the Galerkin formulation are slightly modified by changing signs in the differential operator. A simplified version of the GLS method is presented in [TEZ 92a and TEZ 92d] for Navier-Stokes flows. This is a combination of the SUPG method, which takes into account the advective operator in the least-squares terms, and of the PSPG (Pressure stabilization Petrov-Galerkin) method in which the pressure part of the differential operator is taken into account. An interesting discussion about convenient choices of interpolation functions can be found in [TEZ 92d], and a comparative study between GLS and SUPG formulations for incompressible Navier-Stokes solutions is given in [HAN 95].

The design of the stabilization parameter, which is a crucial ingredient of these formulations, consists in adjusting the numerical diffusion rate added at the elemental level. Roughly speaking most of the stabilization parameter designs are based on element size and on flow regime (depending of the Reynolds number). In most cases the computation of this numerical parameter is a tough work including the choice of constants and element size parameter (see e.g. [HAR 92] for various examples). Some authors have proposed simpler versions for the stabilization parameter either for Stokes or Navier-Stokes flow. For example in [TEZ 92d], the stabilization parameter which is a scalar, is a multidimensional extension of a one-dimensional design presented in [SHA 88] for a scalar advection-diffusion equation. This design has the advantage of being simple and respects the advective and diffusive limits condition; as a matter of fact, numerical computations are expected to be efficient. In [FRA 93a], the authors present a design of stabilization parameter which makes it possible to overcome the drawbacks of the computation of inverse estimates and the element size dependence.

It is worth noticing that in the same time several authors have shown an equivalence between GLS and SUPG type methods, and Galerkin methods employing interpolations enriched with bubble functions for advection-diffusion models and Navier-Stokes equations (see e.g. [BRE 92a, BAI 93, FRA 94a and b, FRA 95, RUS 96] and references therein).

These formulations are widely used today for the solution of compressible or incompressible Navier-Stokes flows. They have found natural extensions in various domains of computational mechanics, such as e.g. for beam problems [FRA 87, LOU 87a], for plate problems [HUG 88b], for arch problems [LOU 87b], plasticity problems [PAS 97, TRU 97], and they have been extended to velocity-pressure-stress formulations for the incompressible Navier-Stokes equations [BEH 93]. A last interesting feature is a generalization of these stabilized methods when classical GLS methods fail. An example is given in [FRA 89] for a singular diffusion problem.

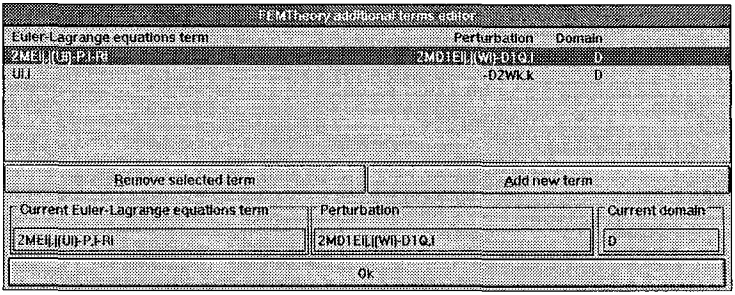
The objective is to be capable to handle these formulations in the symbolic environment FEM_Theory. This is shown in the following sections where simple examples of derivations are given. They have to be completed by the ones of Appendices B and D.

5.2.2 Integration of stabilized formulati ons in FEMTheory

The environment described in Chapter 3 makes it possible to deal with this type of formulation directly, i.e. without any new implementation. This is due to the fact that the algorithm used for the determination of elemental contributions for discretization and approximation procedures is based on the construction of the contribution of each term individually. The only feature added at this stage is a new window, shown in Figure 69 (see Chapter 4 for notations used here for a formulation of Stokes flow), accessed by push button *"Add perturbation term"*. In this window, two equations are taken into account, i.e. «2MEij,j(Ui)-P,i-Ri» (from the momentum equation for Stokes flow problem) and «Ui,i» (from the incompressibility condition for Stokes flow problem) respectively weighted by «2MD1Eij,j(Wi)-D1Q,i» and «D2Wk,k». This is the representation of the following stabilizing Galerkin-Least squares terms :

$$\sum_{\Omega^e \in \Omega^h} \left[\int_{\Omega^e} (2\mu\varepsilon_{ij,j}(u^h) - p^h_{i,i} - f_i) D_1 (2\mu\varepsilon_{ij,j}(w^h) - q^h_{i,i}) dv - \int_{\Omega^e} u^h_{i,i} D_2 w^h_{i,i} dv \right]$$

This will be used at length in the following section.



Euler-Lagrange equations term	Perturbation	Domain
2MEij,j(Ui)-P,i-Ri	2MD1Eij,j(Wi)-D1Q,i	D
Ui,i	-D2Wk,k	D

Current Euler-Lagrange equations term: 2MEij,j(Ui)-P,i-Ri

Perturbation: 2MD1Eij,j(Wi)-D1Q,i

Current domain: D

Figure 69 Adding stabilization terms to the Galerkin formulation

5.2.3 A stabilized formulation for the Stokes flow problem

Mathematical formulation :

The stabilized formulation evaluated here is the one studied in [FRA 87] for the two-dimensional Stokes flow. This formulation has to be compared with the one of the chapter 3.

The Stokes flow can be stated as follows :

Given *f*, find $(u\ ,p)$ with appropriate conditions of continuity on the domain $\Omega\subset\Re^{n_{sd}}$ such that $(n_{sd}=2)$:

$$2\mu\varepsilon_{ij,j}(u)-p_{,i}=f_i \text{ in } \Omega$$

$$u_{i,i}=0 \qquad \text{in } \Omega$$

For the sake of simplicity, the homogeneous Dirichlet problem is adopted for derivation.

The variational formulation chosen leads to the following weak form :

Given *f*, find $(u,p)\in\mathcal{S}\times\mathcal{P}$ such that for all $(w,q)\in\mathcal{W}\times\mathcal{P}$ ($\mathcal{P},\mathcal{W}$ and $\mathcal{S}$ are defined in Chapter 4):

$$-\int_\Omega 2\mu\varepsilon_{ij}(u)\varepsilon_{ij}(w)dv+\int_\Omega pw_{i,i}dv+\int_\Omega u_{i,i}qdv-\int_\Omega f_iw_idv=0$$

Let u^h, p^h, w^h and q^h be respectively the approximations of *u*, *p*, *w* and *q*. Let us define the approximation spaces $\mathcal{S}^h$, $\mathcal{W}^h$ and $\mathcal{P}^h$ intersection between, respectively $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{P}$, and a space of continuous piecewise polynomial finite element interpolations chosen for the fields.

The problem can be approximated by :

Given f, find $(u^h,p^h)\subset(\mathcal{S}^h\times\mathcal{P}^h)$ such that for all $(w^h,q^h)\subset(\mathcal{W}^h\times\mathcal{P}^h)$:

$$R((u^h,p^h);(w^h,q^h))=-\int_\Omega 2\mu\varepsilon_{ij}(u^h)\varepsilon_{ij}(w^h)dv+\int_\Omega p_hw_{i,i}^hdv+\int_\Omega u_{i,i}^hq^hdv-\int_\Omega f_iw_i^hdv=0$$

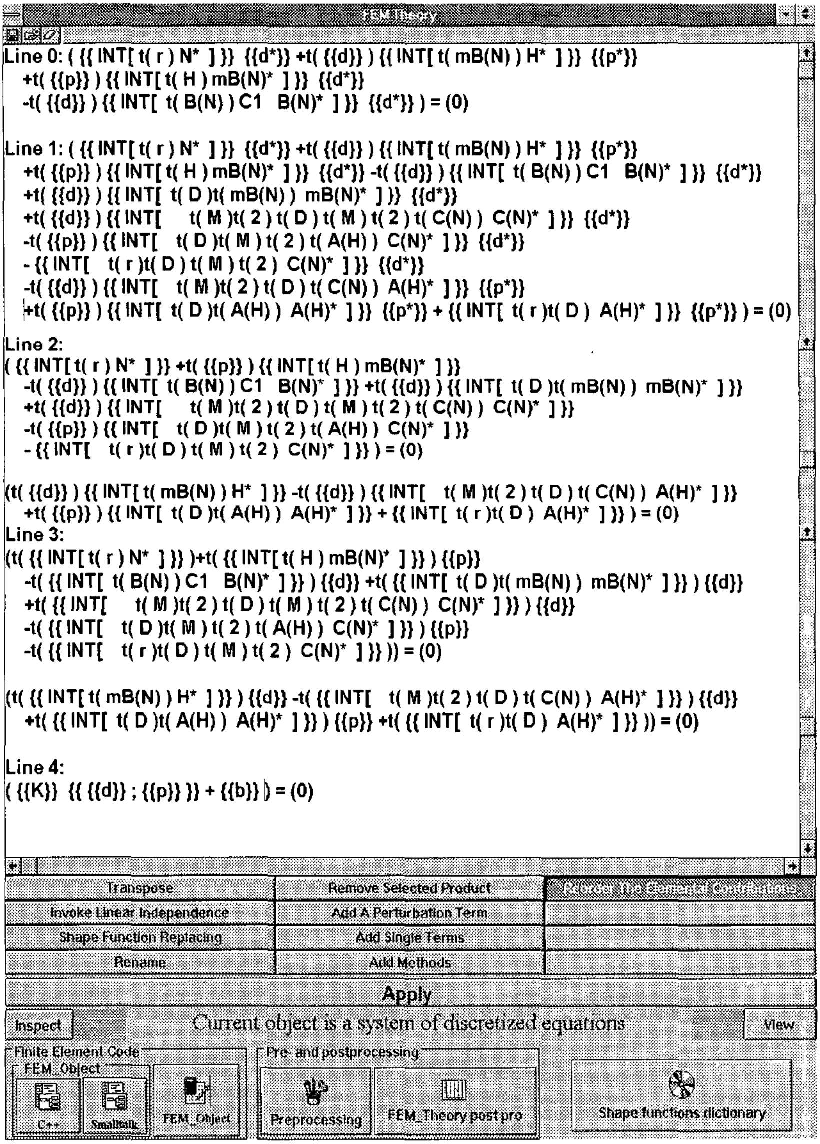
The problem arising in this formulation comes from its mixed character. The stabilized formulation proposed by Franca in [FRA 87] can be written as follows :

Given f, find $(u^h,p^h)\subset(\ {}^h\times\mathcal{P}^h)$ such that for all $(w^h,q^h)\subset(\mathcal{W}^h\times\mathcal{P}^h)$:

$$R((u^h,p^h);(w^h,q^h))+\sum_{\Omega^e\in\Omega^h}\left[\int_{\Omega^e}(2\mu\varepsilon_{ij,j}(u^h)-p_{,i}^h-f_i)D_1(2\mu\varepsilon_{ij,j}(w^h)-q_{,i}^h)dv-\int_{\Omega^e}u_{i,i}^hD_2w_{i,i}^hdv\right]=0$$

where D_1 and D_2 are stabilization parameters.

This method consists in adding mesh dependent terms, in fact a least square form of the Euler-Lagrange equations, to the usual Galerkin formulation. The form of the stabilization terms is discussed at length in [FRA 87], particularly the choice of the form of stabilization parameters $D_1=\dfrac{\delta_1h^2}{2\mu}$ and $D_2=2\mu\delta_2$, where h is a mesh parameter (see discussion about mesh parameter in [HAR 92]), and δ_1 and δ_2 are stabilization parameters. Notice that this stabilization parameter design does not respect the advective-diffusive limit conditions (see [HUG 89]). More recent designs of this method, referred as Galerkin Least Squares method, can be found in [FRA 92]. The choice of the stabilization parameter is a crucial point of the method.



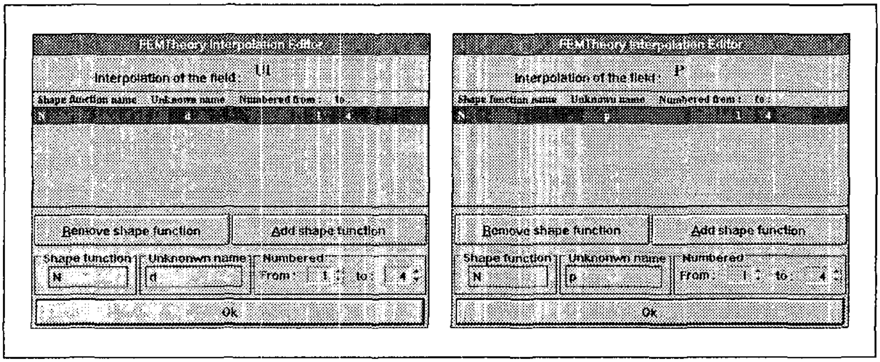
```
Line 0: ( {{ INT[ t( r ) N*  ] }}  {{d*}} +t( {{d}} ) {{ INT[ t( mB(N) ) H*  ] }}  {{p*}}
   +t( {{p}} ) {{ INT[ t( H ) mB(N)*  ] }}  {{d*}}
   -t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ] }}  {{d*}} ) = (0)

Line 1: ( {{ INT[ t( r ) N*  ] }}  {{d*}} +t( {{d}} ) {{ INT[ t( mB(N) ) H*  ] }}  {{p*}}
   +t( {{p}} ) {{ INT[ t( H ) mB(N)*  ] }}  {{d*}} -t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[  t( D )t( mB(N) )  mB(N)*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[     t( M )t( 2 ) t( D ) t( M ) t( 2 ) t( C(N) )  C(N)*  ] }}  {{d*}}
   -t( {{p}} ) {{ INT[    t( D )t( M ) t( 2 ) t( A(H) )  C(N)*  ] }}  {{d*}}
   - {{ INT[    t( r )t( D ) t( M ) t( 2 )  C(N)*  ] }}  {{d*}}
   -t( {{d}} ) {{ INT[    t( M )t( 2 ) t( D ) t( C(N) )  A(H)*  ] }}  {{p*}}
   +t( {{p}} ) {{ INT[  t( D )t( A(H) )  A(H)*  ] }}  {{p*}} + {{ INT[  t( r )t( D )  A(H)*  ] }}  {{p*}} ) = (0)

Line 2:
( {{ INT[ t( r ) N*  ] }} +t( {{p}} ) {{ INT[ t( H ) mB(N)*  ] }}
   -t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ] }} +t( {{d}} ) {{ INT[  t( D )t( mB(N) )  mB(N)*  ] }}
   +t( {{d}} ) {{ INT[     t( M )t( 2 ) t( D ) t( M ) t( 2 ) t( C(N) )  C(N)*  ] }}
   -t( {{p}} ) {{ INT[    t( D )t( M ) t( 2 ) t( A(H) )  C(N)*  ] }}
   - {{ INT[    t( r )t( D ) t( M ) t( 2 )  C(N)*  ] }} ) = (0)

(t( {{d}} ) {{ INT[ t( mB(N) ) H*  ] }} -t( {{d}} ) {{ INT[    t( M )t( 2 ) t( D ) t( C(N) )  A(H)*  ] }}
   +t( {{p}} ) {{ INT[  t( D )t( A(H) )  A(H)*  ] }} + {{ INT[  t( r )t( D )  A(H)*  ] }} ) = (0)
Line 3:
(t( {{ INT[ t( r ) N*  ] }} )+t( {{ INT[ t( H ) mB(N)*  ] }} ) {{p}}
   -t( {{ INT[  t( B(N) ) C1   B(N)*  ] }} ) {{d}} +t( {{ INT[  t( D )t( mB(N) )  mB(N)*  ] }} ) {{d}}
   +t( {{ INT[     t( M )t( 2 ) t( D ) t( M ) t( 2 ) t( C(N) )  C(N)*  ] }} ) {{d}}
   -t( {{ INT[    t( D )t( M ) t( 2 ) t( A(H) )  C(N)*  ] }} ) {{p}}
   -t( {{ INT[    t( r )t( D ) t( M ) t( 2 )  C(N)*  ] }} )) = (0)

(t( {{ INT[ t( mB(N) ) H*  ] }} ) {{d}} -t( {{ INT[    t( M )t( 2 ) t( D ) t( C(N) )  A(H)*  ] }} ) {{d}}
   +t( {{ INT[  t( D )t( A(H) )  A(H)*  ] }} ) {{p}} +t( {{ INT[  t( r )t( D )  A(H)*  ] }} )) = (0)

Line 4:
( {{K}}  {{ {{d}} ; {{p}} }} + {{b}} ) = (0)
```

Current object is a system of discretized equations

Figure 70 Derivation of the stabilized formulation for Stokes flow

5.2.3.1 Derivation of the stabilized formulation in FEMTheory

The step leading to the weak form of [FRA 87] is quite identical with the one of the chapter 4 section, the notations are the same (only the term corresponding to the Neumann boundary condition and the term in "lambda" are missing). The discretized weak form is posted on line 0 in Figure 70. The only difference in the derivation lies in the fact that a piecewise bilinear finite element interpolation has been chosen for both velocity and pressure (Q1/Q1 element) on a quadrilateral element; the choice of the interpolation is made on the screen in Figure 71.



Interpolation of the field : U1

Shape function name	Unknown name	Numbered from :	to :
N	d	1	4

Shape function: N
Unknonwn name: d
Numbered: From : 1 to : 4

Interpolation of the field : P

Shape function name	Unknown name	Numbered from :	to :
N	p	1	4

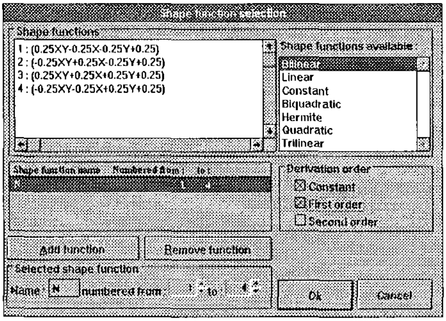
Shape function: N
Unknonwn name: p
Numbered: From : 1 to : 4

Figure 71 Choice of the interpolation for velocity and pressure

The stabilization terms are added by pushing the button *"Add Perturbation Terms"* and the window in Figure 69 appears. It is just necessary to introduce the form of the Lagrange equation and the form of the corresponding weight. The terms introduced are then integrated onto the element and their discretized form is introduced in the weak form. Although the new terms are only introduced at the elemental level (discrete summation of the terms over all the elements), the notation in the new formulation needs not be different here, because all the matrices are evaluated at the elemental level and then globally assembled; this corresponds to a discrete summation over all the discretized domain. The new formulation appears on line 1 in Figure 70. The shape functions are then chosen, bilinear shape functions for «N» (see Figure 72). On line 2, the arbitrariness of the weighting functions is invoked ; the result is a system of two discretized equations. On line 3, the system is assembled to get a system of one equation; the change of notation is the same as the one of the stokes flow formulation in chapter 3. The code is then introduced in the C++ version of numerical code FEM_Object. The numerical tests are shown in the next section.

Remark 1 : a classical two by two quadrature is used here for all the integrals; this is different from the penalty formulation of [ZIM 95a] where a two by two quadrature is used for velocity and a one point quadrature is used for pressure.

Remark 2 : as a piecewise bilinear interpolation is used, the terms «Eij,j(W)» and «Eij,j(U)» are zero and can be dropped here to accelerate symbolic and numeric computation. They are only shown for the symbolic derivation.


Shape function selection

Shape functions

1 : (0.25XY-0.25X-0.25Y+0.25)
2 : (-0.25XY+0.25X-0.25Y+0.25)
3 : (0.25XY+0.25X+0.25Y+0.25)
4 : (-0.25XY-0.25X+0.25Y+0.25)

Shape functions available :

Bilinear
Linear
Constant
Biquadratic
Hermite
Quadratic
Trilinear

Shape function name	Numbered from :	to :
N	1	4

Derivation order

☒ Constant
☒ First order
☐ Second order

Selected shape function

Name : N numbered from : 1 to : 4

Figure 72 Choice of shape functions for interpolated fields

5.2.3.2 Numerical tests of the stabilized element for Stokes flow

This element is also tested on the cavity flow problem. The description of the problem is given in chapter 3. This Q1/Q1 stabilized element is tested on a 17*17 mesh.

The constitutive parameter taken is : $\mu = 1$. The method for computing stabilization parameters «D1» and «D2» is added by hand in the numerical program ($D_1 = \dfrac{\delta_1 h^2}{2\mu}$ and $D_2 = 2\mu\delta_2$). The stabilization parameters used for the results shown in Figure 73 are : $\delta_1 = 0.5$ and $\delta_2 = 0$. These results are in accordance with the one shown in [FRA 87].

A comparison with the formulation presented in the Chapter 4 can now be done. Both formulations lead to acceptable numerical results for velocity field on this example. On the contrary, as it is well known, the Q1/P0 element has poor performances for the pressure evaluation (note that here no smoothing is performed for post-processing the results). The formulation on this numerical case lets a checkerboard phenomenon appear. The Q1/Q1 stabilized element presented in this section has quite good results for the pressure evaluation, but exhibits too much diffusion. The latter comes certainly from the rather coarse mesh used on this precise numerical example. This study has shown the possibility to evaluate rapidly two different formulations, and to compare them. The performance of the generated code allows full-scale testing.

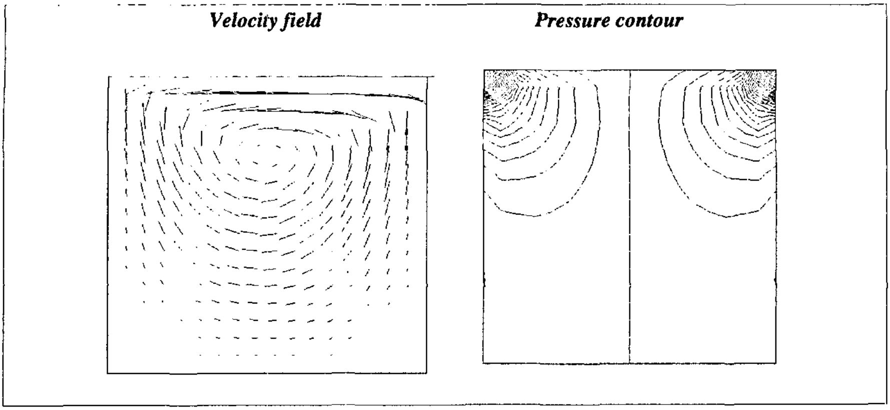


Figure 73 Numerical results for the cavity flow problem (Stokes flow with Q1/Q1 stabilized element)

Remark : the additional application of stabilized methods described in Appendix B shows the generality of the developments described in this section.

5.3 Space-time formulations

5.3.1 Discontinuous space-time formulations

In finite element computational fluid dynamics, facing with the computation of deforming domains leads to a crucial strategic choice. This problem can be solved by adding a new unknown and a new equation to handle the interface (see e.g. [COD 94, SUS 94 and TEZ 97] and references therein); but without using additional unknowns, the formulation to be used needs somehow to embed Lagrangian ingredients. The first possibility would be to use a fully Lagrangian formulation; large and sometimes unnecessary mesh distortions are one of the drawbacks of the method. An alternative approach is to use formulations mixing Lagrangian and Eulerian concepts. One of the most widely used is the Arbitrary Lagrangian Eulerian approach, widely spread in the finite element community [HUG 81] [HUE 88]. Discontinuous in time space-time formulations, initially used on a fixed mesh for accuracy purposes (see e.g. [HUG 88a] for elastodynamics), were used with moving meshes first in [HAN 92 a and b, TEZ 92 b and c]. The great interest of the formulation is its simplicity and its flexibility, i.e. its capability to allow mesh moving (imposed or not). This method has also been used for large-scale flow simulations (see e.g. [BEH 94 and MAS 97] and references therein).
The purpose of this section is to introduce into the environment FEM_Theory, the concepts needed to handle this type of formulation.

5.3.2 Integration of discontinuous space-time formulations concepts in FEMTheory

5.3.2.1 The objects needed for discontinuous space-time formulations

As in chapter 2, the best way to illustrate the new approach we want to introduce in the environment, is to isolate the new concept by means of a simple formulation. The new objects and behavior can then be deduced from it, and a new class can be described.

5.3.2.1.a The discontinuous space-time formulation for a linear one-dimensional advective equation

In this section, the formulation is presented on the resolution of a simple linear one-dimensional advective equation. This is studied at length in [SHA 88]. The purpose of this section is to introduce symbolic object-oriented concepts to manage this kind of space-time formulations (see [TEZ 92b], [TEZ 92c] and [HAN 92b]) in FEM_Theory. The formulation is recalled here.

The strong form of the problem is given as follows:

Find $u(x,t)$ with appropriate continuity conditions on $\Omega = [0,1]$ for $0 \le t \le t_f$ such that :

$u_{,t} + Au_{,x} = 0$ on Ω

with boundary conditions : $u(0,t) = \bar{u}$, $u(1,t) = 0$

and initial conditions : $u(x,0) = u_0$

where A is the advection constant.

The variational formulation is written on the space-time domain Q_n, on a space-time slab bounded by t_n and t_{n+1} as illustrated in Figure 74 (see [BEH 94] for more details about notations). Define the approximation spaces for u solution and w weighting functions :

$$(\mathcal{S}^h)_n = \{u^h \in [H^1(Q_n)]^h \mid u^h = \bar{u} \quad \text{on } (P_n)_{\bar{u}}\}$$
$$(\mathcal{V}^h)_n = \{w^h \in [H^1(Q_n)]^h \mid w^h = 0 \quad \text{on } (P_n)_{\bar{u}}\}$$

The approximation of the variational is :

For each time slab $[t_n, t_{n+1}]$, find $u^h \in (\mathcal{S}^h)_n$ such that $\forall w^h \in (\mathcal{V}^h)_n$:

$$\int_{Q_n} (u^h_{,t} + u^h u^h_{,x}) w^h dq + \sum_{e=1}^{n_{el}} \int_{Q_{ne}} (u^h_{,t} + u^h u^h_{,x}) \tau (w^h_{,t} + u^h w^h_{,x}) dq + \int_{\Omega_n} [[u^h]] (w^h)^+_n dv = 0$$

The design of the stabilization parameter proposed in [SHA 88] is :

$$\tau = \left(\left(\frac{2}{\Delta t} \right)^2 + \left(\frac{2|u|}{h} \right)^2 \right)^{-\frac{1}{2}}$$

where $\Delta t = t_{n+1} - t_n$ and h is the mesh parameter (spatial length of the element in the current case), and where $[[u^h]] = (u^h)^+_n - (u^h)^-_n$ is called 'jump term', corresponding to the following definition: $(u^h)^\pm_n = \lim_{\varepsilon \to 0} u^h(t_n \pm \varepsilon)$.

The first integral of the formulation is the classical Galerkin formulation written on the domain Q_n ; the second one is the Galerkin Least-Squares term added for stabilization purposes ; the last one makes it possible to enforce the continuity of the solution u, in a weak sense, over the global domain. Theoretical details about the formulation can be found in [SHA 88].

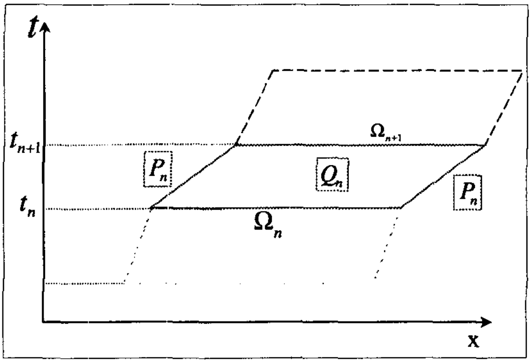


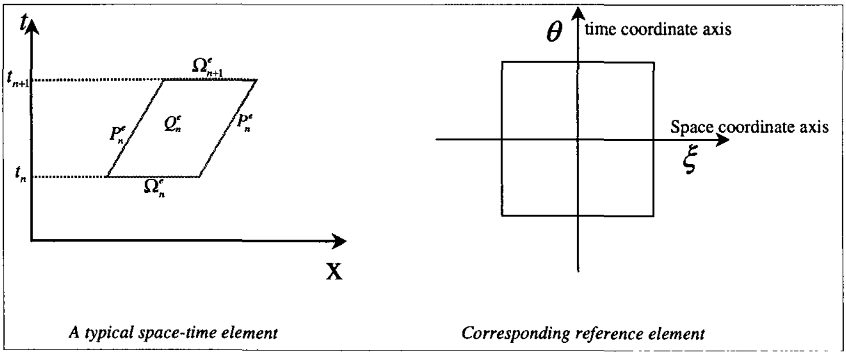
Figure 74 Description of the space-time domain

5.3.2.1.b The objects for the discontinuous space-time formulation

The first two terms of the formulation on the space-time domain can be directly introduced in the FEM_Theory environment. In the sense of the finite element method, the time can be considered as an additional coordinate. So, the numerical treatment is obvious in the symbolic environment. But a new concept is needed to represent the third term, i.e. the 'jump term' $\int_{\Omega_n}[[u^h]](w^h)_n^+ dv$. Part of it is known, i.e. $(u^h)_n^-$ is computed at the previous time slab; and part of it is the current unknown, $(u^h)_n^+$. From the point of view of the finite element method, the formulation leads to the solution of a linear system at each time slab of the form $K d = f$ where d is the vector of the nodal unknowns. The elemental contributions coming from the first two terms are obvious if classical finite elements are used. It is worth describing the elemental contributions due to the 'jump term'. They can be expressed by means of notations of [HUG 87] as follows, on an element illustrated in Figure 75 :

$$K^e_{jump\,term} = \int_{\Omega^e_n} N^t N \, d\Omega \text{ and } f^e_{jump\,term} = K^e_{jump\,term}\, d^-$$

where N is the classical matrix of shape functions of [HUG 87] and d^- is the vector of nodal unknowns computed on the previous time slab. The integration is done on the space domain in the initial configuration (at t_n), i.e. on the surface Ω^e_n, as seen in Figure 75.This shows that the FEM_Theory environment is capable of building elemental matrices such as $K^e_{jump\,term}$; the new concepts to add here are the ones to manage and interpret the 'jump term', i.e. mainly concepts linked to the numerical integration scheme, and to the automatic implementation into the numerical code.



A typical space-time element

Corresponding reference element

Figure 75 An example of space-time element for a one-dimensional space

The idea is then to introduce a new object to represent the 'jump term' in the variational formulation, and to enrich the existing objects to handle this new object, particularly for automatic integration in the numerical code. In order to make the implementation easier, one can note the following :

$$\int_{\Omega_n} [[u^h]](w^h)^+_n dv = [[\int_{\Omega_n} u^h (w^h)^+_n dv]]$$

This makes it possible to have a treatment of the 'jump' at a higher level than inside the integral. The new object, instance of class called **JUMP_TERM**, is represented by the double bracket notation. It is natural to manipulate it as a special term in the formulation. The structure of the object appears in Figure 76. The object has as only a piece of data, its variable which is in this example an integral. Most of the tasks are decentralized to the attribute variable ; the algebraic manipulation methods are inherited from the class **Term**. This object is a specialization of the object term. Additional tasks have to be added to other objects such as products for generating the code with the selector of methods needed for 'jump term' in the numerical code. Note that the space domain of the integral Ω^e_n has to be recognized; the name of the domain used is «Sp» for spatial domain. A generalization of the generation of the code is needed here. Finally, the variational formulation (successively classes **IntEquation**, **DiscretizedEquation**, **System**) is now given an attribute to characterize the type of the formulation, i.e. either "Semi-discrete approach" or "Space-time approach". In the latter, time is considered as a simple coordinate such as a space coordinate.

Class **JUMP_TERM**

Example : $[[\int_{\Omega_n} u^h (w^h)^+_n dv]]$

main attributes :

variable : $\int_{\Omega_n} u^h (w^h)^+_n dv$ (here an integral)

main tasks :

manipulations in the variational form such as addition, product, ... (can be inherited from class **Term**)

generation of code

Figure 76 Typical instance of **JUMP_TERM**

5.3.2.2 Class JUMP_TERM

The structure of the class is summarized on Table 15.

Table 15 Class **JUMP_TERM**

Class JUMP_TERM **Inherits from : Term,StructureWithDimension, FEMTheoryMathmaticalStructures, FEMTheory, Object**		
Inherited tasks	**Inherited attributes**	**Inherited methods**
1) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm: term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow: term (from **FEMTheoryMathmaticalStructures**)
2) algebraic manipulations		+, *, ... (from **Term**)
Tasks	**Attributes**	**Methods**
1) manipulation	- variable	asFunctional comesFromSurfaceLoad deriveWithRespectToVariable: i findAllUnknowns findMatrixCorrespondingToUnknown: discretVar get0KAFUnknownMatrix getDirectionalDerivative getJumpTermCorrespondingToUnknown: discretVar printString replaceYourselfUsingDictionary: dict transpose variable: aTerm
2) analysis		isBodyMatrix isSurfaceMatrix isZero
3) discretization		getDiscretizedForm
4) creation of code		createCPlusLoadMethodsIn: path forTimeDependentElement: elementName

Attributes

Class **JUMP_TERM** has one attribute called **variable**. This class is used either for the continuous problem or for the discrete problem ; the simplicity of this structure does not necessitate the sharing of the tasks among two classes. For the continuous problem, **variable** can a priori be of any class (**Term**, **Integral**, **Expression**, ...). Only class **Integral** is used here. For the discrete problem, **variable** is an instance of **DiscretizedExpression**. The consequence of this, is that both tasks, for continuous and discrete problems lie in the same structure.

Tasks description

Tasks linked to the integration into the hierarchical parent tree organization (see chapter 2) and to algebraic manipulations are inherited from the superclasses. The tasks may be decomposed into four groups. Most of the methods are in fact a specialization of existing methods of class **Term**.

Most of the tasks are decentralized to the attribute **variable**. The only method which is specific to the class is *findMatrixCorrespondingToUnknown: discretVar* ; it returns, in the case of the discrete form, the elemental contribution corresponding to the nodal unknown vector of the 'jump term'.

The analysis tasks are as before specialization of class **Term**. This behavior can be linked to manipulations

The discretization procedure is decentralized to the attribute variable, and the scheme described in Chapter 2 to build elemental contributions don't need any special implementation.

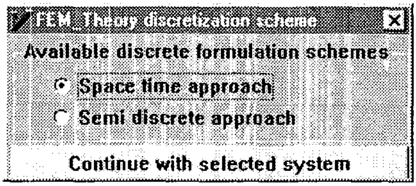
Code generation implies here the creation of the elemental contributions in the left- and right hand side of the discrete linear system (see section 5.3.2.1). Both operations are initiated at this stage, but practical tasks are decentralized to the discrete form stored in attribute **variable**.

Remark : An example of code generated for this space-time formulation is given in section 5.3.3.2.

5.3.2.3 The graphical environment

The FEM_Theory environment is enriched with new graphical features. The first one is the notation used for the 'jump term'. The double bracket notation introduced here can be directly integrated into FEM_Theory. As an example the term $[[\int_{\Omega_n} u\,(w)_n^+ dv]]$ is represented «[[INT{UW // Sp}]]» (see next section). Note, that here the term is shown for the continuous problem.

A new prompter window is added in order to ask the user which type of formulation he whishes to derive (see Figure 77) ; this only influences the choice of the time coordinate to be taken into account as a simple coordinate axis. An example of space-time formulation derivation follows.



FEM_Theory discretization scheme

Available discrete formulation schemes

● Space time approach

○ Semi discrete approach

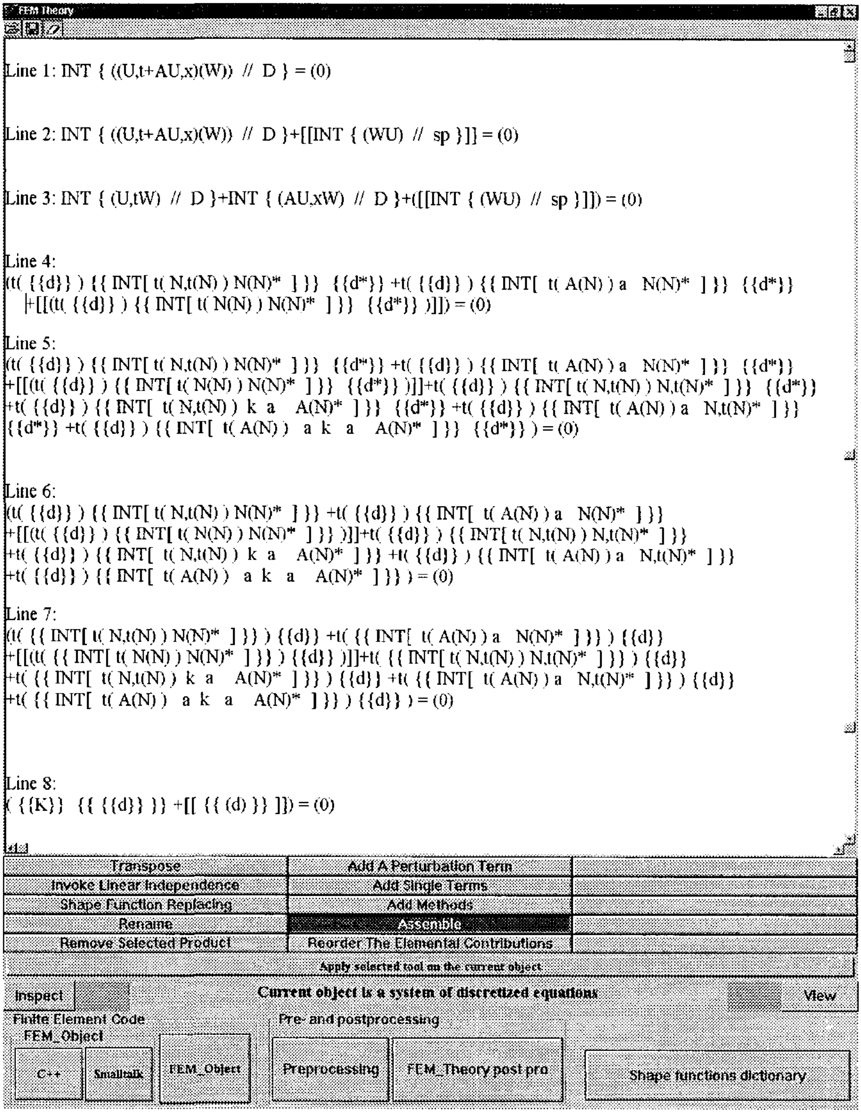
Continue with selected system

Figure 77 Prompter to define the type of the finite element formulation

5.3.3 The linear one-dimensional advection equation in FEM_Theory

5.3.3.1 Derivation of the linear 1-D advection equation

The space-time formulation for the linear one-dimensional advection equation is derived in FEM_Theory, in the window of Figure 78 (the notations are close to the ones of the previous section). As similar derivations have already been shown in previous sections, only a brief line by line description is made here. On line 1 the original Galerkin formulation on the whole space-time domain is posted. On line 2, the 'jump term' is added ; the formulation is written on a time slab, it is a Galerkin continuous in space and discontinuous in time form. On line 4, the formulation is approximated. The reference element chosen here is a four-node element for «U» and «W». Stabilization Galerkin least-squares terms are added on line 5 (to the approximated formulation). The arbitrariness of the weighting function is invoked, the resulting system of discrete equations appears on line 6. The equation is then transposed, in line 7, and an obvious change of notation is made. Bilinear interpolation for «U» and «W» is chosen, and the code is generated. A two by two Gauss integration rule is chosen for all elemental contributions. The code is then generated.



```
Line 1: INT { ((U,t+AU,x)(W)) // D } = (0)

Line 2: INT { ((U,t+AU,x)(W)) // D }+[[INT { (WU) // sp }]] = (0)

Line 3: INT { (U,tW) // D }+INT { (AU,xW) // D }+([[INT { (WU) // sp }]]) = (0)

Line 4:
(t( {{d}} ) {{ INT[ t( N,t(N) ) N(N)* ] }} {{d*}} +t( {{d}} ) {{ INT[ t( A(N) ) a N(N)* ] }} {{d*}}
 +[[(t( {{d}} ) {{ INT[ t( N(N) ) N(N)* ] }} {{d*}} )]]) = (0)

Line 5:
(t( {{d}} ) {{ INT[ t( N,t(N) ) N(N)* ] }} {{d*}} +t( {{d}} ) {{ INT[ t( A(N) ) a N(N)* ] }} {{d*}}
+[[(t( {{d}} ) {{ INT[ t( N(N) ) N(N)* ] }} {{d*}} )]]+t( {{d}} ) {{ INT[ t( N,t(N) ) N,t(N)* ] }} {{d*}}
+t( {{d}} ) {{ INT[ t( N,t(N) ) k a A(N)* ] }} {{d*}} +t( {{d}} ) {{ INT[ t( A(N) ) a N,t(N)* ] }}
{{d*}} +t( {{d}} ) {{ INT[ t( A(N) ) a k a A(N)* ] }} {{d*}} ) = (0)

Line 6:
(t( {{d}} ) {{ INT[ t( N,t(N) ) N(N)* ] }} +t( {{d}} ) {{ INT[ t( A(N) ) a N(N)* ] }}
+[[(t( {{d}} ) {{ INT[ t( N(N) ) N(N)* ] }} )]]+t( {{d}} ) {{ INT[ t( N,t(N) ) N,t(N)* ] }}
+t( {{d}} ) {{ INT[ t( N,t(N) ) k a A(N)* ] }} +t( {{d}} ) {{ INT[ t( A(N) ) a N,t(N)* ] }}
+t( {{d}} ) {{ INT[ t( A(N) ) a k a A(N)* ] }} ) = (0)

Line 7:
(t( {{ INT[ t( N,t(N) ) N(N)* ] }} ) {{d}} +t( {{ INT[ t( A(N) ) a N(N)* ] }} ) {{d}}
+[[(t( {{ INT[ t( N(N) ) N(N)* ] }} ) {{d}} )]]+t( {{ INT[ t( N,t(N) ) N,t(N)* ] }} ) {{d}}
+t( {{ INT[ t( N,t(N) ) k a A(N)* ] }} ) {{d}} +t( {{ INT[ t( A(N) ) a N,t(N)* ] }} ) {{d}}
+t( {{ INT[ t( A(N) ) a k a A(N)* ] }} ) {{d}} ) = (0)

Line 8:
( {{K}} {{ {{d}} }} +[[ {{ (d) }} ]]) = (0)
```

Figure 78 Derivation of the space-time formulation for 1-D advection equation

5.3.3.2 Code generated automatically into FEM_Object

As described in chapter 4, a new class is generated. As the FEM_Object code has been enhanced to support deforming domains with space-time formulations (see [EYH 96c]), the new element **NewElement** is added now as a subclass of a class called **STF_ELEMENT** which contains special features for domains moving, as shown in the hierarchy of FEM_Object in Figure 79. Note that on the list of methods given in Figure 80, a new method to compute the 'jump term' in the load vector of the right-hand side in FEM_Object appears (see chapter 3). The contribution to left hand side of the 'jump term' is automatically integrated into the stiffness matrix.

```
FEMObject
        Dictionary
        Dof
        Domain
        FEMComponent
                Element
                        PlainStrain
                        STF_ELEMENT
                                NewElement
                Load
                Material
                Node
                TimeItegrationScheme
                TimeStep
        LinearSystem
        Skyline
```

Figure 79 Partial view of the hierarchy of FEM_Object for space-time formulation

```
computeGaussPts
giveGaussPtsArray
...
computeStiffnessMatrixAt:
...
computeJumpTermAt:
...
giveJacobianMatrixAt:

constructor and destructor
```

Figure 80 Partial view of the methods added for space-time formulation in FEM_Object

5.3.3.3 Numerical tests

The test done uses as initial condition a discontinuity over one element as shown in Figure 81. Space is decomposed into 20 elements, the height of the time slab is $\Delta t = 0.05s$. The boundary conditions are $u(0) = 1$ and $u(1) = 0$. The first test is done with a fixed mesh. The results are shown in Figure 82 with the label STF (space-time formulation) for various values of the stabilization parameter at *t=0.5*. This shows that for various values of stabilization parameters the oscillations before and after the discontinuity are attenuated and that the discontinuity is caught correctly. A second derivation has been done with a semi-discrete approach, with an equivalent stabilization scheme. The results are also reported in Figure 82. The numerical integration scheme in time used for solving the problem in FEM_Object is a generalized trapezoidal rule presented in [HUG 87 Chap. 8]. The results are posted for various values of the stabilization parameter. The first remark that can be made is that the formulation can not catch the sharp discontinuity as the space-time formulation. Adding stabilization, rapidly adds too much diffusion. But the position of the discontinuity can also be caught. In conclusion, the space-time formulation seems to give better results in capturing of sharp discontinuities.

Finally, the mesh moving procedure is introduced in the numerical computation. The mesh is moved with the advection velocity ($A=1$). The results are shown in Figure 83. It shows that the local advective effects disappear ; as a matter of fact, the exact solution is obtained. This feature can be interesting whenever sharp discontinuities have to be caught.

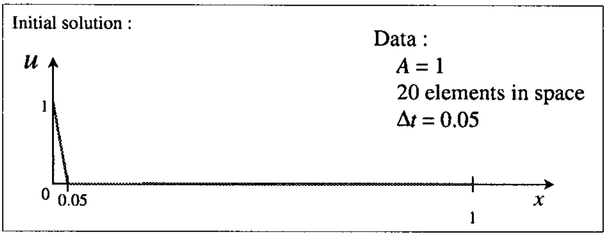


Figure 81 Initial solution for *u* for the numerical test of the linear advection equation

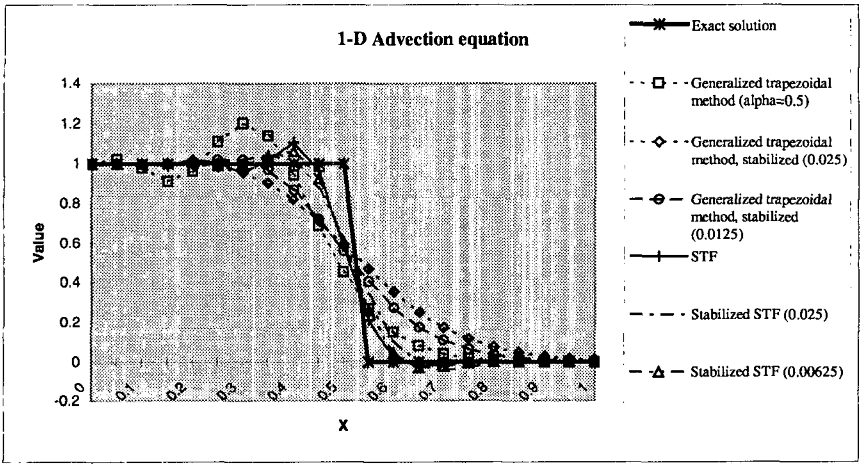


Figure 82 Numerical test of the linear advection equation at $t = 10\ \Delta t$

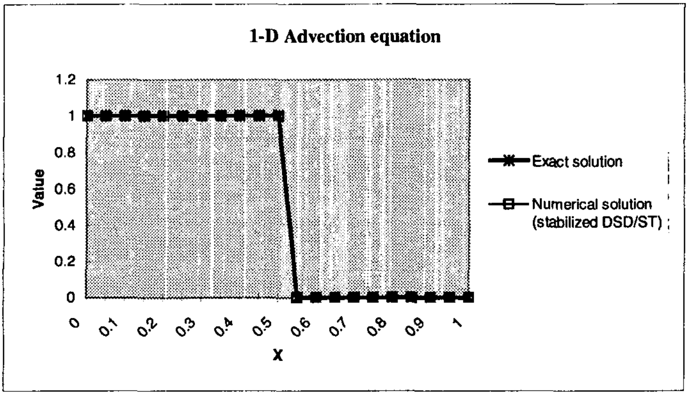


Figure 83 Numerical solution for moving domain at $t = 10\ \Delta t$

Remark : as an application of the concepts developed in this chapter, a formulation for the incompressible Navier-Stokes equations is given in Chapter 7.

Chapter 6 A nonlinear approach in FEM_Theory: Application to advection dominated equation models

6.1 Objective

The aim of this part is to develop a convenient framework for nonlinear problems. In the previous sections an environment capable of representing various types of linear variational formulations was described. A basic framework to work on linear mixed or/and convection dominated variational and discontinuous in time space-time formulations was elaborated. This was done with the aim of solving Navier-Stokes flow. The natural following step to achieve this is to handle nonlinear problems.

Representing nonlinear problems in FEM_Theory brings no additional difficulties. This can be done either by adding new objects **Term**, corresponding to nonlinear operators (in the sense of "material" or "geometry"), or by multiplying linear expressions.

The approach chosen here is to provide the environment with linearization techniques. The linearized problem may then be solved numerically using a Newton type algorithm.

6.2 A theoretical approach for linearization

In this section, some theoretical features of linearization that are to be used and implemented into the symbolic environment are recalled. More details can be found e.g. in [MAR 83, Chap. 4] and [HUG 78].

6.2.1 Definition of linearization

Consider a function *f*, $u \mapsto f(u)$, defined with appropriate conditions of continuity. The linear part of *f* at u may be obtained by using Taylor's formula expansion at first order:

$$L_u f(\delta u) = f(u) + Df(u) \cdot \delta u$$

where δu is an increment.

The problem is then to define a convenient derivative to compute the gradient part $Df(u) \cdot \delta u$ of the linear part of *f*. In various situations, the standard directional derivative can be used in the consistent linearization procedure.

6.2.2 Directional derivative

Definitions

Given a function *f*: $u \mapsto f(u)$, defined with appropriate conditions of continuity, the directional derivative of *f* at *u* in the direction δu is by definition:

$$\frac{d}{d\varepsilon} f(u+\varepsilon\, \delta u) \Big|_{\varepsilon=0}$$

where ε is a scalar parameter.

Remark: $\frac{d}{d\varepsilon} f(u+\varepsilon\, \delta u) \Big|_{\varepsilon=0} = \lim\limits_{\varepsilon \to 0}\left(\frac{f(u+\varepsilon\, \delta u)-f(u)}{\varepsilon}\right)$

The directional derivative measures the rate of change of the function *f* in the direction δu at point *u* and can be used to compute the gradient part of the linear part of *f*.

$$Df(u)\cdot \delta u = \frac{d}{d\varepsilon} f(u+\varepsilon\, \delta u) \Big|_{\varepsilon=0}$$

The latter allows carrying out consistent linearization for various problems. More fundamental issues about the directional derivative may be found in [HUG 78] and some examples of its application to the linearization of formulations in the domain of solid mechanics are given in [KOZ 94], [IBR 93].

Property

The directional derivative is linear. Therefore, for f and g two given functions with appropriate conditions of continuity, one has:

$D(f.g)(u)\cdot \delta u = Df(u).\delta u\ g(u) + f(u)\ Dg(u)\cdot \delta u$

$D(f+g)\cdot \delta u = Df(u)\cdot \delta u + Dg(u)\cdot \delta u$

Thus, this property can be used in the symbolic environment.

Remark: The linearization of variational formulations leads generally to employ Newton type algorithms. A quadratic convergence could be expected if consistent linearization is performed.

6.3 Object-oriented concepts for symbolic derivations of nonlinear problems

6.3.1 The objects for a consistent linearization scheme in FEM_Theory

Two major concepts were introduced in the previous section. The first one is the definition of the linear part of a functional. The second one is a way to compute the gradient part of the linearization, the directional derivative. These theoretical definitions and properties are naturally implemented into the symbolic environment.

One object appears in this description: the directional increment. Thinking of it as an increment makes it possible to consider it as a special kind of term. Therefore, this object is implemented as a subclass of **Term**. A typical example of object increment is shown in Figure 84. It is worth noting that this object can be used for the continuum problem and the discrete one.

Class **Increment**

Example: δu

main attributes :

function : u

main tasks :

manipulations in the variational form such as addition, product, ... (can be inherited from class **Term**), analysis, discretization

Figure 84 Typical instance of class **Increment**

6.3.2 Implementation in FEM_Theory

6.3.2.1 The linearization procedure

The linearization is applied to an object equation, instance of class **IntEquation**, i.e. the problem to find u such that $f(u)=0$ is replaced by the problem $L_u f(\delta u) = f(u) + Df(u)\cdot \delta u = 0$. The code corresponding to this operation is shown in Figure 85. A new instance of class **IntEquation** is created, for which the left-hand side (attribute **lhs**) is given the value of *consistentLinearization*, result of the consistent linearization of the left-hand side of the current equation. As shown in Figure 87 in an example of equation, the message *computeConsistentLinearization* is sent to the left-hand side of the system, which is a functional (instance of class **Functional**). The corresponding method is posted in Figure 86. This method implements exactly the definition of the consistent linearization given in section 6.2.1. The message *computeDirectionalDerivative* goes down the roots of the hierarchical tree as illustrated in Figure 87. Note that the linearity properties of the directional derivative are implemented at the level of objects **SumList** and **ProdList**.

Implementation of the linearization remains very close to the theoretical framework and remains open to future developments.

```
computeConsistentLinearization
    | directionalDerivative consistentLinearization reply |

    self putAllLhs expand.

    consistentLinearization := (self giveLhs) computeConsistentLinearization.

    reply :=  IntEquation new
                        lhs: consistentLinearization ;
                        rhs: ('0' formYourExpression);
                        listOfTerms: listOfTerms;
                        listOfUnknowns: listOfUnknowns;
                        spaceDimension: spaceDimension;
                        problemDimension: problemDimension;
                        problemType: problemType.
    ^ reply
```

Figure 85 Method *computeConsistentLinearization* of class **IntEquation**

```
computeConsistentLinearization


            ^ (self +(self getDirectionalDerivative))
```

Figure 86 Method *computeConsistentLinearization* of class **Functional**

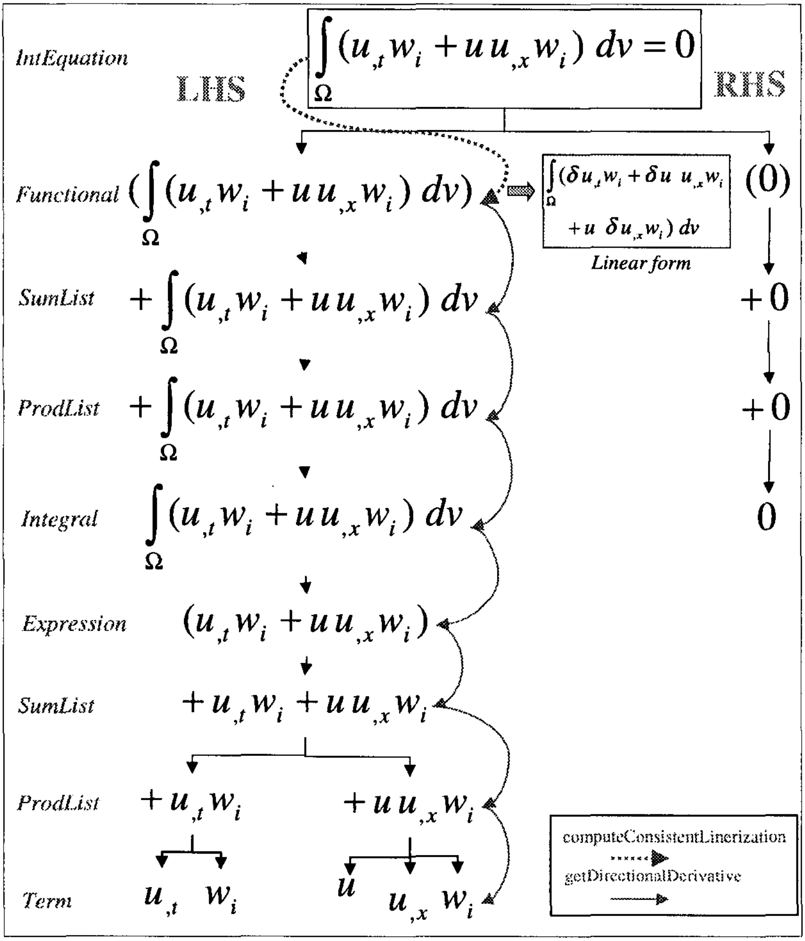

IntEquation
$\int_\Omega (u_{,t} w_i + u\, u_{,x} w_i)\, dv = 0$
LHS
RHS
Functional
$(\int_\Omega (u_{,t} w_i + u\, u_{,x} w_i)\, dv)$
$\int_\Omega (\delta u_{,t} w_i + \delta u\ u_{,x} w_i + u\ \delta u_{,x} w_i)\, dv$
Linear form
(0)
SumList
$+\int_\Omega (u_{,t} w_i + u\, u_{,x} w_i)\, dv$
$+0$
ProdList
$+\int_\Omega (u_{,t} w_i + u\, u_{,x} w_i)\, dv$
$+0$
Integral
$\int_\Omega (u_{,t} w_i + u\, u_{,x} w_i)\, dv$
0
Expression
$(u_{,t} w_i + u\, u_{,x} w_i)$
SumList
$+u_{,t} w_i + u\, u_{,x} w_i$
ProdList
$+u_{,t} w_i$
$+u\, u_{,x} w_i$
Term
$u_{,t}$
w_i
u
$u_{,x}$
w_i
computeConsistentLinerization
getDirectionalDerivative

Figure 87 Sketch of the linearization scheme in a typical equation

6.3.2.2 Class Increment

The class is described on Table 16.

Attributes

Class **Increment** has one attribute called **function**. This class is used either for the continuous problem or the discrete problem; consequently, methods for the continuum problem and for the discrete one are grouped in the same structure. The attribute **function** is a priori of class either **Term** or **DiscretizationMatrix**.

Tasks description

The tasks linked to the integration into the hierarchical parent tree organization (see chapter 2) and to algebraic manipulations are inherited from the superclasses. The tasks may be decomposed into three groups. Most of the methods are in fact a specialization of existing ones in class **Term**.

Manipulation methods: Most of the tasks are decentralized to the attribute.

Analysis methods: The analysis tasks are as before specializations of class **Term**. This behavior can be linked to the manipulations one.

Discretization: The discretization procedure is decentralized to the attribute variable, and the scheme described in chapter 3 to build elemental contributions does not need any special implementation. The result is a product (instance of **ProdList**) of the matrix form of **function** and the increment of the nodal unknowns vector.

Table 16 Class **Increment**

Class Increment **Inherits from : Term,StructureWithDimension, FEMTheory, Object**		
Inherited tasks	**Inherited attributes**	**Inherited methods**
1) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm: term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow: term
2) part of the behavior of class **Term**	Some attributes of class **Term**	
Tasks	**Attributes**	**Methods**
1) manipulation	- function	findAllUnknowns function: aTerm get0KAFUnknownMatrix getYourOperator giveElementaryMatrix giveFunction giveMatrixType printString transpose
2) analysis		is0KAF is0KAFUnknown isAnUnknown isIncrement isKAF isKAFUnknown isUnknown isVector
3) discretization		getDiscretizedForm

6.3.3 The graphical environment

The new tool *"Compute Consistent Linearization"* appears in the main window of FEM_Theory (see section 3.4.2). An additional notation is mandatory for the representation of the linear part of a functional in the main window of FEM_Theory. Thus, an instance of class **Increment** is written by adding the symbol «ð» in front of the name of the quantity. For example, the increment in u, δu, is written «ðU» (see example in section 6.5).

6.4 Enhancing the automatic programming procedure

In a sense, the linearization procedure consists in replacing a problem under the classical form $N(d) = f$ by the problem $L_a N(\delta d) = K_{\tan}(d)\,\delta d + N(d) - f = 0$. Using a Newton like algorithm described on Table 17 then solves the latter. A major additional concept is needed for the nonlinear approach. A new option is needed in the creation of code for computing the elemental contribution to the tangent modulus. These feature is added at the level of the product, i.e. class **ProdList**.

A new concept is also added here. In the iterative procedure, the elemental contributions are evaluated at the integration point and then summed. Thus, at each point are computed some values from the nodal values at the previous iteration. In order to enhance the efficiency of the code automatically generated, the idea is to automatically generate a new object for which, the only task is both to store and to compute intermediate values when necessary. This special feature is illustrated in section 6.5.3, in the context of the nonlinear 1-D advection equation. This procedure shows that even complex programming paradigms can easily be introduced in the context of automatic finite element coding.

Table 17 Description of a Newton like iterative procedure

Find d by using the iterative process:

Given an initial guess d^0.

At iteration i, solve : $K_{\tan}(d^{i-1})\,\delta d^i + N(d^{i-1}) - f = 0$

Update: $d^i = d^{i-1} + \delta d^i$

Given a convergence criteria ε, check $\dfrac{\|N(d^i) - f\|}{\|N(d^0) - f\|} \le \varepsilon$

If convergence is achieved, then $d = d^i$; else $i = i + 1$.

6.5 Application to a nonlinear one-dimensional advection equation model

6.5.1 Strong and weak form of the problem

In the previous chapter a linear one-dimensional equation was studied. We study here the nonlinear equivalent equation model. The strong form is recalled on Table 18 and the formulation on Table 19 (see Chapter 6 for details about the stabilized space-time formulation).

Table 18 Strong form of 1-D nonlinear advection equation

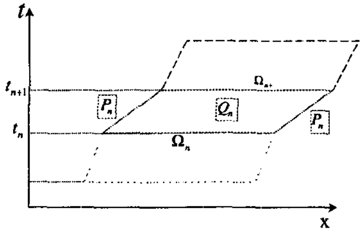
Find $u(x,t)$ in $\Omega = [0,1]$ such that:

$$u_{,t} + uu_{,x} = 0 \quad \text{in } \Omega$$

Boundary conditions: $u(0,t) = \bar{u},\ u(1,t) = 0$

Initial conditions: $u(x,0) = u_0$

Table 19 Stabilized space-time formulation for 1-D advection equation



For each time slab $[t_n, t_{n+1}]$ find $u^h \in (\mathcal{S}^h)_n$ such that $\forall w^h \in (\mathcal{V}^h)_n$:

$$\int_{Q_n} (u^h_{,t} + u^h u^h_{,x}) w^h \, dq + \sum_{e=1}^{n_{el}} \int_{Q_{ne}} (u^h_{,t} + u^h u^h_{,x}) \tau (w^h_{,t} + u^h w^h_{,x}) dq + \int_{\Omega_n} [[u^h]] (w^h)^+_n \, dv = 0$$

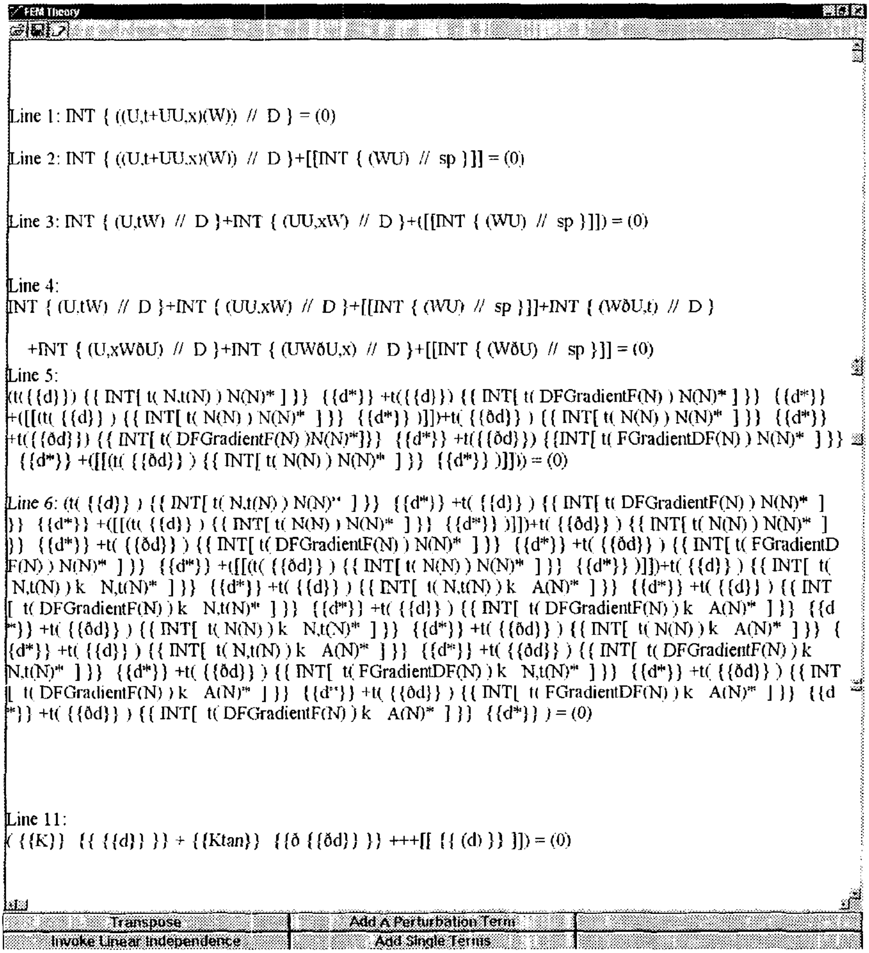
$$(\mathcal{S}^h)_n = \left\{ u^h \in [H^1(Q_n)]^h \mid u^h = \bar{u} \quad \text{on } (P_n)_{\bar{u}} \right\}$$

$$(\mathcal{V}^h)_n = \left\{ u^h \in [H^1(Q_n)]^h \mid u^h = 0 \quad \text{on } (P_n)_{\bar{u}} \right\}$$

where $\tau = \left(\left(\dfrac{2}{\Delta t} \right)^2 + \left(\dfrac{2|u|}{h} \right)^2 \right)^{-\frac{1}{2}}$ (h length of the element)

6.5.2 Derivation of the stabilized space-time formulation for 1-D nonlinear advection in FEM_Theory

The derivation described on Table 19 in FEM_Theory (see Figure 88) is similar to the linear one in the previous chapter. The main difference lies in the linearization procedure of the Galerkin formulation, which appears on line 4. On line 5, the approximated formulation appears ; the reference element chosen here is a quad, and u is interpolated at each node. Note that the elemental contributions of the tangent modulus are a factor of «∂d». On line 5, the linear part of the stabilization terms is introduced. As discussed in the previous section, the environment makes it possible to carry out consistent linearization. In order to improve the numerical efficiency, sometimes, consistent linearization is not requested. So, in the formulation some terms are "frozen" in the linearization procedure. Consequently, quadratic convergence of the Newton algorithm is no longer expected, but the number of terms participating to the tangent modulus can drastically decrease. The idea is then to "freeze" some terms in the formulation on Table 19. This is done in case of the stabilization terms added to the Galerkin formulation, where the stabilization term τ and part of the nonlinear advection term of the equation are frozen. Instead of introducing the stabilization term requested by the theory $\sum_{e=1}^{n_{el}} \int_{Q_{ne}} (u^h_{,t} + u^h u^h_{,x}) \cdot \tau(u) \cdot (w^h_{,t} + u^h w^h_{,x}) dq$, i.e. «U,t+UU,x» weighted by «TW,t+TUW,x» (FEM_Theory notations), $\sum_{e=1}^{n_{el}} \int_{Q_{ne}} (u^h_{,t} + u^h u^h_{,x}) \cdot \tau \cdot (w^h_{,t} + A w^h_{,x}) dq$ is introduced, i.e. «U,t+AU,x» weighted by «TW,t+TUW,x» and integrated into the element. So the expected "freeze" procedure is achieved in the derivation, because linearization is obtained through manipulations taking into account the unknowns of the problem, i.e. u. An additional procedure is then mandatory to fix the equality $A \equiv u^h$. The expression resulting from the addition of stabilization terms to the original Galerkin formulation is posted on line 6. A similar procedure to the one of the previous chapter for the linear equation leads to the final form posted on line 11. Bilinear shape functions are chosen for u on the space-time domain. The system is thus under the form, using classical notations that let appear the index for iterations: $K_{\tan}(d^i)\delta d^{i+1} = N(d^i) - f$. The new automatic programming procedure makes it possible to introduce the elemental contributions to the tangent modulus and the residual in the code FEM_Object. The form of the code obtained illustrates the changes made to obtain the nonlinear code.



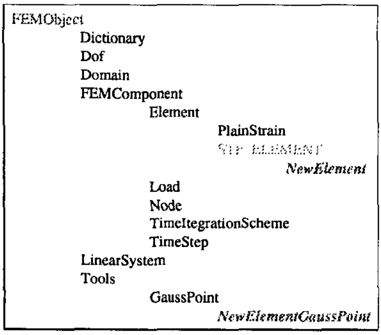
```
Line 1: INT { ((U,t+UU,x)(W)) // D } = (0)

Line 2: INT { ((U,t+UU,x)(W)) // D }+[[INT { (WU) // sp }]] = (0)

Line 3: INT { (U,tW) // D }+INT { (UU,xW) // D }+([[INT { (WU) // sp }]]) = (0)

Line 4:
INT { (U,tW) // D }+INT { (UU,xW) // D }+[[INT { (WU) // sp }]]+INT { (WδU,t) // D }

   +INT { (U,xWδU) // D }+INT { (UWδU,x) // D }+[[INT { (WδU) // sp }]] = (0)
Line 5:
(t({{d}}) {{ INT[ t( N,t(N) ) N(N)* ] }}  {{d*}} +t({{d}}) {{ INT[ t( DFGradientF(N) ) N(N)* ] }}  {{d*}}
+([[(t( {{d}} ) {{ INT[ t( N(N) ) N(N)*  ] }}  {{d*}} )]])+t( {{δd}} ) {{ INT[ t( N(N) ) N(N)*  ] }}  {{d*}}
+t({{δd}}) {{ INT[ t( DFGradientF(N) )N(N)*]}}  {{d*}} +t({{δd}}) {{INT[ t( FGradientDF(N) ) N(N)*  ] }}
 {{d*}} +([[(t( {{δd}} ) {{ INT[ t( N(N) ) N(N)*  ] }}  {{d*}} )]]) = (0)

Line 6: (t( {{d}} ) {{ INT[ t( N,t(N) ) N(N)*  ] }}  {{d*}} +t( {{d}} ) {{ INT[ t( DFGradientF(N) ) N(N)*  ]
}}  {{d*}} +([[(t( {{d}} ) {{ INT[ t( N(N) ) N(N)*  ] }}  {{d*}} )]])+t( {{δd}} ) {{ INT[ t( N(N) ) N(N)*  ]
}}  {{d*}} +t( {{δd}} ) {{ INT[ t( DFGradientF(N) ) N(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT[ t( FGradientD
F(N) ) N(N)*  ] }}  {{d*}} +([[(t( {{δd}} ) {{ INT[ t( N(N) ) N(N)*  ] }}  {{d*}} )]])+t( {{d}} ) {{ INT[  t(
 N,t(N) ) k   N,t(N)*  ] }}  {{d*}} +t( {{d}} ) {{ INT[  t( N,t(N) ) k   A(N)*  ] }}  {{d*}} +t( {{d}} ) {{ INT
[  t( DFGradientF(N) ) k   N,t(N)*  ] }}  {{d*}} +t( {{d}} ) {{ INT[  t( DFGradientF(N) ) k   A(N)*  ] }}  {{d
*}} +t( {{δd}} ) {{ INT[  t( N(N) ) k   N,t(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT[  t( N(N) ) k   A(N)*  ] }}  {
{d*}} +t( {{d}} ) {{ INT[  t( N,t(N) ) k   A(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT[  t( DFGradientF(N) ) k
N,t(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT[  t( FGradientDF(N) ) k   N,t(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT
[  t( DFGradientF(N) ) k   A(N)*  ] }}  {{d*}} +t( {{δd}} ) {{ INT[  t( FGradientDF(N) ) k   A(N)*  ] }}  {{d
*}} +t( {{δd}} ) {{ INT[  t( DFGradientF(N) ) k   A(N)*  ] }}  {{d*}} ) = (0)

Line 11:
( {{K}}  {{ {{d}} }} + {{Ktan}}  {{δ {{δd}} }} +++[[ {{ (d) }} ]]) = (0)
```

Figure 88 Derivation of a formulation for a pure 1-D advection equation

6.5.3 Numerical code generated for nonlinear formulation

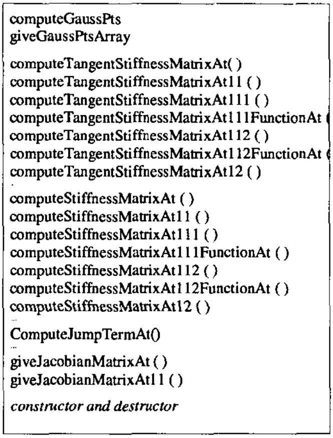
As seen before, the code is introduced into the object-oriented finite element code FEM_Object (see [DUB 93]). Details about the enhancing of the code to nonlinear Newton like algorithms is discussed in [EYH 96c]. The hierarchy showing the new classes added to FEM_Object is shown in Figure 89. The methods shown in Figure 90 is added to class **NewElement**. Note that this new element is an element using the DSD/ST procedure (a subclass of **STF_ELEMENT** that has a method to compute the "jump term"). The new component added here is the method *computeTangentStiffnessMatrixAt: stepN*, needed to compute the elemental contribution to the tangent modulus.

The class **NewElementGaussPoint** is automatically added as a subclass of **GaussPoint**. In this case, the integration point, at which all the computations are made, has the capability to manage and store the computation of the scalar *u* and its gradient. This enhances the efficiency of nonlinear iterations.



```
FEMObject
        Dictionary
        Dof
        Domain
        FEMComponent
                Element
                        PlainStrain
                        STF ELEMENT
                                NewElement
                Load
                Node
                TimeItegrationScheme
                TimeStep
        LinearSystem
        Tools
                GaussPoint
                        NewElementGaussPoint
```

Figure 89 Partial view of classes added for a nonlinear problem



```
computeGaussPts
giveGaussPtsArray

computeTangentStiffnessMatrixAt( )
computeTangentStiffnessMatrixAt11 ( )
computeTangentStiffnessMatrixAt111 ( )
computeTangentStiffnessMatrixAt111FunctionAt (
computeTangentStiffnessMatrixAt112 ( )
computeTangentStiffnessMatrixAt112FunctionAt (
computeTangentStiffnessMatrixAt12 ( )

computeStiffnessMatrixAt ( )
computeStiffnessMatrixAt11 ( )
computeStiffnessMatrixAt111 ( )
computeStiffnessMatrixAt111FunctionAt ( )
computeStiffnessMatrixAt112 ( )
computeStiffnessMatrixAt112FunctionAt ( )
computeStiffnessMatrixAt12 ( )

ComputeJumpTermAt()

giveJacobianMatrixAt ( )
giveJacobianMatrixAt11 ( )

constructor and destructor
```

Figure 90 Partial view of the methods added to class **NewElement**

NewElementGaussPoint
subclass of **GaussPoint**

Attributes :
- GradientU
- ScalarU

Methods :
- giveGradientU( )
- giveScalarU( )

Figure 91 Partial description of the class **NewElementGaussPoint**

6.5.4 Numerical results

The tests are performed using as initial condition a smooth discontinuity on the first quarter of the domain [0,1] as shown in Figure 92. Space is decomposed into 20 elements, the height of the time slab isΔ *t=0.05*. The boundary conditions are $u(0) = 1$ and $u(1) = 0$. The test is done on a fixed mesh, i.e. the DSD/ST procedure is not applied here. The results are shown in Figure 93; they are in concordance with the theoretical ones, i.e. one can observe the development of the shock wave, and its displacement (see e.g. [ZIE 91]). It is interesting to note that this formulation is able to catch and to follow the shock wave without any artifact.

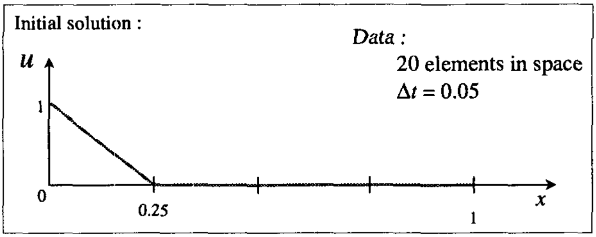


Figure 92 Description of the numerical test for 1-D nonlinear advection equation

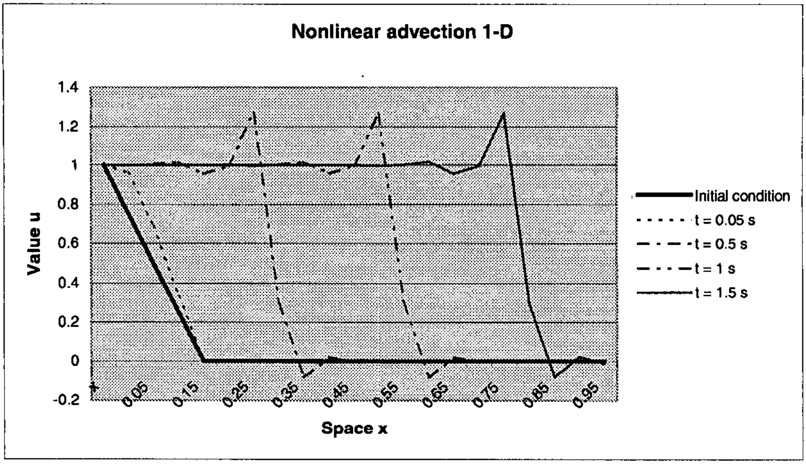


Figure 93 Numerical results for the nonlinear 1-D advection equation

6.6 Towards a general purpose environment for nonlinear problems

At this point we have achieved a basic environment capable of deriving various kinds of nonlinear problems. The approach adopted in an object-oriented way relies on three main features:

- a description of nonlinear terms in the formulations
- a procedure of linearization
- a procedure for automatic coding

Each of them can easily be modified and extended to the solution of new problems. The application of the Object-Oriented paradigm to nonlinear problems can be considered to be successful. Navier-Stokes equations could be tackled following similar procedures as used in Chapters 5 and 6; the only programming requirements are the matrix forms corresponding to the new spatial differential operators.

Chapter 7 Application to the solution of the Navier-Stokes problem

7.1 Enhancement to solve Navier-Stokes equation

In Chapter 5, various strategies for the solution of advection dominated equation models were presented. Stabilization schemes and space-time formulations discontinuous in time were introduced into the FEM_Theory environment. In chapter 6, a linearization scheme was added as a new tool in FEM_Theory. In this chapter, the environment is now enhanced to support the derivation of Navier-Stokes problems. As partially shown in the previous chapter, the equations can be represented and manipulated in FEM_Theory; this is true also for the Navier-Stokes equations. The environment simply needs a generalization of the discretization scheme for the advection part of the equations, i.e. $u \cdot \nabla u$ where u is the velocity. First, it is necessary to enhance its capacities to recognize the new differential operators introduced with the equations. Second, the corresponding discrete operators have to be introduced as subclasses of class **FEMTheorySpatialDifferentialOperators**. This represents a slight change in the hierarchy as shown in Figure 94. So, the methods of the object 'product' (class **ProdList**) to determine the differential operators applied to the trial solution and to the weighting function are generalized to be able to recognize the advective operator; the elemental forms corresponding to these operators are added here for a vector field such as it was done in the previous chapter for the scalar field. The new class **UGradDU** corresponds to the matrix form of the differential operator $\ell_u^1(a) = u \cdot \nabla a$, where u is a given vector and a the vector on which the operator is applied; the class **DUGradU** corresponds to the matrix form of the differential operator $\ell_u^2(a) = a \cdot \nabla u$. The discrete operators are described on Table 20.

Notice that the same differential operators can be used for a space-time formulation. Two examples of formulations follow.

Table 20 Discrete forms of advection operator in 2D illustrated on a four-node bilinear element

Given a vector u :

$$u : \begin{vmatrix} u_1 \\ u_2 \end{vmatrix}$$

Elemental contribution to the operator $\ell_u^1(a) = u \cdot \nabla a$: $[\ell_u^1(a)]_e^h = H^1 d$

where $H^1 = [H_1^1, H_2^1, H_3^1, H_4^1]$ and d is the vector of nodal contribution for field a,

with $H_i^1 = \begin{bmatrix} u_{1,1} N_i & u_{1,2} N_i \\ u_{2,1} N_i & u_{2,2} N_i \end{bmatrix}$

Elemental contribution to the operator $\ell_u^2(a) = a \cdot \nabla u$: $[\ell_u^2(a)]_e^h = H^2 d$

where $H^2 = [H_1^2, H_2^2, H_3^2, H_4^2]$ and d is the vector of nodal contribution for field a,

with: $H_i^2 = \begin{bmatrix} u_1 N_{i,1} + u_2 N_{i,2} & 0 \\ 0 & u_1 N_{i,1} + u_2 N_{i,2} \end{bmatrix}$

NB: N_1, N_2, N_3 and N_4 are the classical bilinear functions

```
Object
...
    Collection
                    ...
                    FEMTheoryOrderedCollection
                        EquationsCollection
                        ExpressionLists
                            Prodlist
                            SumList
...
    FEMTheory
        FEMTheoryMathematicalStructures
            StructureWithDimension
                Expression
                Functional
                Integral
                Term
        FEMTheorySpatialDifferentialOperators
            SCALAR
                DFGradientF
                FgradientDF
                Gradient
                Scalar
            TENSOR
            VECTOR
                Div
                DivSymmGrad
                DUGradU
                Grad
                SymmGrad
                UgradDU
                Vector
        GeometryReference
            NpbDim2
                Q4
            NpbDim3
        Unit
```

NB: Changes are highlighted in gray; new classes are in bold italic

Figure 94 Illustration of the changes into the hierarchy of FEM_Theory for Navier-Stokes

7.2 A stabilized formulation for the steady state Navier-Stokes problem

This formulation is derived in order to check the changes described in the previous section. The strong form of the problem is posted on Table 21, the proposed stabilized approximated formulation on Table 22 (see Chapter 3 for all the notations used) . The stabilization scheme adopted here is SUPG/PSPG suggested in [TEZ 92a, TEZ 92c] (see Chapter 5 for more details).

The derivation is lead in 14 steps; it is quite identical with the ones performed in previous chapters (see screen of FEM_Theory in Figure 95). Notice that here, as in the derivation depicted in chapter 6, a consistent linearization is performed; but some terms are dropped out afterwards in the gradient part for the sake of the simplicity of the numerical computation. A classical Q4 element is chosen. The velocity and pressure fields are interpolated at the four corner nodes. A 2 by 2 Gauss quadrature is adopted. The design of stabilization parameters is the one proposed in [TEZ 92c] in which the time dependent term is omitted. For the sake of simplicity, the stabilization part coming from the continuity equation is also omitted. This can be done for low Reynolds numbers (see the derivation of Stokes flow in Chapter 3). It is important to notice that this derivation scheme allows us to adopt various strategies in the numerical computation. These results were obtained using a 'ramping' iterative scheme, i.e. by increasing slowly the Reynolds number from a steady stokes flow. At each increment, the Reynolds number is increased and convergence is achieved. The first iterations are Picard type iterations (the tangent stiffness matrix is replaced by the stiffness matrix). This makes it possible first to check the convergence and second to ensure the convergence at each value of the Reynolds number. The following iterations are modified Newton type ones (the tangent stiffness matrix is the non-consistent one obtained through the derivation). This shows the flexibility of the code generated automatically. Numerical results obtained on the example of the cavity flow problem for a 32 by 32 mesh are shown in Figure 96 and Figure 97. Results are comparable to existing ones (for example [GHI 82] and [SCH 83], or [TEZ 92c] and [STO 97] for similar computations).

Table 21 Strong form for the steady state Navier-Stokes problem

Find u velocity and p pressure with appropriate continuity requirements, such that :

Ω in $\Re^{n_{sd}}$

$n_{sd} = 2$

The momentum equation :

$\sigma_{ij,j} + f_i = \rho\, u_j u_{i,j}$ on Ω

The continuity equation :

$u_{i,i} = 0$ on Ω

The boundary conditions :

$\sigma_{ij} n_j = F_i$ on $\partial_2\Omega$

$u_i = \bar{u}_i$ on $\partial_1\Omega$

with $\partial\Omega = \partial_1\Omega \cup \partial_2\Omega$

The constitutive equation :

$\sigma_{ij} = -p\,\delta_{ij} + 2\mu\,\varepsilon_{ij}(u)$ on Ω

With the kinematics law :

$\varepsilon_{kl}(u) = \frac{1}{2}(u_{k,l} + u_{l,k})$ on Ω

Table 22 Approximated stabilized formulation for the steady state Navier-Stokes problem

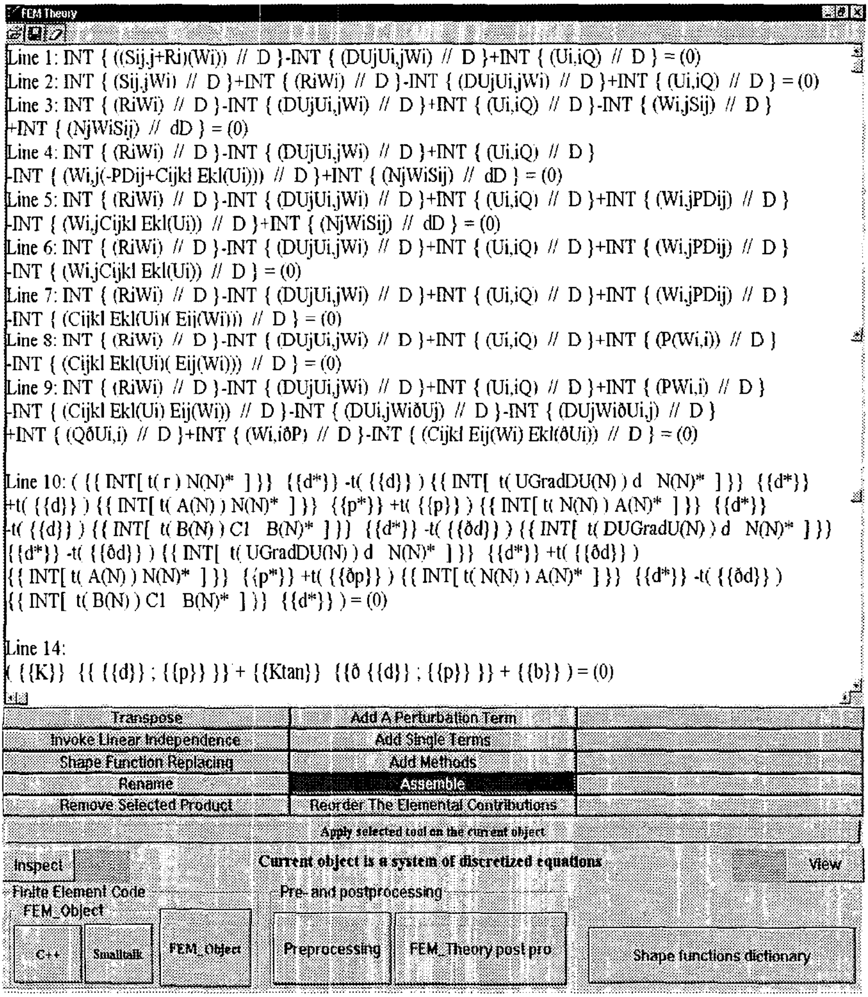
Given f, find $(u^h, p^h) \subset (\mathcal{S}^h \times \mathcal{P}^h)$ such that for all $(w^h, q^h) \subset (\mathcal{W}^h \times \mathcal{P}^h)$:

$$\int_\Omega \rho\, u_j^h u_{i,j}^h w_i^h dv - \int_\Omega 2\mu\varepsilon_{ij}(u^h)\varepsilon_{ij}(w^h) dv + \int_\Omega p_h w_{i,i}^h dv + \int_\Omega u_{i,i}^h q^h dv - \int_\Omega f_i w_i^h dv$$

$$+ \sum_{\Omega^e \in \Omega^h} \left[\int_{\Omega^e} (\rho\, u_j^h u_{i,j}^h - 2\mu\varepsilon_{ij,j}(u^h) + p_{,i}^h - f_i)\, \tau_{mom}\, (\rho\, u_j^h w_{i,j}^h - 2\mu\varepsilon_{ij,j}(w^h) + q_{,i}^h) dv \right] = 0$$

with:

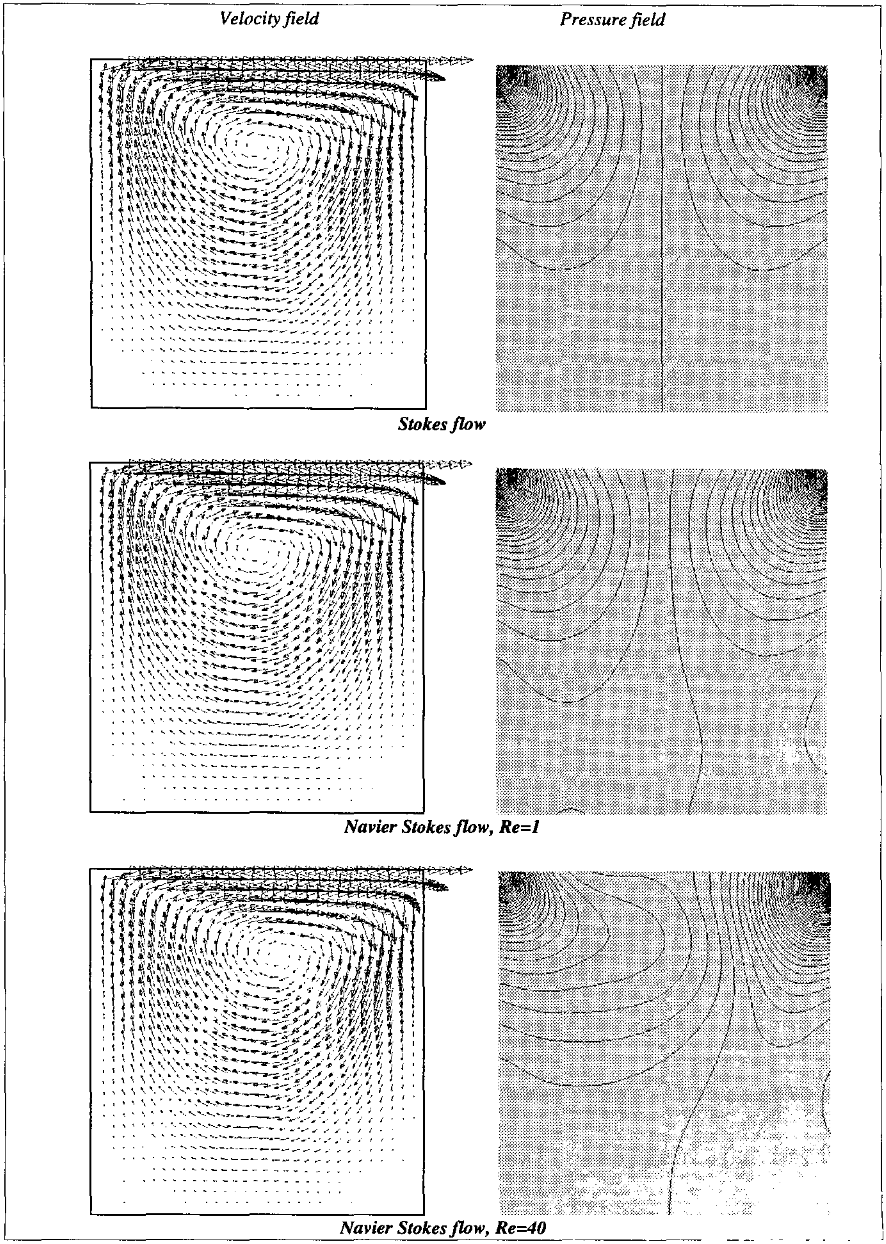
$$\tau_{mom} = \left(\left(\frac{2|u|}{h} \right)^2 + \left(\frac{4\mu}{h^2} \right)^2 \right)^{-1/2}$$ (for details about stabilization see e.g. [BEH 94])



```
Line 1: INT { ((Sij,j+Ri)(Wi)) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D } = (0)
Line 2: INT { (Sij,jWi) // D }+INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D } = (0)
Line 3: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }-INT { (Wi,jSij) // D }
+INT { (NjWiSij) // dD } = (0)
Line 4: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }
-INT { (Wi,j(-PDij+Cijkl Ekl(Ui))) // D }+INT { (NjWiSij) // dD } = (0)
Line 5: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }+INT { (Wi,jPDij) // D }
-INT { (Wi,jCijkl Ekl(Ui)) // D }+INT { (NjWiSij) // dD } = (0)
Line 6: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }+INT { (Wi,jPDij) // D }
-INT { (Wi,jCijkl Ekl(Ui)) // D } = (0)
Line 7: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }+INT { (Wi,jPDij) // D }
-INT { (Cijkl Ekl(Ui)( Eij(Wi))) // D } = (0)
Line 8: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }+INT { (P(Wi,i)) // D }
-INT { (Cijkl Ekl(Ui)( Eij(Wi))) // D } = (0)
Line 9: INT { (RiWi) // D }-INT { (DUjUi,jWi) // D }+INT { (Ui,iQ) // D }+INT { (PWi,i) // D }
-INT { (Cijkl Ekl(Ui) Eij(Wi)) // D }-INT { (DUi,jWiðUj) // D }-INT { (DUjWiðUi,j) // D }
+INT { (QðUi,i) // D }+INT { (Wi,iðP) // D }-INT { (Cijkl Eij(Wi) Ekl(ðUi)) // D } = (0)

Line 10: ( {{ INT[ t( r ) N(N)*  ]}}  {{d*}} -t( {{d}} ) {{ INT[  t( UGradDU(N) ) d   N(N)*  ]}}  {{d*}}
+t( {{d}} ) {{ INT[ t( A(N) ) N(N)*  ]}}  {{p*}} +t( {{p}} ) {{ INT[ t( N(N) ) A(N)*  ]}}  {{d*}}
-t( {{d}} ) {{ INT[  t( B(N) ) C1   B(N)*  ]}}  {{d*}} -t( {{ðd}} ) {{ INT[  t( DUGradU(N) ) d   N(N)*  ]}}
{{d*}} -t( {{ðd}} ) {{ INT[  t( UGradDU(N) ) d   N(N)*  ]}}  {{d*}} +t( {{ðd}} )
{{ INT[ t( A(N) ) N(N)*  ]}}  {{p*}} +t( {{ðp}} ) {{ INT[ t( N(N) ) A(N)*  ]}}  {{d*}} -t( {{ðd}} )
{{ INT[  t( B(N) ) C1   B(N)*  ]}}  {{d*}} ) = (0)

Line 14:
( {{K}}  {{ {{d}} ; {{p}} }} + {{Ktan}}  {{ð {{d}} ; {{p}} }} + {{b}} ) = (0)
```

Current object is a system of discretized equations

Figure 95 Derivation for the steady state Navier Stokes problem in FEM_Theory



Velocity field

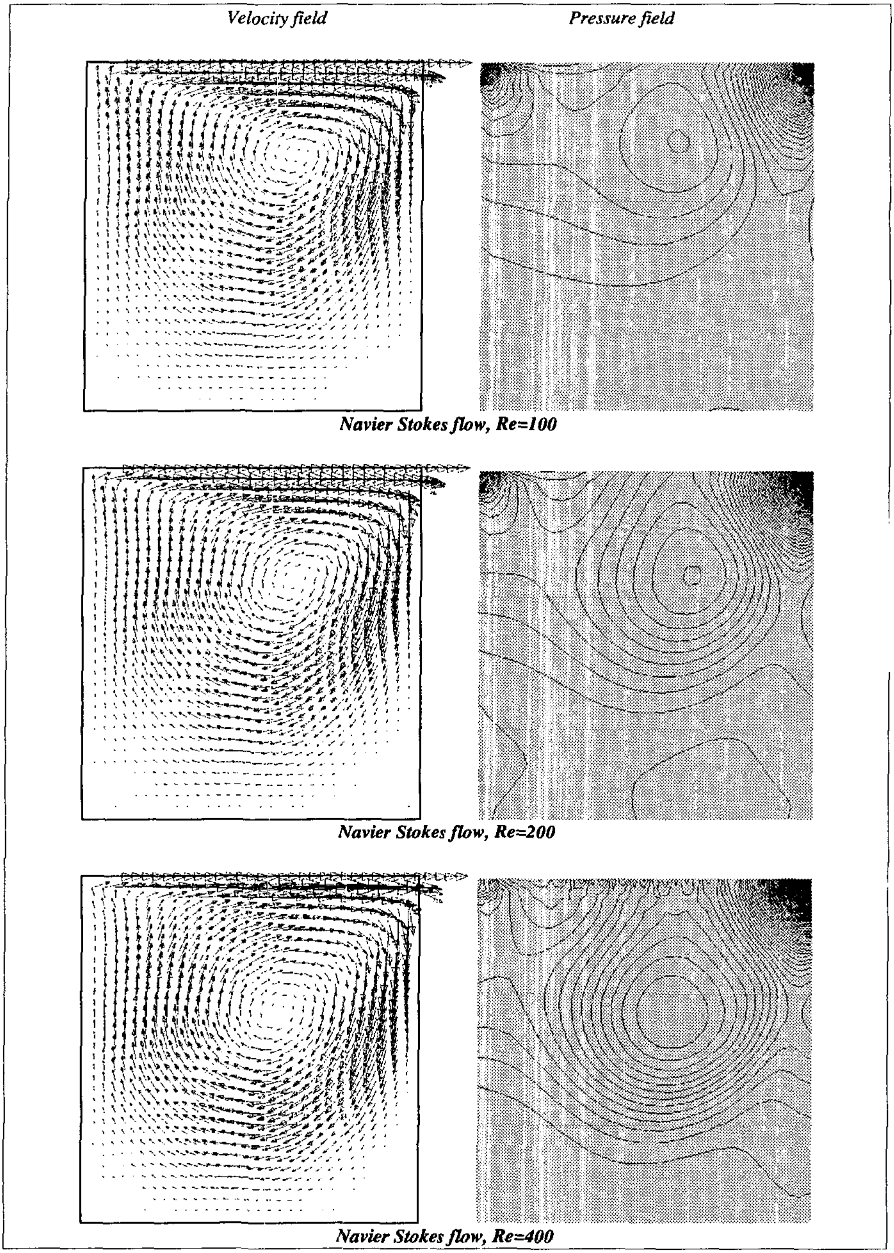
Pressure field

Stokes flow

Navier Stokes flow, Re=1

Navier Stokes flow, Re=40

Figure 96 Numerical results for the cavity flow problem at various Reynolds numbers (continued next figure)



Velocity field

Pressure field

Navier Stokes flow, Re=100

Navier Stokes flow, Re=200

Navier Stokes flow, Re=400

Figure 97 Numerical results for the cavity flow problem at various Reynolds numbers

7.3 A stabilized space-time formulation discontinuous in time for Navier-Stokes problem : Application to the dam break problem

7.3.1 A space-time formulation for the Navier-Stokes problem with moving boundaries

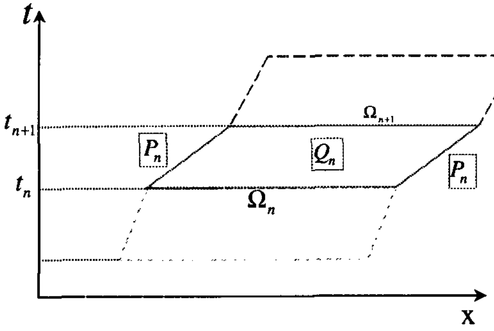
The strong statement of the unsteady Navier-Stokes problem is shown on Table 23, and the formulation chosen here on Table 24. The formulation was first proposed in [TEZ 92 b]; this is the multi-dimensional form of the formulation described in Chapter 6. In the variational formulation of Table 24 (see Chapter 4 and [BEH 94] for a detailed description of the notations), the stabilization terms, third and forth integrals, are the ones described in [TEZ 92c}. The first one is a SUPG/PSPG stabilization term, the second corresponds to the continuity equation. Ensuing previous derivations, no new fundamental concepts are introduced at this stage of the mathematical formulation. As matter of fact, the derivation can be led in the symbolic environment.

A classical 8-nodes brick element is chosen. The velocity and pressure fields are interpolated at each node. Linear shape functions are used for the interpolation and for the kinematics of the element. A 2*2*2 Gaussian integration is adopted. Only the characteristics specific to this 3-D element needed at this stage have to be introduced. First, the way to perform a numerical surface integration on the sides is added to the automatic programming scheme. Second the dictionary of shape functions is enriched with linear shape functions. The element is shown in Figure 98.

Table 23 Strong statement for the time dependent Navier-Stokes problem

Given f and F, find u velocity and p pressure with appropriate conditions of continuity such that:	
Ωin $R^{n_{sd}}$	
$n_{sd} = 2$	
The momentum equation :	
$\sigma_{ij,j} + f_i = \rho\, u_{i,t} + \rho\, u_j u_{i,j}$	on $\Omega \times T$
The continuity equation :	
$u_{i,i} = 0$	on $\Omega \times T$
The boundary conditions :	
$\sigma_{ij} n_j = F_i$	on $\partial_2 \Omega \times T$
$u_i = \overline{u}_i$	on $\partial_1 \Omega \times T$
with $\partial\Omega = \partial_1\Omega \cup \partial_2\Omega$	
The constitutive equation :	
$\sigma_{ij} = -p\,\delta_{ij} + 2\mu\, \varepsilon_{ij}(u)$	on $\Omega \times T$
The initial conditions :	
$u(t=0,x) = u_0$	on Ω
$u_{i,i}\,(t=0,x) = 0$	on Ω
With the kinematics law :	
$\varepsilon_{kl}(u) = \frac{1}{2}(u_{k,l} + u_{l,k})$	on $\Omega \times T$

Table 24 A space-time formulation for Navier-Stokes problem



Given f , for each time slab $[t_n, t_{n+1}]$, find $(u^h, p^h) \subset ((\mathcal{S}^h)_n \times (\mathcal{P}^h)_n)$ such that for all $(w^h, q^h) \subset ((\mathcal{W}^h)_n \times (\mathcal{P}^h)_n)$:

$$\int_{Q_n} (\rho\, u^h_{i,t} + \rho\, u^h_j u^h_{i,j}) w^h_i dv - \int_{Q_n} 2\mu\varepsilon_{ij}(u^h)\varepsilon_{ij}(w^h) dv + \int_{Q_n} p_h w^h_{i,i} dv + \int_{Q_n} u^h_{i,i} q^h dv - \int_{Q_n} f_i w^h_i dv$$

$$+ \sum_{\Omega^e \in \Omega^h} \left[\int_{\Omega^e} (\rho\, u^h_{i,t} + \rho\, u^h_j u^h_{i,j} - 2\mu\varepsilon_{ij,j}(u^h) + p^h_{,i} - f_i) \tau_{mom} (\rho\, w^h_{i,t} + \rho\, u^h_j w^h_{i,j} - 2\mu\varepsilon_{ij,j}(w^h) + q^h_{,i}) dv \right]$$

$$+ \sum_{\Omega^e \in \Omega^h} \left[\int_{\Omega^e} u^h_{i,i} \tau_{cont} w^h_{i,i} dv \right] + \int_{\Omega_n} \rho\ [[u^h]](w^h)^+_n dv = 0$$

where :

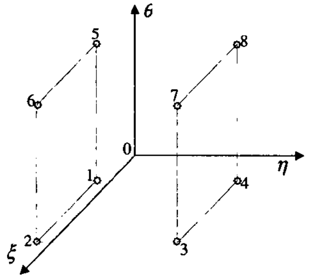
$$(\mathcal{S}^h)_n = \{ u^h \in [H^1(Q_n)]^h \mid u^h = \bar{u} \quad \text{on} \, (P_n)_{\bar{u}} \}$$

$$(\mathcal{W}^h)_n = \{ u^h \in [H^1(Q_n)]^h \mid u^h = 0 \quad \text{on} \, (P_n)_{\bar{u}} \}$$

$$(\mathcal{P}^h)_n = \{ p^h \in [L_2(Q_n)]^h \}$$

with :

$\tau_{mom} = \left(\left(\frac{2|u|}{h} \right)^2 + \left(\frac{4\mu}{h^2} \right)^2 \right)^{-1/2}$ and τ_{cont} defined as in [BEH 94].



$$\begin{array}{ll}
N_1 = -\frac{1}{8}(\xi-1)(\eta-1)(\theta-1) & N_5 = \frac{1}{8}(\xi-1)(\eta-1)(\theta+1) \\
N_2 = \frac{1}{8}(\xi+1)(\eta-1)(\theta-1) & N_6 = -\frac{1}{8}(\xi+1)(\eta-1)(\theta+1) \\
N_3 = -\frac{1}{8}(\xi+1)(\eta+1)(\theta-1) & N_7 = \frac{1}{8}(\xi+1)(\eta+1)(\theta+1) \\
N_4 = \frac{1}{8}(\xi-1)(\eta+1)(\theta-1) & N_8 = -\frac{1}{8}(\xi-1)(\eta+1)(\theta+1)
\end{array}$$

Figure 98 Description of the 8-nodes 3-D linear reference element

7.3.2 Application to the dam break problem

The moving boundaries scheme is put in prominent position on the dam break problem. The description of the problem is given in Figure 99; here we take $\mu = 0.001$, $\rho = 1$ and the time step is 0.1. The results are shown at various time values (Figure 100 and Figure 101). The deformed mesh, isolines of pressure and velocities are given. Notice that the mesh is updated at each iteration by using the total velocity. A comparison with existing similar results [HAN 92b] and experimental results [MAR 52] is made on the advancing of the front (see Figure 102). These results confirm the potential of this numerical scheme.

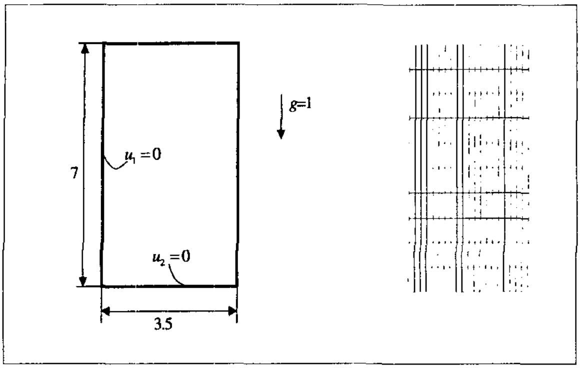


Figure 99 Description of the dam break problem and 2D view of the mesh

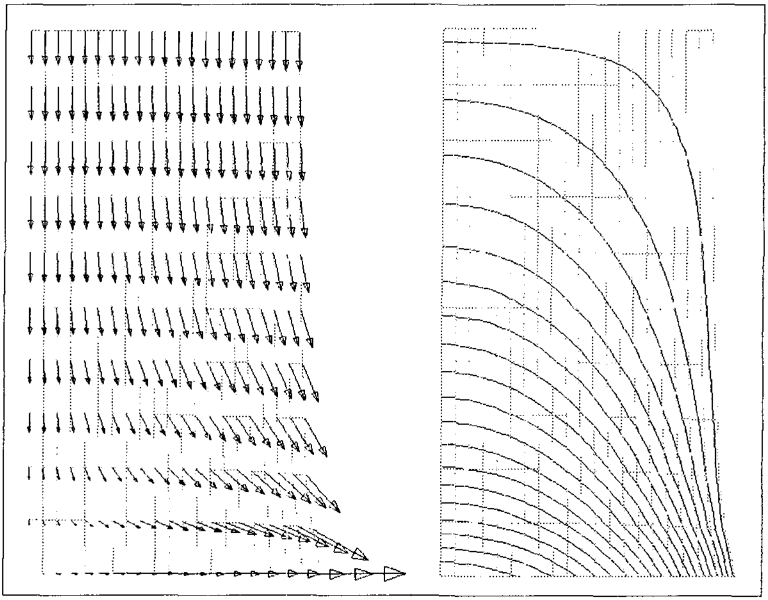


Figure 100 Vector and pressure fields for the dam break problem at *t=0.5 s*

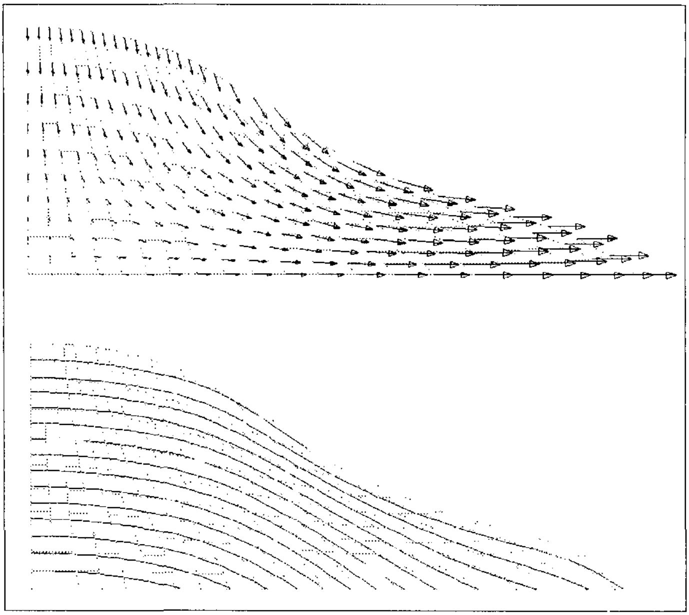


Figure 101 Vector and pressure fields for the dam break problem at *t=3 s*

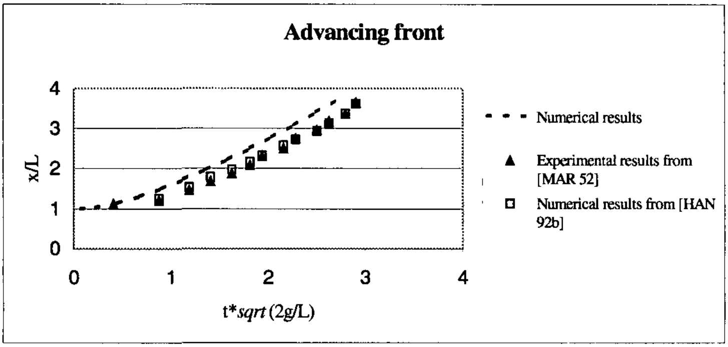


Figure 102 Comparison of the advancing front

7.3.3 Closer to the mechanics

This example of derivation shows the fast and natural extendibility capabilities of the symbolic environment. The only extensions needed for this formulation are the generalization of the scheme to handle correctly the spatial differential operators, the introduction of the new matrix forms for the spatial differential operators and the enrichment of the shape functions data base. These changes are taken into account at a high level of abstraction, very close to the mathematical formulation. The initial aim of handling new formulations at a level closer to the original mathematical level can be considered as successfully achieved.

Chapter 8 Conclusion

8.1 A brief overview

In this work, a computerized framework for the derivations of finite element formulations was presented. This work is based on a hybrid symbolic/numerical approach. An environment based on the object oriented paradigm, capable of performing symbolic manipulations was built. Based on a thorough analysis of a Galerkin formulation applied to elastodynamics, a symbolic environment was developed in which natural concepts for continuum problems such as term, sum, product, expression, integral, variational formulation, directional derivative and system can be manipulated. Equivalent concepts for the discrete problem and its manipulations were also created, like elemental forms, discretized expressions, etc... At this stage already all the parameters of classical finite elements are taken into account for the treatment of linear initial boundary value problems. In such a context, any usual finite element model can be constructed, and the cumbersome computations and manipulations inherent to this type of approach are left to the computer. A simple interactive graphical object oriented interface closely related to the description of the objects facilitates their manipulations or, more precisely, the communication with them. Numerical computations are performed in a classical object oriented finite element code. The link between both symbolic and numerical worlds, is achieved using an object oriented concept for the automatic programming of elemental forms. The implementation of the symbolic environment is integrated into a Smalltalk environment; and generation of a finite element code is possible either in Smalltalk or in C++, the latter permitting to achieve a relatively high numerical efficiency and thus allowing interesting numerical tests to be performed.

The approach was tested on various mechanical problems including nonlinear ones: heat diffusion, linear elasticity in statics and dynamics, Stokes flow in the incompressible limit, Navier-Stokes flow in the incompressible limit. Various finite element formulations were used on these problems: like classical Galerkin formulations, Galerkin least-squares formulations, Galerkin space-time formulations, continuous in space-discontinuous in time.

8.2 Analysis of the genericity

The main objective of the thesis was to develop a generic environment in order to be able to treat a large number of different problems. To achieve this, an environment was developed to represent the various formulations; this led, roughly speaking, to using concepts of sums of products, where the term is the basic entity. The index notation was adopted because it leads to a simple visual representation of the different manipulations; as a result, the integration of a new type of problems is straightforward. Take the example of a classical electromagnetic problem; the problem is governed by the following equations (see e.g. [CHA 80]):

$$\left|\begin{array}{l} \operatorname{curl} E = -\frac{\partial B}{\partial t} \\ \operatorname{curl} H = J + \frac{\partial D}{\partial t} \\ \operatorname{div} D = \rho \\ \operatorname{div} B = 0 \end{array}\right. \text{ with additional constitutive relations: } \left|\begin{array}{l} D = \Re_e(E) \\ B = \Re_m(H) \\ \text{Ohm's law :} \\ J = \Re_o(E) \end{array}\right.$$

where E and H are respectively the electric and magnetic fields, D and B are the electric and magnetic flux vectors, J is the electric current density and ρ is the electric charge density. Note that vectors D and B are related to the electric and magnetic fields E and H. The treatment of these equations requires a new operator, *curl*. Moreover, the direct solution of these equations is rarely attempted; they are usually combined. To adapt the symbolic environment to this type of problem a few additions are necessary, in order to represent the equations correctly. The operator *curl* is expressed using index notation: $\operatorname{curl} A : e_{ijk} A_{j,k}$ where $e_{ijk} = 0$ if two indices are the same, $e_{ijk} = 1$ if i, j and k are permutations of 1,2, 3, $e_{ijk} = -1$ if i, j and k are permutations of 1,3,2. Adding a new object to represent e_{ijk} in the environment permits to represent the new operator. This new object will also permit the representation of the cross-product: $A \times B : e_{ijk} A_j B_k$. In the same manner, an object to represent the Dirac function (δ_{ij}) could be added. It can be deduced from this example that the representation of new types of problems can be made easily in the symbolic environment, although minor extensions to the environment may be needed. Moreover, new manipulation tools similar to integration by parts could be added to facilitate the manipulations of this new type of equations. Further, the finite element approximation will necessitate enhancements in order to recognize the new differential operators and the library will be enriched with corresponding discrete operators. Extension to alternative weighted residual methods (e.g. collocation) could be done as well, by giving a meaning to the weighting function (a Dirac function in the case of collocation).

The choice of the index notation as mode of representation of partial differential equations is thus shown to be pertinent, and the use of the object-oriented, the second challenging choice in this work, is very convenient for the overall approach. Although it is obviously impossible to build an environment capable of dealing with all types of problems, we have given a proof that the proposed environment is easily adaptable.

The choice to integrate the new elements into an existing code was necessary to give a proof of feasibility. The automatic programming schemes remain intimately related to the target finite element code; but, the concepts of automatic finite element programming are general. To achieve a complete generalization of the programming scheme, it would be necessary to integrate part of the components of the numerical code into the symbolic environment.

At this stage of the development, the prototype can still be improved. In order to get a more user-friendly environment, e.g. for educational purposes, it would be necessary to consolidate both the domain of application of the different available tools, and the graphical interface. The efficiency of the generated code could be improved too, e.g. by adding temporary variables in automatically programmed expressions and by improving symbolic computations.

8.3 Towards a general purpose environment for finite elements developments

This work represents an important step towards a general environment for easy development of computerized solution schemes for mechanical problems. This prototype could be enriched with concepts of logic programming techniques to help the user in decision making. The environment is still limited, at this stage, to the introduction of finite element matrices. Extension to the algorithmic description of finite element formulations should allow the introduction of new solution schemes, e.g. new time integration schemes, strategies for updating variables, strategies for updating meshes or remeshing, etc... This is a particularly crucial point for nonlinear finite element analysis of coupled systems. As a consequence, enhancement to strategies such as parallelism, an important ingredient for high performance computations, would be natural. This could be done either in the symbolic part or in the numerical part of the environment. In order to achieve an optimal fast prototyping tool of finite element model solutions, it would necessary to couple the symbolic and the numerical environment with flexible pre- and post-processors, in order to facilitate data structure generation and to post-process the results. Finally, the application of the ideas developed in this study could be extended to alternative numerical solution schemes for partial differential equations based on weighted residual methods such as e.g. collocation or the boundary integral method.

References

[ANG 92] I.G. Angus, Parallelism, Object-Oriented Programming methods, Portable software and C++, Proceedings of 8th conf. held in conjunction with AEC Systems 92, Dallas, June 7-9 1992, Ed. Barry, Goodno and Wright, ASCE (1992) pp. 506-513.

[BAI 93] C. Baiocchi, F. Brezzi, and L. P. Franca, Virtual bubbles and Galerkin-least-squares type methods (Ga.L.S), Comput. Methods Appl. Mech. Engrg., vol. 105 (1993) pp. 125-141.

[BAR 89] N.S. Bardel, The application of symbolic computing to the hierarchical finite element method, Internat. J. Numer. Methods Engrg., vol. 28 (1989) pp. 1181-1204.

[BAR 92] C. Barbier, Automatic generation of bending element matrices for finite element method using REDUCE, Engineering Computations, vol. 9 (1992) pp. 477-494.

[BAT 82] K. J. Bathe, Finite Element procedures in engineering analysis, Prentice-Hall, (1982).

[BAU 92] J.W. Baugh and D.R. Rehak, Data abstraction in engineering software development, Comp. Civ. Engr., vol. 6 (1992) pp. 282-299.

[BEH 92] M. Behr, Stabilized finite element methods for incompressible flows with emphasis on moving boundaries and interfaces, Ph.D thesis report, University of Minnesota (1992).

[BEH 93] M. Behr, L.P. Franca and T.E. Tezduyar, Stabilized finite methods for the velocity-pressure-stress formulation of incompressible flows, Comput. Methods Appl. Mech. Engrg., 104 (1993) pp. 31-48.

[BEH 94] M. Behr and T.E. Tezduyar, Finite element solution strategies for large-scale flow simulation, Comput. Methods Appl. Mech. Engrg., 112 (1994) pp. 3-24.

[BES 97] J. Besson and R. Foerch, Large scale object-oriented finite element code design, Comput. Methods Appl. Mech. Engrg., 142 (1997) 165-187.

[BRE 92a] F. Brezzi, M.O. Bristeau, L. P. Franca, M. Mallet and G. Rogé , A relationship between stabilized finite element methods and the Galerkin method with bubble functions, Comput. Methods Appl. Mech. Engrg., vol. 96 (1992) pp. 117-129.

[BRE 92b] P. Breitkopf and G. Touzot, Architecture des logiciels et langages de modélisation, La Revue Européenne des éléments finis, vol. 1 (3) (1992) pp. 333-368.

[BRO 82] A. Brooks and Th. J. R. Hughes, Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., vol. 32 (1982) pp. 199-259.

[BUF 97] M. Buffat, I. Yudiana and C. Leribault, Parallel simulation of turbulent compressible flows with unstructured domain partitioning. Performance on T3D and SP2 using OOP, in Parallel computational fluid dynamics: Algorithm and results Advanced Computers, Eds. Schiano, A. Ecer, J. Periaux and N. Satofuka, Elsevier (1992) pp. 76-83.

[CAM 97] F. Cameron, Automatic generation of efficient routines for evaluating multivariate polynomials arising in finite element computations, Adv. in Engr. Soft., vol. 28 (1997) pp. 239-245.

[CAR 94] A. Cardona, I. Klapka and M. Géradin, Design of a new finite element programming environment, Engineering Computations, vol. 11 (1994) pp. 365-381.

[CEC 77] M.M. Cecchi and C. Lami, Automatic generation of stiffness matrices for finite element analysis, Internat. J. Numer. Methods Engrg., vol. 11 (1977) pp 396-400.

[CHA 80] M.V.K Chari and P.P. Silvester (Eds.), Finite elements in Electric and Magnetic Field problems, John Wiley & Son (1980).

[CHA 88] R. Chambon and J.B. Thomas, Langages pour le calcul des structures, Ed. Fouet, Ladevèze and Ohayon, Pluralis, vol. 2 (1988) pp. 261-271.

[CHO 92] D.K. Choi and S. Nomura, Application of symbolic computation to two-dimensional elasticity, Computers & Structures, vol. 43 (1992) pp 645-649.

[CHO 93] A.J. Chorin and J.E. Marsden, A mathematical introduction to fluid mechanics, Springer-Verlag (1993).

[CLI 89] T. Cline, H. Abelson and W. Harris, Symbolic computing in engineering design, AI EDAM, vol. 3 (1989) pp 195-206.

[COD 94] R, Codina, U. Schäfer and E. Oñate, Mould filling simulation using finite elements, Int. J. Num. Heat Fluid Flow, vol. 4 (1994) 291-310.

[COL 88] E. Collain, J.M. Fouet and G. Regnier, Pour un calcul de structures orienté objets, Ed. Fouet, Ladevèze and Ohayon, Pluralis, vol. 2 (1988) pp. 371-390.

[COM 96] S. Commend and T. Zimmermann, Finite Element Prepocessing with Java, http://dgcwww.epfl.ch/WWWLSC/feppwj.html, (1996).

[CUR 93] I.G. Currie, Fundamental mechanics of fluids, McGraw-Hill (1993).

[DEV 92a] P.R.B. Devloo, C.A. Magalhaes and A.T. Noel, On the implementation of the p-adaptive finite element method using the object oriented programming philosofy, in Numerical methods in engineering and applied sciences, part 1 , CIMNE, Barcelona (1992).

[DEV 92b] P.R.B. Devloo, An object oriented approach to finite element programming (Phase I): a system independent windowing environment for developing interactive scientific programs, Advances in Engineering Software, vol. 14 (1992) pp. 41-46.

[DEV 94] P.R.B. Devloo, Efficient issues in an object oriented programming environment, Proceedings of CST 94 , Athens Greece, vol. Artificial intelligence and object oriented approaches for structural engineering, Civil Comp Press, (1994) pp. 147-151.

[DRO 96] J. Drolet, Towards a cross-platform finite element application framework: A toll to simplify finite element simulations, Proceedings of 1st Structural Specialty Conference , May 29th to June 1st 1996, Edmonton,Canada, (1996).

[DUB 91] Y. Dubois-Pèlerin, P. Bomme and Th. Zimmermann, Object-oriented finite element programming concepts, Proceedings of European conference on new advances in computational structural mechanics, ed. P. Ladevèze and O.C. Zienkiewicz, Elsevier Science Publishers (1991), pp. 95-101.

[DUB 92a] Y. Dubois-Pèlerin, Th. Zimmermann and P. Bomme, Object-oriented finite element programming : II. A prototype program in Smalltalk, Comput. Methods Appl. Mech. Engrg., vol. 98 (1992) pp. 361-397.

[DUB 92b] Y. Dubois-Pèlerin and Th. Zimmermann, Object-Oriented finite element programming : Theory and C++ Implementation for FEM_Object$_{C++}$^{TM} 001 , Elmepress international (1992).

[DUB 93] Y. Dubois-Pèlerin and Th. Zimmermann, Object-oriented finite element programming : III. An efficient implementation in C++, Comput. Methods Appl. Mech. Engrg., vol. 108 (1993) pp. 165-183.

[DUB 95] Y. Dubois-Pélerin and P. Pegon, Object-Oriented programming in nonlinear finite element analysis, Submitted to Computers & Structures (1995).

[DUB 97] Y.D. Dubois-Pélerin and P. Pegon, Improving modularity in object-oriented finite element programming, Commun. numer. methods engin., 13 (1997) pp. 193-198.

[ENG 81] M.S. Engelman, G. Strang and K.J. Bathe, The application of quasi-Newton methods in fluid mechanics, Int. J. Num. Meth. Engr. , vol. 17 (1981) pp. 707-718.

[EYH 94a] D. Eyheramendy and Th. Zimmermann, Object-oriented finite element programming : Beyond fast prototyping, Proceedings of CST 94 , Athens Greece, vol. Artificial intelligence and object oriented approaches for structural engineering, Civil Comp Press, (1994) pp. 121-127.

[EYH 95a] D. Eyheramendy and Th. Zimmermann, Programmation orientée objet appliquée à la méthode des éléments finis : dérivations symboliques, programmation automatique, La Revue Européenne des éléments finis, vol. 4 (1995) pp. 327-360.

[EYH 95b] D. Eyheramendy and Th. Zimmermann, Génération automatique de Code Eléments Finis dans un environnement Orienté Objet, Actes du 2ième Colloque national en calcul des structures de Giens (1995) pp. 717-722.

[EYH 96a] D. Eyheramendy and Th. Zimmermann, Object-oriented finite elements : II. A symbolic environment for automatic programming, Comput. Methods Appl. Mech. Engrg., 132 (1996) pp. 259-276.

[EYH 96b] D. Eyheramendy and Th. Zimmermann, Object-oriented Finite Element Programming : An interactive environment for symbolic derivations, Application to an Initial Boundary Value Problem, Advances in Engineering Software 27 (1996) 3-10.

[EYH 96c] D. Eyheramendy, Activity report (Sept. 95- Mar. 96) – AHPCRC/University of Minnesota, Minneapolis, (1996).

[EYH 96d] D. Eyheramendy and Th. Zimmermann, Communication aux journèes nationales en prog. Orienté obj. Pour E.F., Besançon (1996).

[EYH 97a] D. Eyheramendy and Th. Zimmermann, Dérivations symboliques pour code éléments finis - Application à un problème d'élasticité, Actes du 3ième Colloque national en calcul des structures de Giens, Hermès (1997) pp. 837-842.

[EYH 97b] D. Eyheramendy and Th. Zimmermann, Fonctionnalité d'un environnement orienté objet pour le développement de code éléments finis, Actes du 3ième Colloque national en calcul des structures de Giens, Hermès (1997) pp. 553-558.

[EYH 97c] D. Eyheramendy and Th. Zimmermann, Symbolic derivations and automatic generation of finite elements in an object-oriented environment, Proceedings of the 4th US National Congress on Computational Mechanics, San Francisco, Aug. 6-8 1997.

[EYH 97 d] D. Eyheramendy and Th. Zimmermann, Object-oriented finite elements : III. Theory and application of automatic programming, Comput. Methods Appl. Mech. Engrg., (1997) in press.

[FEN 90] G.L. Fenves, Object-Oriented programming for engineering software development, Engr. with Comp., vol. 6 (1990) pp. 1-15.

[FIL 91] J.S.R.A. Filho and P.R.B. Devloo, Object-Oriented programming in scientific computations : The beginning of a new area, Engineering Computations, vol. 8 (1991) 81-87.

[FLE 91a] C.A.J. Fletcher, Computational techniques for Fluid dynamics, vol. I, Fundamental and general techniques, Springer Series in computational Physics, Springer-Verlag (1991).

[FLE 91b] C.A.J. Fletcher, Computational techniques for Fluid dynamics, vol. II, Specific techniques for different flow categories, Springer Series in computational Physics, Springer-Verlag (1991).

[FOE 96] R. Foerch, Un environnement orienté objet pour la modélisation numérique des matériaux en calcul des structures, Ph.D. thesis report, Ecole Nationale Supérieure des Mines de Paris, (1996).

[FOE 95] R. Foerch, J. Besson, P. Pilvin and G. Cailletaud, Formulation des relations de comportement dans les calculs par éléments finis : approche C++, Actes du 2nd Colloque national en calcul des structures, Giens, Hermès (1995) pp. 547-552.

[FOR 90] B.W.R Forde, R.O. Foschi and S.F Stiemer, Object-Oriented Finite Element Analysis, Computers & Structures, vol. 34 (1990) pp. 355-374.

[FRA 87] L. P. Franca, New mixed finite element methods, Ph.D. thesis report, Stanford University, 1987.

[FRA 88] L. P. Franca and T.J.R. Hughes, Two classes of mixed finite element methods, Comput. Methods Appl. Mech. Engrg., vol. 69 (1988) pp. 89-129.

[FRA 89] L. P. Franca and E. G. Dutra Do Carmo, The Galerkin Gradient Least-Squares Method, Comput. Methods Appl. Mech. Engrg., vol. 74 (1989) pp. 41-54.

[FRA 92a] L. P. Franca, S. L. Frey and Th. J. R. Hughes, Stabilized finite element methods : I. Application to the advective-diffusive model, Comput. Methods Appl. Mech. Engrg., vol. 95 (1992) pp. 253-276.

[FRA 92b] L. P. Franca and S. L. Frey, Stabilized finite element methods : II. The incompressible Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., vol. 99 (1992) pp. 209-233.

[FRA 93a] L. P. Franca and A.L. Madureira, Element diameter free stability parameters for stabilized methods applied to fluids, Comput. Methods Appl. Mech. Engrg., vol. 105 (1993) pp. 395-403.

[FRA 93b] L. P. Franca and T.J.R. Hughes, Convergence analyses of Galerkin least-squares methods for symmetric advective-diffusive forms of the Stokes and Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., vol. 105 (1993) pp. 285-298.

[FRA 94a] L. P. Franca and C. Farhat, Anti-stabilizing effects of bubble functions, Proceedings of the Third Congress on Computational Mechanics (IACM), Chiba Japan, vol. I (1994) pp 1452-1453.

[FRA 94b] L. P. Franca and C. Farhat, On the limitations of bubble functions, Comput. Methods Appl. Mech. Engrg., vol. 117 (1994) pp. 225-230.

[FRA 95] L. P. Franca and C. Farhat, Bubble functions prompt unusual stabilized finite element methods, Comput. Methods Appl. Mech. Engrg., vol. 123 (1995) pp. 299-308.

[FRE 97] R. Frenette, D. Eyheramendy and Th. Zimmermann, Numerical modeling of dam-break type problems for Navier-Stokes and granular flows, Proceedings of 1st international Conference on Debris-flow hazards mitigation : mechanics, prediction, and assessment, San Francisco, Aug. 7-9 1997, Ed. C.L. Chen, (1997) pp. 586-595.

[FRI 92] P. Fritzson and D. Fritzson, The need for high-level programming support in scientific computing applied to mechanical analysis, Computers & Structures, vol. 45 (1992) pp. 387-395.

[GHI 82] U. Ghia, K.N. Ghia and C.T. Shin, High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method, J. Comp. Phy., vol. 48 (1982) pp. 387-411.

[GOL 94] N.A. Golias, T.D. Tsiboukis and E.E. Kriezis, Automatic finite element analysis : Application to the shielding by a spherical shell, Archiv für Elektrotechnik, vol. 77 (1994) 85-93.

[GEL 95] J. C. Gelin and L. Walterthum, Conception d'un logiciel orienté-objets pour la simulation de processus de formage, Actes du 2nd Colloque national en calcul des structures, Giens, Hermès (1995) pp. 552-558.

[GRE 91] P.M. Gresho, Some current CFD issues relevant to the incompressible Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., vol. 87 (1991) pp. 201-252.

[GUN 71] R.H. Gunderson and A. Cetiner, Element stiffness matrix generator, J. Struct. Div., Poceedings of ASCE, January 1971, (1971) pp. 363-375.

[HAN 90] P. Hansbo and A. Szepessy, A velocity-pressure streamline diffusion finite element method for the incompressible Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., vol. 84 (1990) pp. 175-192.

[HAN 92a] P. Hansbo, The characteristic streamline diffusion method for convection-diffusion problems, Comput. Methods Appl. Mech. Engrg., vol. 96 (1992) pp. 239-253.

[HAN 92b] P. Hansbo, The characteristic streamline diffusion method for the time-dependent incompressible Navier-Stokes equations , Comput. Methods Appl. Mech. Engrg., vol. 96 (1992) pp. 239-253.

[HAN 95] S.K. Hannani, M. Stalinas and P. Dupont, Incompressible Navier-Stokes computations with SUPG and GLS formulations – A comparison study, Comput. Methods Appl. Mech. Engrg., vol. 124 (1995) pp. 153-170.

[HAR 92] I. Harari and T.J.R. Hughes, What are *C* and *h*? : Inequalities for the analysis and design of finite element method, Comput. Methods Appl. Mech. Engrg., vol. 97 (1992) pp. 157-192.

[HOA 80] S. V. Hoa and S. Sankar, A computer program for automatic generation of stiffness and mass matrices in finite-element analyis, Computers & Structures, vol. 11 (1980) pp. 147-161.

[HSI 97] S.H. Hsieh and E.D. Sotelino, A message-passing class library C++ for portable parallel programming, Eng. with computers, 13 (1997) 20-34.

[HUE 88] A. Huerta and W.K. Liu, Viscous flow with large free surface motion, Comput. Methods Appl. Mech. Engrg., vol. 69 (1988) pp. 277-324.

[HUG 76] Hughes, T. J. R.; Taylor, R.L. ; Sackman, J.L.; Curnier, A. and Kanoknukulchaï, A finite element method for a class of contact-impact problems, Comput. Methods Appl. Mech. Engrg., 8 (1976) pp. 249-276.

[HUG 78] T.J.R. Hughes and K. S. Pister, Consistent linearization in mechanics of solids and structures, Computers & Structures, vol. 8 (1978) pp. 391-397.

[HUG 79] T.J.R. Hughes, W.K. Liu and A. Brooks, Finite element analysis of incompressible viscous flows by the penalty function formulation, J. Computational Physics, vol. 30 (1979) pp. 1-60.

[HUG 81] T.J.R. Hughes, W.K. Liu and Th. Zimmermann, Lagragian-Eulerian finite element formulation for incompressible viscous flows, Comput. Methods Appl. Mech. Engrg., vol. 29 (1981) pp. 329-349.

[HUG 87] T. J. R. Hughes, The Finite Element Method, Prentice-Hall, (1987).

[HUG 86] T.J.R. Hughes, L.P. Franca and M. Balestra, A new finite element formulation for computational fluid dynamics: V. Circumventing the Babuska-Brezzi condition: A stable Petrov-Galerkin formulation of the stokes problem accomodating equal-order interpolations, Comput. Methods Appl. Mech. Engrg., vol. 59 (1986) pp. 85-99.

[HUG 88a] T.J.R. Hughes and G.M. Hulbert, Space-time finite element methods for elastodynamics : formulations and error estimates, Comput. Methods Appl. Mech. Engrg., 66 (1988) pp. 339-363.

[HUG 88b] T.J.R. Hughes and L. Franca, A mixed finite element formulation for Reissner-Mindlin plate theory: Uniform convergence of all higher-order spaces, Comput. Methods Appl. Mech. Engrg., vol. 67 (1988) pp. 223-240.

[HUG 89] T.J.R. Hughes, L.P. Franca and M. Balestra, A new finite element formulation for computational fluid dynamics: VIII. The Galerkin/Least-squares method for advective-diffusive equations, Comput. Methods Appl. Mech. Engrg., vol. 73 (1989) pp. 173-189.

[IBR 92] A. Ibrahimbegovic, Stress resultant geometrically nonlinear shell theory with drilling rotations. Part I: A consistent formulation, LSC Internal report 92/24, Swiss Federal Institute of Technology, (1992).

[IBR 93] A. Ibrahimbegovic and F. Frey, Geometrically non-linear method of incompatible modes in application to finite elasticity with independent rotations, Int. J. Num. Methods in Eng., 36 (1993) 4185-4200.

[IOA 92] N.I. Ioakimidis, Elementary applications of MATHEMETICA to the solution of elasticity problems by the finite element method, Comput. Methods Appl. Mech. Engrg., vol. 102 (1993) pp. 29-40.

[JER 97] B. Jeremic and S. Sture, Tensor objects in Finite Element Programming, To be published in Int. J. Num. Meth. Engr.,(1997).

[JOH 94] C. Johnson, Numerical solution of partial differential equations by the finite element method, Cambridge University press (1994).

[KAW 95] H. Kawata, S. Yoshimura, G. Yagawa and H. Kawai, Object-oriented system for evaluation of fracture mechanics-Parameters of linear and nonlinear 3D cracks, Proceedings of IECS 95, S.N. Atluri, G. Yagawa, T.A. Cruse (Eds.), vol. 1 (1995) pp. 39-44.

[KOR 79] A.R. Korncoff and S.J. Fenves, Symbolic generation of finite element stiffness matrices, Computers & Structures, vol. 10 (1979) pp. 119-124.

[KOZ 94] I. Kozar and A. Ibrahimbegovic, Finite Element formulation of finite rotation solid element, LSC internal report, Swiss Federal Institute of Technology, (1994).

[LEE 91] H.H. Lee and J.S. Arora, Object-Oriented programming for engineering Applications, Engr. with Comp., vol. 7 (1991) pp. 225-235.

[LEF 91] L. Leff and Y.Y. Yun, The symbolic finite element analysis system, Computers & Structures, vol. 41 (1991) pp. 227-231.

[LOU 87a] A.F.D. Loula, T.J.R. Hughes, L. Franca and I. Miranda, Mixed Petrov-Galerkin methods for the Timoshenko beam problem, Comput. Methods Appl. Mech. Engrg., vol. 63 (1987) pp. 133-154.

[LOU 87b] A.F.D. Loula, T.J.R. Hughes, L. Franca and I. Miranda, Stability, convergence and accuracy of a new mixed finite element method for the circular arch problem, Comput. Methods Appl. Mech. Engrg., vol. 63 (1987) pp. 281-303.

[LUC 92] D. Lucas, B. Dressler and D. Aubry, Object-oriented finite element programming using the ADA language, Numerical Methods in Engineering '92, C. Hirsh and al. (Eds.), (1992) pp. 591-598.

[LUF 71] R.W. Luft, J.M. Roesset and J.J. Connor, Automatic generation of finite element matrices, J. Struct. Div., Poceedings of ASCE, January 1971, (1971) pp. 349-361.

[MAC 92] R.I. Mackie, Object-oriented programming of the finite element method, Int. J. Num. Meth. Engr. , vol. 35 (1992) 425-436.

[MAC 97] R.I. Mackie, Using Objects to handle complexity in Finite Element Software, Eng. with computers, 13 (1997) pp. 99-111.

[MAL 78] D.S. Malkus and T.J.R. Hughes, Mixed finite element methods - Reduced and selective integration techniques : a unification of concepts, Comput. Methods Appl. Mech. Engrg., 15 (1978) pp. 63-81.

[MAR 83] J.E. Marsden and T.J.R. Hughes, Mathematical foundations of elasticity, Prentice-Hall, (1983).

[MAR 52] J.C. Martin and W.J. Moyce, An experimental study of the collapse of liquid columns on a rigid horizontal plane, Philos. Trans. Roy. Soc. London Ser. A244, (1952) pp. 312-324.

[MAS90] G. Masini, A. Napoli, D. Léonard, K. Trombe, Les langages à objets, InterEditions,(1990).

[MAS 97] A. Masud and T.J.R. Hughes, A space-time Galerkin/Least-squares finite element of the Navier-Stokes equations for moving domain problems, Comput. Methods Appl. Mech. Engrg., vol. 146 (1997) pp. 91-126.

[MEN 93] Ph. Menétrey and Th. Zimmermann, Object-Oriented Non-Linear Finite Element Analysis : Application to J2 plasticity, Computers & Structures, vol. 49 n° 5 (1993) pp. 767-777.

[MIL 88] G.R Miller., A LISP-Based Object-Oriented approach to structural analysis, Engr. with Comp., vol. 4 (1988) pp. 197-203.

[MIL 91] G.R Miller., An Object-Oriented Approach to Structural Analysis and Design, Computers & Structures, vol. 40 n° 1 (1991) pp. 75-82.

[NIE 94] L.O. Nielsen, A C++ class library for FEM special purpose software, Internal report, Department of Structural Engineering, Technical University of Denmark, Serie R vol. 308 (1994).

[NOO 79] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in structural mechanics-progress and potential, Computers & Structures, vol. 10 (1979) pp. 95-118.

[NOO 81] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in nonlinear finite element analysis, Computers & Structures, vol. 13 (1981) pp. 379-403.

[NOO 90] Symbolic computations and their impact on mechanics, Winter Annual Meeting of the American Society of Mechanical Engineers, Dallas, Texas, November 25-30, 1990, ed. by A.K. Noor, I. Elishakoff and G. Hulbert, PVP ; vol. 205 (1990).

[PAS 97] M. Pastor, M. Quecedo and O.C. Zienkiewicz, A mixed displacement-pressure formulation for numerical analysis of plastic failure, Comp. & Struct., vol. 62 (1997) pp. 13-23.

[PES 88] R.L. Peskin and M.F. Russo, An Object-Oriented System Environment For Partial Differential Equation Solution, Proceedings, ASME Computations in Engineering, (1988) pp 409-415.

[POT 97a] Potapov S and Jacquart G., Un algorithme ALE de dynamique rapide basé sur une approche mixte éléments finis-volumes finis, Actes du 3ième Colloque national en calcul des structures de Giens, Hermès (1997) pp. 509-514.

[POT 97b] Potapov S, Un algorithme ALE de dynamique rapide basé sur une approche mixte Eléments finis-Volumes finis. Implémentation en langage orienté objet C++ , Phd thesis report, Ecole Centrale Paris, (1997).

[RAP 93] B. Raphael and C.S. Krishnamoorthy, Automating finite element development using Object-Oriented techniques, Engineering Computations, vol. 10 (1993) 267-278.

[RUS 96] A. Russo, Bubble stabilization of finite element methods for the linearized incompressible Navier-Stokes equations, Comput. Methods Appl. Mech. Engrg., 132 (1996) pp. 335-343.

[REH 89] 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).

[ROS 92a] J.T. Ross, J.P. Morrow, L.R. Wagner and G.F. Luger, Two paradigms for OOP models for scientific applications, Proceedings of 8th conf. held in conjunction with AEC Systems 92, Dallas, June 7-9 1992, Ed. Barry, Goodno and Wright, ASCE (1992) pp. 535-542.

[ROS 92b] J.T. Ross, L.R. Wagner and G.F. Luger, Object-oriented programming for scientific codes. I: Thoughts and concepts, Comp. Civ. Engr., vol. 6 (1992) pp. 480-496.

[ROS 92c] J.T. Ross, L.R. Wagner and G.F. Luger, Object-oriented programming for scientific codes. II: Examples in C++, Comp. Civ. Engr., vol. 6 (1992) pp. 480-496.

[SCH 83] R. Schreiber and H.B. Keller, Driven cavity flows by efficient numerical techniques, J. Computational Physics , vol. 49 (1983) 310-333.

[SCH 92] S.P. Scholz, Elements of an object-oriented FEM ++ program in C++, Comp. and Struct., 43 (1992) pp. 517-529.

[SCH 93] R. Schreiber and H.B. Keller, Driven cavity flows by efficient numerical techniques, J. Com. Phy., vol. 49 (1983) pp. 310-333.

[SHA 88] F. Shakib, Finite Element analysis of the compressible Euler and Navier-Stokes equations, Ph.D. thesis report, Stanford University, 1988.

[SIL 94] P. P. Silvester and S. V. Chamlian, Symbolic Generation of Finite Elements for Skin-Effect Integral Equations, IEEE Transactions on magnetics, vol 30 n° 5 (1994) pp. 3594-3597.

[SIM 93] J.C. Simo and F. Armero, Improved versions of assumed enhanced strain tri-linear elements for 3D finite deformation problems, Comput. Methods Appl. Mech. Engrg., 110 (1993) pp. 359-386.

[SIM 94] J.M. SIM, An object-oriented development system for finite element analysis, Phd thesis report, Arizona State University, (1994).

[SMA 93a] Smalltalk for Win32, Reference guide, Digitalk Inc. (1993).

[SMA 93b] Smalltalk for Win32, Encyclopedia of classes, Digitalk Inc. (1993).

[STO 97] M. Storti, N. Nigro and S. Indelsohn, Equal order interpolations: a unified approach to stabilize the incompressible and advective effects, Comput. Methods Appl. Mech. Engrg., 143 (1997) pp. 317-331.

[SUB 96] Subpanes/V for VisualSmalltalk, Programming Guide, ObjectShare Systems Inc., (1996).

[SUS 94] M. Sussman, P. Smereka and S. Osher, A level set approach for computing solutions to incompressible two-phase flow, J. Comp. Phy., vol. 114 (1994) 146-159.

[TEZ 92a] T.E. Tezduyar, S. Mittal, S.E. Ray and R. Shih, Incompressible flow computations with stabilized bilinear and linear equal-order-interpolation velocity-pressure elements, Comput. Methods Appl. Mech. Engrg., 95 (1992) pp. 221-242.

[TEZ 92b] T.E. Tezduyar, M. Behr and J. Liou, A new strategy for finite element computations involving moving boundaries and interfaces - The deforming-spatial-domain/space-time procedure : I. The concept and the preliminary numerical tests, Comput. Methods Appl. Mech. Engrg., 94 (1992) pp. 339-351.

[TEZ 92c] T.E. Tezduyar, M. Behr, S. Mittal and J. Liou, A new strategy for finite element computations involving moving boundaries and interfaces - The deforming-spatial-domain/space-time procedure : II. Computation of free-surface flows, two liquid flows, and flows with drifting cylinders, Comput. Methods Appl. Mech. Engrg., 94 (1992) pp. 353-371.

[TEZ 92d] T.E. Tezduyar, Stabilized finite element formulations for incompressible flow computations, Advances in applied Mechanics, vol. 28 (1992) pp. 1-44.

[TEZ 97] T.E. Tezduyar, S. Aliabadi and M. Behr, Enhanced-discretization interface-capturing technique, AHPCRC-University of Minnesota, Preprint 97-019, (1997).

[TRU 97] A. Truty and Th. Zimmermann, A robust formulation for FE-analysis of elasto-plastic media, Numerical methods in Geomechanics, NUMOG VI, Pietruszczak & Pande (eds) Balkena (1997) pp. 381-386.

[TWO 93] W.W. Tworzydlo and J.T. Oden, Towards an automated environment in computational mechanics, Comput. Methods Appl. Mech. Engrg., vol. 104 (1993) pp. 87-143.

[VER 88] P. Verpaux, T. Charras and A. Millard, CASTEM 2000: une approche moderne du calcul de structure, Ed. Fouet, Ladevèze and Ohayon, Pluralis, vol. 2 (1988) pp. 261-271.

[VIS 95a] VisualSmalltalk Enterprise - 32 Bit Pure Object-Oriented Programming System, User's guide, ParkPlace Digitalk, (1995).

[VIS 95b] VisualSmalltalk Enterprise - 32 Bit Pure Object-Oriented Programming System, Language Reference, ParkPlace Digitalk, (1995).

[VIS 95c] VisualSmalltalk Enterprise - 32 Bit Pure Object-Oriented Programming System, Encyclopedia of classes for Win32, ParkPlace Digitalk, (1995).

[WAL 96] L. Walterthum, Programmation orientée objet et calcul par éléments finis. Application à la conception d'un logiciel de simulation en mise en forme des matériaux, Ph.D. thesis report, Université de Franche-Comté, (1996).

[WAN 86] P.S. Wang, FINGER : A symbolic System For Automatic Generation of Numerical Programs in Finite Element Analysis, J. Symbolic Computation, vol. 2 (1986) pp 305-316.

[WIN 96] WindowBuilder Pro/V 3.1, Tutorial and Reference Guide, ObjectShare Systems Inc., (1996).

[YAG 90] 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.

[YAN 94] C.Y. Yang, An algebraic-expressed finite element model for symbolic computation Computers & Structures, vol. 52 n° 5 (1994) pp. 1069-1077.

[YU 94] G.G. Yu, Object-oriented models for numerical and finite element analysis, PhD thesis report, The Ohio State University, (1994).

[ZEG 94] G.W. Zeglinski and R.P.S. Han, Object oriented matrix classes for use in a finite element code using C++, , Int. J. Num. Meth. Engr. , vol. 37 (1994) 3921-3937.

[ZIE 91] O. C. Zienkiewicz and R. L. Taylor, The finite element method, 4th ed. Vol. 2, Solid and fluid mechanics, Dynamics and Non-Linearity, McGraw-Hill (1991).

[ZIM 92a] 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.

[ZIM 92b] Th. Zimmermann, Y. Dubois-Pèlerin and P. Bomme, Object-oriented finite element programming : Theory and Smalltalk V Implementation for FEM_Object$_{pc}$^{TM}001, Elmepress international (1992).

[ZIM 92c] Th. Zimmermann and Y.D. Dubois-Pélerin, The object-oriented approach to finite elements: Concepts and implementations, Proc. of 1st European Conf. on Num. Meth. in Engr., Brussels, Ed. C. Hirsch, 7-11 September 1992.

[ZIM 94] Th. Zimmermann and D. Eyheramendy, Object-oriented finite elements programming : Automatic programming, Proceedings of the Third Congress on Computational Mechanics (IACM), Chiba Japan, vol. II (1994) pp 1527-1528.

[ZIM 95 a] Th. Zimmermann and D. Eyheramendy, Symbolic object-oriented Finite Element Programming - Application to incompressible viscous flow, Proceedings of IECS 95, S.N. Atluri, G. Yagawa, T.A. Cruse (Eds.), vol. 1 (1995) pp. 21-26.

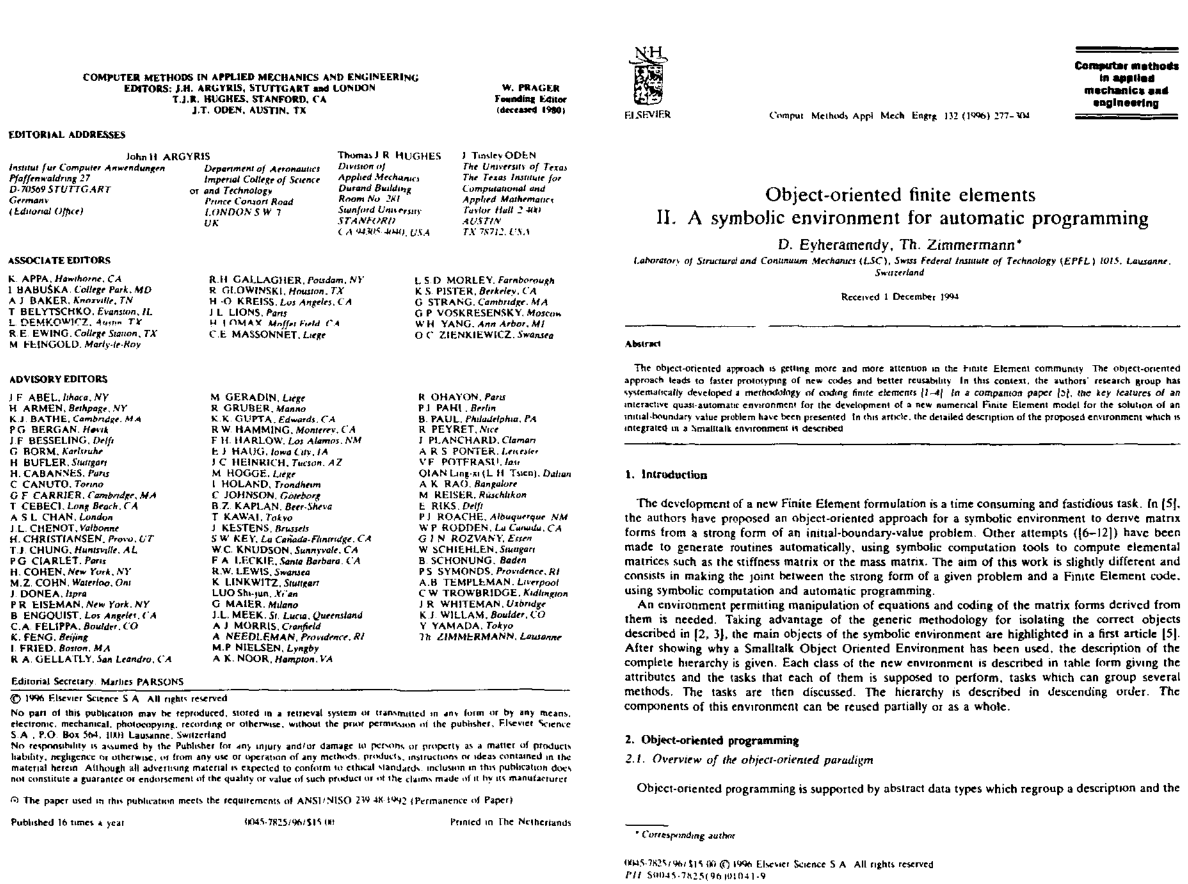
[ZIM 95 b] Th. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier and S. Commend, Object-Oriented Finite Element Techniques : Towards a general purpose environment, Proceedings of the 4th Int. Conf. on the application of AI in Civil and Structural Eng., Cambridge, (1995).

[ZIM 96] Th. Zimmermann and D. Eyheramendy, Object-oriented finite elements : I. Principles of symbolic derivations and automatic programming, Comput. Methods Appl. Mech. Engrg., 132 (1996) pp. 277-304.

[ZIM 97] Th. Zimmermann, D. Eyheramendy, P. Bomme, S. Commend and R.S. Arruda, Object-oriented finite element programming : Languages, Symbolic derivations, Reasoning capabilities, proceedings of NAFEM 97-Stuttgart, vol. 1 (1997) pp. 652-663.

Appendix A

This article was published in
Computer Methods in Applied Mechanics and Engineering
vol. 132 (1996) pp. 277-304.



Published 16 times a year 0045-7825/96/$15 00 Printed in The Netherlands

ELSEVIER

Compur Methods Appl Mech Engrg 132 (1996) 277-304

Computer methods
in applied
mechanics and
engineering

Object-oriented finite elements
II. A symbolic environment for automatic programming

D. Eyheramendy, Th. Zimmermann*

Laboratory of Structural and Continuum Mechanics (LSC), Swiss Federal Institute of Technology (EPFL) 1015, Lausanne, Switzerland

Received 1 December 1994

Abstract

The object-oriented approach is getting more and more attention in the Finite Element community The object-oriented approach leads to faster prototyping of new codes and better reusability In this context, the authors' research group has systematically developed a methodology *of coding finite elements [1-4] In a companion paper [5], the key features of an* interactive quasi-automatic environment for the development of a new numerical Finite Element model for the solution of an initial-boundary value problem have been presented In this article, the detailed description of the proposed environment which is integrated in a Smalltalk environment is described

1. Introduction

The development of a new Finite Element formulation is a time consuming and fastidious task. In [5], the authors have proposed an object-oriented approach for a symbolic environment to derive matrix forms from a strong form of an initial-boundary-value problem. Other attempts ([6-12]) have been made to generate routines automatically, using symbolic computation tools to compute elemental matrices such as the stiffness matrix or the mass matrix. The aim of this work is slightly different and consists in making the joint between the strong form of a given problem and a Finite Element code, using symbolic computation and automatic programming.

An environment permitting manipulation of equations and coding of the matrix forms derived from them is needed. Taking advantage of the generic methodology for isolating the correct objects described in [2, 3], the main objects of the symbolic environment are highlighted in a first article [5]. After showing why a Smalltalk Object Oriented Environment has been used, the description of the complete hierarchy is given. Each class of the new environment is described in table form giving the attributes and the tasks that each of them is supposed to perform, tasks which can group several methods. The tasks are then discussed. The hierarchy is described in descending order. The components of this environment can be reused partially or as a whole.

2. Object-oriented programming

2.1. Overview of the object-oriented paradigm

Object-oriented programming is supported by abstract data types which regroup a description and the

* *Corresponding author*

0045-7825/96/$15 00 © 1996 Elsevier Science S A All rights reserved
PII S0045-7825(96)01041-9

tasks it is able to perform. The basic entity is the object. Its conceptualization is described in the class. Each class possesses its own data structure, the attributes, and the behavior associated to them, the methods. Encapsulation ensures that the object is the only one allowed to access its own data and its own tasks. Classes are organised as a hierarchy, which allows simple or multiple inheritance of data and tasks. Objects communicate through messages. The fact that different objects can answer to the same message and then have a proper behavior results from polymorphism. A complete definition of the paradigm can be found in [13].

The object-oriented paradigm is enhanced when the non-anticipation principle (state encapsulation) is respected (see [2, 3]). This simply means that while programming, one does not rely on any prescribed state of the system: requests to an object do not assume that any specific task has already been executed or that data already exist. This is illustrated by the following methods:

```
giveSpaceDimension
   spaceDimension exists
      ifFalse:[self getSpaceDimension].
   ^spaceDimension
```

```
giveJacobianMatrix
   jacobianMatrix exists
      ifFalse:[self computeJacobianMatrix].
   ^jacobianMatrix
```

This way of programming gets rid of all forms of sequentiality. Tasks are performed just when necessary.

2.2. *An object-oriented programming language: Smalltalk*

The object-oriented language chosen for this application is Smalltalk. A description of the language can be found in [14]. The version used here is Smalltalk for Win32, a PC version of the language; it is completed by a fast development toolkit for the creation of windows to provide the symbolic environment with a userfriendly graphical interface.

Smalltalk is a simple-inheritance object-oriented language. The encapsulation of data and tasks is complete. Data and tasks are private; only the list of methods is public (selector of methods).

The management of the memory is automatic. Allocation and deallocation of the memory is made using a mechanism called garbage collector (see [14]).

But the real power of this language for the programmer consists in the fact that more than a language, Smalltalk is a complete environment. It is composed of a number of windows corresponding to tools. Debugging then becomes an easy task. The environment is described in [14]. Moreover, Smalltalk has a lot of predefined classes described in [15]; this is also of considerable help for the programmer.

Another reason why Smalltalk is a good prototyping tool is that it is an interpreted language. Each method can be tested as soon as it has been written. This makes Smalltalk one of the most powerful tools for testing new ideas. Furthermore, the lack of computational efficiency is not an obstacle for the environment described hereafter, which involves symbolic manipulations rather than large computations.

3. Description of the hierarchy of FEM_Theory

As usual, in an object-oriented environment, the implementation implies three types of operations:
- the first one is adding new classes. It represents the main way of implementing the native hierarchy
- the second one is adding subclasses to existing classes. It is called generalization and specialization. It is an elegant way to reuse already programmed classes. So the new class inherits attributes and methods from its super classes and is given new attributes and implements new methods
- finally, new methods may be added to a class.

The native hierarchy is modified as shown in Fig. 1.

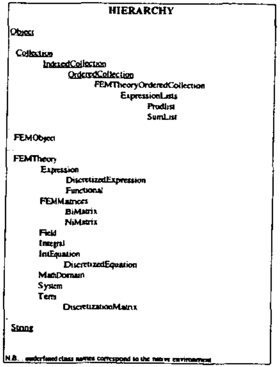


Fig. 1. Hierarchy of FEM_Theory inserted in the one of Smalltalk

The main new classes are grouped under class **FEMTheory**. As discussed in [5], the classes are directly inspired from the mathematical and mechanical entities, classes **Expression** and **IntEquation** and their specialization which are, respectively, **DiscretizedExpression** and **DiscretizedEquation**, class **System** which is a set of equations, class **Term** and its specialization **DiscretizationMatrix** which conceptualizes the different components of an expression. Class **FEMMatrices** is implemented by subclasses **BiMatrix** and **NiMatrix**. Its role is to instanciate matrices from particular structures such as matrices B or N (see [16]). Class **Functional** and **Integral** make it possible to build the weak form.

In the class OrderedCollection we find the subclass **FEMTheoryOrderedCollection** which groups the specialized ordered collections, the sums and products, class **SumList** and class **ProdList**.

The main methods added to existing classes are found in class **String**. These methods make it possible to get instances of classes **Term** and **Expression** from a single string (expression given by the user on the keyboard). The different classes may now be described in detail.

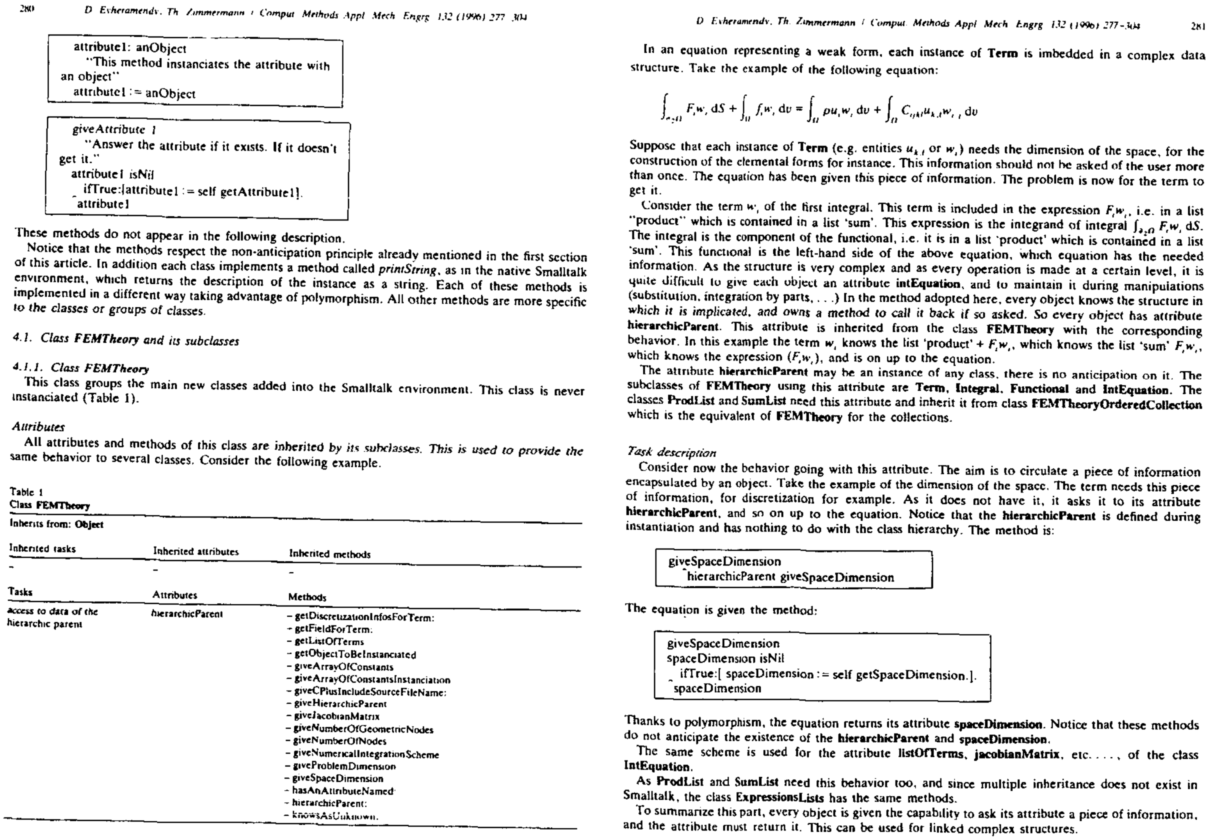
4. Description of the classes of FEM_Theory

Generally speaking, each class implements three methods to handle its own attributes: *attribute:*, *giveAttribute* and *getAttribute*. Consider a class with an attribute **attribute1**.

The following three methods are provided:
- *attribute1:* which instanciates **attribute1**
- *giveAttribute1* which returns **attribute1** if it already exists
- *getAttribute1* which gets it from another source.

This last method allows the class to ask one of its attributes or to ask the user the needed information.

These methods are:



```
attribute1: anObject
   "This method instanciates the attribute with
an object"
   attribute1 := anObject
```

```
giveAttribute 1
   "Answer the attribute if it exists. If it doesn't
get it."
   attribute1 isNil
   _ ifTrue:[attribute1 := self getAttribute1].
   ^attribute1
```

These methods do not appear in the following description.

Notice that the methods respect the non-anticipation principle already mentioned in the first section of this article. In addition each class implements a method called *printString*, as in the native Smalltalk environment, which returns the description of the instance as a string. Each of these methods is implemented in a different way taking advantage of polymorphism. All other methods are more specific *to the classes or groups of classes.*

4.1. *Class FEMTheory and its subclasses*

4.1.1. *Class FEMTheory*

This class groups the main new classes added into the Smalltalk environment. This class is never instanciated (Table 1).

Attributes

All attributes and methods of this class are inherited *by its subclasses. This is used to provide the* same behavior to several classes. Consider the following example.

Table 1
Class **FEMTheory**

Inherits from: **Object**

Inherited tasks	Inherited attributes	Inherited methods
-	-	-
Tasks	**Attributes**	**Methods**
access to data of the hierarchic parent	hierarchicParent	- getDiscretizationInfosForTerm: - getFieldForTerm: - getListOfTerms - getObjectToBeInstanciated - giveArrayOfConstants - giveArrayOfConstantsInstanciation - giveCPlusIncludeSourceFileName: - giveHierarchicParent - giveJacobianMatrix - giveNumberOfGeometricNodes - giveNumberOfNodes - giveNumericalIntegrationScheme - giveProblemDimension - giveSpaceDimension - hasAnAttributeNamed: - hierarchicParent: - knowsAsUnknown.

In an equation representing a weak form, each instance of **Term** is imbedded in a complex data structure. Take the example of the following equation:

$$\int_{\partial_2 \Omega} F_i w_i \, dS + \int_\Omega f_i w_i \, dv = \int_\Omega \rho u_i w_i \, dv + \int_\Omega C_{ijkl} u_{k,l} w_{i,j} \, dv$$

Suppose that each instance of **Term** (e.g. entities $u_{k,l}$ or w_i) needs the dimension of the space, for the construction of the elemental forms for instance. This information should not be asked of the user more than once. The equation has been given this piece of information. The problem is now for the term to get it.

Consider the term w_i of the first integral. This term is included in the expression $F_i w_i$, i.e. in a list "product" which is contained in a list 'sum'. This expression is the integrand of integral $\int_{\partial_2 \Omega} F_i w_i \, dS$. The integral is the component of the functional, i.e. it is in a list 'product' which is contained in a list 'sum'. This functional is the left-hand side of the above equation, which equation has the needed information. As the structure is very complex and as every operation is made at a certain level, it is quite difficult to give each object an attribute **intEquation**, and to maintain it during manipulations (substitution, integration by parts, . . .) In the method adopted here, every object knows the structure in *which it is implicated, and owns a method to call it back if so asked.* So every object has attribute **hierarchicParent**. This attribute is inherited from the class **FEMTheory** with the corresponding behavior. In this example the term w_i knows the list 'product' + $F_i w_i$, which knows the list 'sum' $F_i w_i$, which knows the expression $(F_i w_i)$, and is on up to the equation.

The attribute **hierarchicParent** may be an instance of any class, there is no anticipation on it. The subclasses of **FEMTheory** using this attribute are **Term**, **Integral**, **Functional** and **IntEquation**. The classes **ProdList** and **SumList** need this attribute and inherit it from class **FEMTheoryOrderedCollection** which is the equivalent of **FEMTheory** for the collections.

Task description

Consider now the behavior going with this attribute. The aim is to circulate a piece of information encapsulated by an object. Take the example of the dimension of the space. The term needs this piece of information, for discretization for example. As it does not have it, it asks it to its attribute **hierarchicParent**, and so on up to the equation. Notice that the **hierarchicParent** is defined during instantiation and has nothing to do with the class hierarchy. The method is:

```
giveSpaceDimension
   ^hierarchicParent giveSpaceDimension
```

The equation is given the method:

```
giveSpaceDimension
spaceDimension isNil
   ifTrue:[ spaceDimension := self getSpaceDimension.].
^spaceDimension
```

Thanks to polymorphism, the equation returns its attribute **spaceDimension**. Notice that these methods do not anticipate the existence of the **hierarchicParent** and **spaceDimension**.

The same scheme is used for the attribute **listOfTerms**, **jacobianMatrix**, etc. . . . , of the class **IntEquation**.

As **ProdList** and **SumList** need this behavior too, and since multiple inheritance does not exist in Smalltalk, the class **ExpressionsLists** has the same methods.

To summarize this part, every object is given the capability to ask its attribute a piece of information, and the attribute must return it. This can be used for linked complex structures.

4.1.2. Class **Expression** and its subclasses

4.1.2.1. Class ***Expression***. This class is the representation of all the mathematical expressions manipulated for derivation (Table 2).

Table 2
Class **Expression**

Inherits from: **FEMTheory**, **Object**

Inherited tasks	Inherited Attributes	Inherited methods
(1) creation		
(2) access to data of the hierarchic parent	hierarchicParent	**{all the methods of FEMTheory}**
Tasks	**Attributes**	**Methods**
(1) manipulation	sumList inverseIndex	+, addDist:, −, *, multDist:,/, negated, inverse allExpand addYourselfToSumList expand getDerivedExpression getIntegratedExpression getInstanciationName getIntegratedExpression giveDivTerm giveDivTermMultiplier giveListOfUnknowns isElementaryExpression mayBeIntegratedByParts printString printStringForEvaluation printStringForEvaluationCPlus remove: anObject replace: exp 1 by: exp2 simplify substituteTerm: term1 with: term2 squared
	listOfVariables	deriveWithRespectToVariable: i deriveInGlobalCoordinateWithRespectToVariable: i
(2) discretization		getDiscretizedForm
(3) creation of methods	instanciationName	addToCollection:expressionName addToFile:expressionName createCPlusMethod: met inClass: class createCPlusMethodWithArgument: met inClass:class createMethod: met inClass: class createMethodWithArgument:met inClass:class giveArrayoOfConstantsName giveConstantsComputationStringForElement giveConstantsInstanciationStringForElement giveConstantsStringForElement giveCPlusConstantsInstanciationString giveCPlusConstantsString giveCPlusCoordinatesString giveCPlusCurrentPointAndLoadInstanciationString giveCPlusNodesCoordinates giveCPlusNodesCoordinatesInstanciationString giveCurrentPointAndLoadInstanciationStringForElement

Attributes

As described above, the expression may be seen as a sum of entities. That is why the first attribute is **sumList**, instance of class **SumList**. The second attribute, which generalizes the type of possible expressions, is **inverseIndex**. It is 1 or −1 and represents the exponent of an expression, giving its inverse. For example, for $1/(4X + Y)$, **sumList** is $4X + Y$ and **inverseIndex** is −1.

Finally, the expression knows its variables, listed in attribute **listOfVariables** instance of **FEMTheoryOrderedCollection**. For expression $4X + Y$ the collection contains term X and term Y.

Notice that this class can represent expressions like $(\frac{1}{4}(X + 1)(Y - 1))$ which can be an example of shape function, as well as $(\sigma_{ij,j} w_i)$ which can be member of a variational principle.

Task description

(1) The first behavior given to this class corresponds to the basic mathematical operations +, −, *, /. The basic operations are completed with methods *addDist:*, *multDist:*, *negated* and *inverse*. The difference between + and *addDist:* is best shown through an example: $(ax + b) + (y + c)$ gives a new instance of expression $((ax + b) + (y + c))$, whereas $(ax + b)$ *addDist:* $(y + c)$ gives $(ax + b + y + c)$. Similarly, *multDist:* effectuates a partial distribution, whereas * returns a new expression built from the receiver and the argument. The operation *negated* which takes the opposite of the receiver is decentralized to the **sumList**.

In the first article [5], some basic operations have been introduced like substitution, expansion, factorization and simplification. The basic set is now completed.

The method *allExpand* does a complete expansion of the receiver. The principle of this operation is to check if the product of expressions contains only terms, and if it does not, to complete the distribution in the product by sending the message *self expand*. The behavior is then decentralized to **sumList**. Also, two methods of substitution exist. The first one *replace: aProduct by: anExp* replaces a simple product (corresponding to the first argument) by an expression (the second argument). The second method *substitute: aTerm with: anExpression* only replaces a single term by an expression. The message is sent to the expression but the manipulation operations are done by **sumList**. This explains the simplicity and the clarity of these methods.

Other tasks may be decomposed in parts. The first one concerns the expressions using the indicial notation, such as $(\sigma_{ij,j} + f_i)$. The specific behavior has to do with the integration by parts, for which most operations are decentralized to the integrand which is an instance of class **Expression**. Five methods are necessary here. The first one *mayBeIntegratedByParts* checks if the receiver may be integrated by parts or not. This method tests if the expression belongs to one of the three following types: $A_{ij,j}U_i$ or $A_{i,i}U$ or $A_{,i}U$. Note that these types are sufficient to deal with elasticity, heat conduction and beam problems. The methods *giveDivTerm* and *giveDivTermMultiplier* return, respectively, the term which is derived and the one which is not, i.e. $A_{ij,j}$ and U_i of expression $A_{ij,j}U_i$.

The method *getDerivatedExpression* and *getIntegratedExpression* builds the expressions to get the integral integrated by parts (see Fig. 2).

The second one concerns functions such as $\frac{1}{4}(x + 1)(y - 1)$. The behavior attached to this type of expression is the derivation and the creation of methods to compute the expression numerically. The derivation methods *deriveWithRespectTo: i* and *deriveInGlobalAxesWith: jacobianMatrix withRespectTo: i*, respectively, represent the derivation with respect to local and global axes. These methods depend on the nature of the argument **i**. If **i** is a term, the receiver is

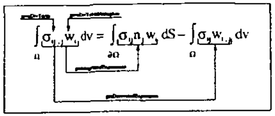

$\int_{\Omega} \sigma_{ij,j} w_i \, dv = \int_{\partial\Omega} \sigma_{ij} n_j w_i \, dS - \int_{\Omega} \sigma_{ij} w_{i,j} \, dv$

Fig. 2. Description of integration by parts

derived with respect to it, if it is a number (1, 2, . . .) the receiver is derived with respect to the *i*th variable of the argument **listOfVariables**. In fact in these methods the derivation scheme is performed by **sumList** (the basic and quite natural idea is to derive sums and products) with the correct variable of derivation. The method *deriveInGlobalAxesWith: jacobianMatrix withRespectTo: i* builds an array containing the partial differentials with respect to local axes, and returns the *i*th component of the product between this array and the jacobian matrix.

(2) In the method *getDiscretizedForm* an instance of **DiscretizedExpression** is created.

(3) Finally, the expression is able to create code in an existing Finite Element code. The method for this is *createMethodWithArgument: aSelector inElement: aClassElement*. This method is specific to the finite element code into which the implementation is done. This is discussed in a followup article. The principle of this method is to build a string, corresponding to source code and to compile it in a given class. The method *printStringForEvaluation* returns the string representing the expression with the mathematical operators. So it is just necessary, in the created method, to instanciate the different variables used in the expression.

This scheme exists at present for two differents codes: FEM_Object$_{pc}$™ (Smalltalk version, see [17]) and *FEM_Object$_{C++}$™ (C++ version, see [18])*.

*4.1.2.2. Subclasses of class **Expression**.*

(a) *Class **Functional***. The class **Functional** (Table 3) represents the functional needed to represent the problem: *it is an expression containing integrals*. E.g. functional $\int_\Omega C_{ijkl}u_{k,l}w_{i,j}\,dv + \int_\Omega f_i w_i\,dv$ has two integrals $\int_\Omega C_{ijkl}u_{k,l}w_{i,j}\,dv$ and $\int_\Omega f_i w_i\,dv$.

Attributes

All attributes are inherited from its superclass.

Task description

A part of its behavior is inherited from its superclasses, i.e. the accessibility to the data of the hierarchic parent from class **FEMTheory** and some manipulation tasks from class **Expression**.

Table 3
Class **Functional**

Inherits from: **Expression**, **FEMTheory**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm: term getListOfTerms giveHierarchicParent *giveSpaceDimension* knowAsUnknow
(3) manipulation	sumList	allExpand expand printString replace: exp1 by: exp2 substituteTerm: term1 with: term2
Tasks	**Attributes**	**Methods**
(1) manipulation		+,− getDiscretizedForm integrateByPartsSelection: selectedIntegral minimizeFunctional printString substituteSelection: selectedIntegral
(2) discretization		getDiscretizedForm

(1) The added manipulation method is *integrateByPartsSelection: anIntegral*. In this method, the attribute **sumList** is asked to ask the argument to integrate by parts. The behavior is similar for the method *substitute Selection: aSelectedIntegral*.

(2) The main behavior is the discretisation. The method *getDiscretizedForm* instanciates a **DiscretizedExpression** for which the attribute **sumList** is put in the result of the **sumList** discretization

Notice that all methods are delegated to the attribute **sumList**.

(b) *Class **DiscretizedExpression***. This class represents expressions needed for modelizing discrete problems (Table 4).

Attributes

As for the class **Functional**, this class inherits its attributes from class **Expression**.

Task description

(1) The discretized expression needs to transpose itself:

$$d^tK + f^t \xrightarrow{\text{transpose}} K^td + f$$

The manipulation is done by the attribute **sumList**. The result of the discretisation of the weak form is an expression.

The task which consists in invoking linear independence of each component of a sum is *invokeLinearIndependence*. This operation is illustrated by the example coming next.

Table 4
Class **DiscretizedExpression**

Inherits from: **FEMTheory**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) access to data of the hierarchic parent	hierarchicParent	giveHierarchicParent giveSpaceDimension
(3) manipulation	sumList	+, addDist: , −, *, multiDist , /, negated, inverse allExpand expand printString replace: exp1 by: exp2 substituteTerm: term1 with: term2
Tasks	**Attributes**	**Methods**
(1) manipulation		integrateYourComponentsOnDomain: string invokeLinearIndependence transpose
(2) analysis		findAllUnknowns findMatrixCorrespondingToBodyLoads *findMatrixCorrespondingToSurfaceLoads* gfindMatrixCorrespondingToUnknown get0KAFUnknownMatrix getKnownMatrix
(3) creation of methods		createMethodsInElement class createCPlusMethodsInElement class

Consider the following equation: $d'Kd^* + f'd^* + d'Gp^* + p'G'd^* = 0$ must be true for every d^* and p^*. So each coefficient of d^* and p^* must be equal to zero. It yields the system of equations:

$$\begin{cases} d'K + f' + p'G' = 0 \\ d'G = 0 \end{cases} .$$

The result is a set of equations, an instance of **System**. In the example treated above, the discretized expression $d'Kd^* + f'd^* + d'Gp^* + p'G'd^* = 0$ builds **listOfVariables** $\{d^*, p^*\}$ and derives the expression with respect to each of its variables, i.e. d^* and p^*. The result is here a system of two equations. The number of equations depends on the number of virtual fields of the formulation

(2) The expression can find the matrices corresponding to the zero admissible field and add them to the attribute **listOfVariables** thanks to the method *get0KAFUnknownMatrix*. The rest of the behavior is decentralized to **sumList**.

(3) At last, the **DiscretizedExpression** is able to create a code; in the method *createMethodsInElement: anElementClass* for Smalltalk and in the method *createCPlusMethodsInElement: anElementName* for the C++ version, the instance decentralizes this behavior to its attribute **sumList**.

*4.1.3. Classes **FEMMatrices**, **BiMatrix** and **NiMatrix*** (Tables 5 – 7)

The classes **FEMMatrices**, **BiMatrix** and **NiMatrix** have no attributes. Their only behavior is to instanciate matrices; that means that instances of these classes never appear. Their role is to build the correct structures for N_i and B_i with a given shape function [16]. It can be considered as the implementation of the two arrays defining the shape functions N and global shape function derivative B matrices.

The methods associated with classes **BiMatrix** and **NiMatrix** are able to build the elementary structures. For example, the class method of **BiMatrix**, *new21WithExpression: anExpression* returns an instance of **Matrix** representing the structure for the entities: dimension of space $n_{sd} = 2$ and number of degrees of freedom per node for the given field $n_{dof} = 1$. The answer is:

$$B_i = \begin{bmatrix} N_{i,1} \\ N_{i,2} \end{bmatrix}$$

and the corresponding method:

```
new21WithExpression: anExpression

 "Answer the B matrix expressed in the local coordinates
axes"

 |n11 n21 biMatrix|
 n11 := anExpression deriveWithRespectToVariable:1.
 n21 := anExpression deriveWithRespectToVariable:2.

 biMatrix := Matrix new: 2@1.
 biMatrix at: 1@1 put: n11.
 biMatrix at: 2@1 put: n21.

 ^biMatrix
```

The problem now is to chose the correct message to instanciate it. This is done by the method *newWith: aPoint withExpression: anExpression* is inherited from **FEMMatrices**. In this method, the class builds the string (it is the selector of method) 'new21WithExpression:' from the argument **aPoint** (2@1) and sends it to itself with the message: *self perform: new21WithExpression: with: anArray*. The message is built dynamically during the execution of the code. Nowhere in the code, does the message *BiMatrix new21WithExpression: anExp* exist. This faculty to construct the code and to execute it next dynamically is proper to Smalltalk, which is an interpreted language. This is the way chosen to program arrays with multiple entries All types of matrix structures are to be implemented as subclasses of **FEMMatrices**.

Table 5
Class **FEMMatrices**

Inherits from **FEMTheory, Object**

Inherited tasks	Inherited attributes	Inherited methods
Tasks	Attributes	Methods
creation		new: dimension withExpression: expression

Table 6
Class **BiMatrix**

Inherits from **FEMMatrices, FEMTheory, Object**

Inherited tasks	Inherited attributes	Inherited methods
creation		new: dimension withExpression: expression
Tasks	Attributes	Methods
creation		new11WithExpression new21WithExpression new22WithExpression new33WithExpression new31WithExpression

Table 7
Class **NiMatrix**

Inherits from: **FEMMatrices, FEMTheory, Object**

Inherited tasks	Inherited attributes	Inherited methods
creation		new: dimension withExpression: expression
Tasks	Attributes	Methods
creation		new11WithExpression new21WithExpression new22WithExpression new33WithExpression

*4.1.4. Class **Integral***

This class is the representation of the mathematical integral (Table 8)

Attributes

As the term and the discretisation matrix are respectively the basic entities of the expressions and the

Table 8
Class **Integral**

Inherits from **FEMTheory**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) access to data of the *hierarchic parent*	hierarchicParent	getDiscretizationInfosForTerm term *getListOfTerms* giveHierarchicParent giveSpaceDimension knowsAsUnknow: term
Tasks	**Attributes**	**Methods**
(1) manipulation	integrand	+, negated distributeIntegralTo: discretExp expand integrateByParts *printString* replace: exp1 by: exp2 substituteTerm: term1 with: term2
	domain	giveDomainBoundary may BeAddedTo: anInt addYourselfToSumList: aSum getArrayOfFunctions getCoordinatesFunctions getJacobianMatrix giveGaussPointArray giveGaussWeightArray
(2) discretization		getDiscretizedForm
(3) creation of methods		createCPlusGaussPointInstanciationMethodsIn: path forElement. elementName createCPlusGaussPointsMethodsIn. pathName forElement: elementName createCPlusMethd. symbol in pathName forElement elementName createCPlusMethodGaussianIntegration: symbol in: pathName forElement: *elementName* createGaussPointsAttributesMethodsIn: element *createGaussPointsInstanciationMethodsIn: element* createGaussPointsMethodsIn: element createMethod: symbol inElement: element create MethodGaussianIntegration: symbol inElement: element createMethodWithArgument: symbol inElement: element createMethodWithArgumentGaussianIntegration: symbol inElement: element

discretized expression, the integral is the basic one of the functional. Moreover the symbolic forms of elementary matrices use integrals.

The only attributes of this class are **integrand** which is an instance of class **Expression** and **domain** which is a string name of the domain, for example 'D' or 'dD' representing the domain D (an upper case) and its boundary (character d preceding an upper case). This last notion is important for finding the integrals upon a boundary domain.

Task description

(1) The class **Integral** implements integrals in the mathematical sense.

First, two instances can be added or subtracted, methods + and −. The method *mayBeAddedTo: anIntegral* verifies if the receiver and the argument have the same domain of integration and according to the answer returns a sum of integrals or an integral with the integrands added. For example,

$$\left(\int_{D_1} f\,\mathrm{d}v\right)+\left(\int_{D_2} g\,\mathrm{d}v\right) \quad \text{gives} \quad \left(\int_{D_1} f\,\mathrm{d}v+\int_{D_2} g\,\mathrm{d}v\right) \quad \text{whereas}$$

$$\left(\int_{D_1} f\,\mathrm{d}v\right)+\left(\int_{D_1} g\,\mathrm{d}v\right) \quad \text{gives} \quad \left(\int_{D_1} f+g\,\mathrm{d}v\right).$$

Both methods of substitution *replace: expression1 by: expression2* and *substituteTerm: term1 for: anExpression* only send the same message to the attribute **integrand**.

The main manipulation method is the integration by parts. The result is a sum of integrals. In *this method, two integrals are instanciated and the integrand is asked its integrated and its* derived forms. Then, a new instance of **SumList** is created and is added two instances of **ProdList** containing the integrals.

The integral is able to build its boundary domain with the method *giveBoundaryDomain*, the integral concatenates the string 'd' with the string **domain**. The notion of boundary is needed for the derivation.

The method *expand* only expands the integrand of the integral.

(2) In the discretized method *getDiscretizedForm*, the integral asks its integrand to get its discretized form and then integrates the components of the result, an instance of **DiscretizedExpression**, to integrate its components (method *distributeYourselfTo: discretExpression*).

(3) The last part of the behavior is the creation of methods to compute numerically a given symbolic integral; this creation is done for a given FEM code, as explained before At this state of the development only gaussian quadrature exists for both codes, FEM_Object Smalltalk and C++, but any new kind of integration can be added rather easily.

*4.1.5. Class **IntEquation** and its subclasses*

These classes are very important because they are part of the objects which are manipulated directly by the user. To begin with, the user builds an instance of equation and then manipulates it.

*4.1.5.1. Class **IntEquation**.* This class represents equations containing integrals (Table 9). It represents variational forms (and weak forms).

Attributes

The main attributes of this class are **lhs** and **rhs**. They represent the left-hand and the right hand-sides of *the equation. They may be instances of class* **Functional**. The behavior of the equation is particularized by one of the instances of **lhs** and **rhs**.

It has been shown in the above section that several pieces of information were useful for the discretization operation. As the information must be stored only once and can be reached by every object and due to the complex data structure, the equation stores them. As every object knows to which structure it belongs (through attribute **hierarchicParent**), it can expect to get the information from it. At the root of this tree is the equation.

The **IntEquation** has two attributes, instance lists, one containing the list of terms, **listOfTerms**, which has even been discretized, and one containing the unknowns of the formulation, **listOfUnknowns**. Most of the methods are here to manage these lists. Both are instances of class **OrderedCollection**. The first one contains instances of **Term**, the second one instances of **Array**. This instance of **Array** contains instances of **Term**. The first term corresponds to the trial solution, the second one corresponds to the associated weighting function.

Moreover, the equation knows the dimension of the space, attribute **spaceDimension**, and the dimension of the problem, attribute **problemDimension**.

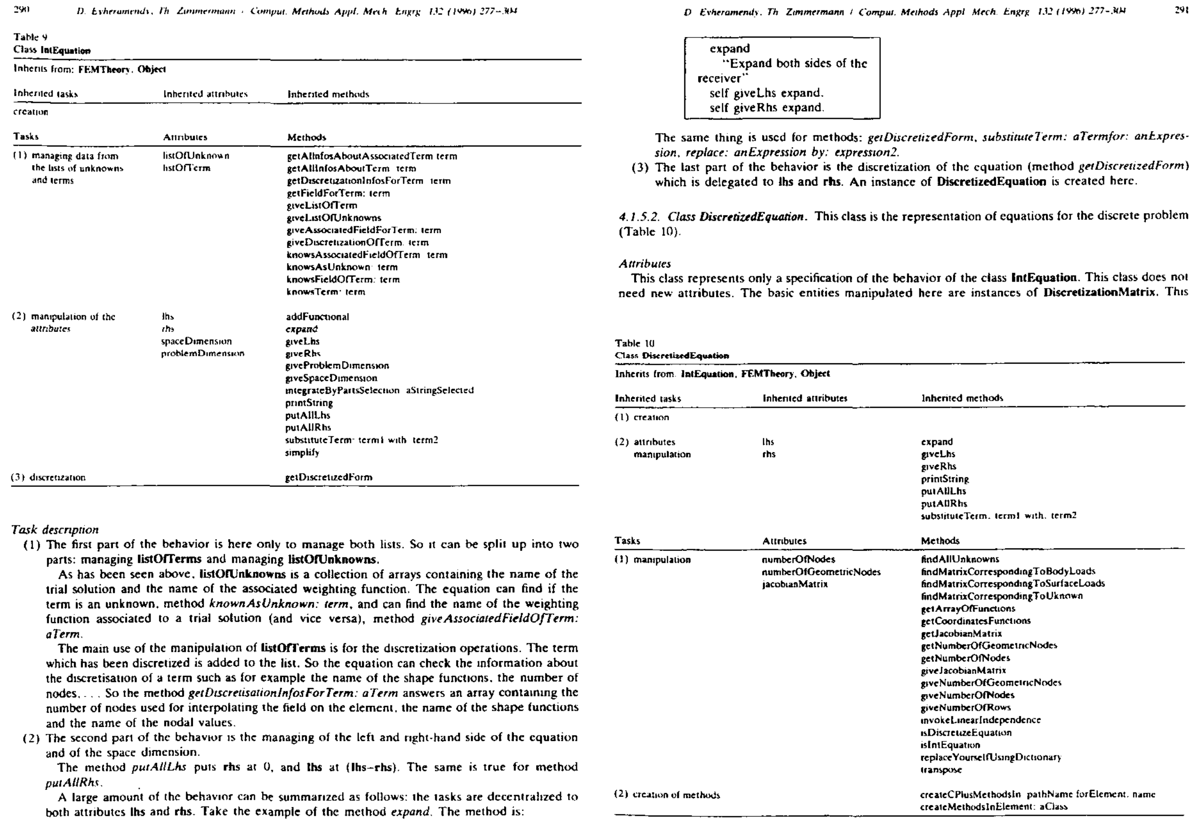


Table 9
Class **IntEquation**

Inherits from: **FEMTheory**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
creation		

Tasks	Attributes	Methods
(1) managing data from the lists of unknowns and terms	listOfUnknown listOfTerm	getAllnfosAboutAssociatedTerm term getAllInfosAboutTerm term getDiscretizationInfosForTerm term getFieldForTerm: term giveListOfTerm giveListOfUnknowns giveAssociatedFieldForTerm: term giveDiscretizationOfTerm. term knowsAssociatedFieldOfTerm term knowsAsUnknown: term knowsFieldOfTerm: term knowsTerm: term
(2) manipulation of the *attributes*	lhs *rhs* spaceDimension problemDimension	addFunctional *expand* giveLhs giveRhs giveProblemDimension giveSpaceDimension integrateByPartsSelection aStringSelected printString putAllLhs putAllRhs substituteTerm: term1 with term2 simplify
(3) discretization		getDiscretizedForm

Task description

(1) The first part of the behavior is here only to manage both lists. So it can be split up into two parts: managing **listOfTerms** and managing **listOfUnknowns**.

As has been seen above, **listOfUnknowns** is a collection of arrays containing the name of the trial solution and the name of the associated weighting function. The equation can find if the term is an unknown, method *knownAsUnknown: term*, and can find the name of the weighting function associated to a trial solution (and vice versa), method *giveAssociatedFieldOfTerm: aTerm*.

The main use of the manipulation of **listOfTerms** is for the discretization operations. The term which has been discretized is added to the list. So the equation can check the information about the discretisation of a term such as for example the name of the shape functions, the number of nodes. . . . So the method *getDiscretisationInfosForTerm: aTerm* answers an array containing the number of nodes used for interpolating the field on the element, the name of the shape functions and the name of the nodal values.

(2) The second part of the behavior is the managing of the left and right-hand side of the equation and of the space dimension.

The method *putAllLhs* puts **rhs** at 0, and **lhs** at (**lhs**–**rhs**). The same is true for method *putAllRhs*.

A large amount of the behavior can be summarized as follows: the tasks are decentralized to both attributes **lhs** and **rhs**. Take the example of the method *expand*. The method is:

```
expand
   "Expand both sides of the
receiver"
  self giveLhs expand.
  self giveRhs expand.
```

The same thing is used for methods: *getDiscretizedForm*, *substituteTerm: aTermfor: anExpression*, *replace: anExpression by: expression2*.

(3) The last part of the behavior is the discretization of the equation (method *getDiscretizedForm*) which is delegated to **lhs** and **rhs**. An instance of **DiscretizedEquation** is created here.

*4.1.5.2. Class **DiscretizedEquation**.* This class is the representation of equations for the discrete problem (Table 10).

Attributes

This class represents only a specification of the behavior of the class **IntEquation**. This class does not need new attributes. The basic entities manipulated here are instances of **DiscretizationMatrix**. This

Table 10
Class **DiscretizedEquation**

Inherits from: **IntEquation**, **FEMTheory**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) attributes manipulation	lhs rhs	expand giveLhs giveRhs printString putAllLhs putAllRhs substituteTerm. term1 with. term2

Tasks	Attributes	Methods
(1) manipulation	numberOfNodes numberOfGeometricNodes jacobianMatrix	findAllUnknowns findMatrixCorrespondingToBodyLoads findMatrixCorrespondingToSurfaceLoads findMatrixCorrespondingToUknown getArrayOfFunctions getCoordinatesFunctions getJacobianMatrix getNumberOfGeometricNodes getNumberOfNodes giveJacobianMatrix giveNumberOfGeometricNodes giveNumberOfNodes giveNumberOfRows invokeLinearIndependence isDiscretizeEquation isIntEquation replaceYourselfUsingDictionary transpose
(2) creation of methods		createCPlusMethodsIn pathName forElement. name createMethodsInElement: aClass

class corresponds to class **IntEquation** for the discretized problem. For this reason, attributes **numberOfNodes**, **numberOfGeometricNodes** and **jacobianMatrix** are given to this class. The jacobian matrix which is the representation of the change of variable from local to global coordinates for the element can be considered as a characteristic of the finite element formulation. Every object can request it, e.g. integrals in order to compute themselves when the integrand is expressed using local coordinates (the integration is done on the global domain).

Task description

(1) This class has only two methods of manipulating inherited attributes. The equation can identify its product term by term. It corresponds to the method *invokeLinearIndependence* of **DiscretizedExpression**. As the user manipulates equations, he only sends messages to it. The behavior is split between the left and right-hand sides of the equation.

The principle is the same for the method *transpose*.

```
transpose
  "Transpose the receiver"
self giveLhs transpose.
self giveRhs transpose.
```

(2) The other method makes it possible to create FEM code. The tasks are delegated to the **lhs** and **rhs**.

4.1.6. *Class System*

This class is the representation of the system of equations, for the discrete or the continuous problem (Table 11).

Attributes

Generally speaking, the result of the identification of each term of the equation is a system of equations. The following system for instance, results from the derivation of the equations of Stokes flow:

$$\begin{cases} Kd + Gp = f \\ G'd + Mp = h \end{cases}$$

Consequently, the main attribute of class **System** is **equationsCollection**. This is an instance of **OrderedCollection**. Moreover, the system knows the total number of nodes and geometric nodes, **numberOfNodes** and **numberOfGeometricNodes**, the dimension of the space and of the problem, **spaceDimension** and **problemDimension**, two arrays containing the characteristics used in the problem (data which will be found in the data file during the numerical computation), **arrayOfConstants** and **arrayOfConstantsInstanciation**, the jacobian matrix (see class **Expression**), **jacobianMatrix**, and the type of numerical integration scheme for computing the integrals in the numerical code, **numericalIntegrationScheme**. Note that each integral can be computed in a different way if desired.

Task description

(1) The method *transpose* permits to *transpose each equation of the system. The tasks are* decentralized to each instance of **Equation**. The method *replaceYourselfUsingDictionary: aDictionaryOfFunctions* replaces each instance of **Term** given in the dictionary by the corresponding values. The replacement operation is decentralized to the equations.

(2) The last methods permit the user to create a finite element code. The message *createNewElement* or *createCPlusNewElement* is sent to an instance of the system by the user. The tasks are then particular to each type of code created. Generally speaking, this object has to prepare the creation of further code, in for example creating a class with the new element name for the Smalltalk version, or creating first header and source files for the C++ version. The rest of the behavior is decentralized to the equations. It shows that every object does what it has to (according its attributes) and nothing else.

Table 11
Class System

Inherits from. **FEMTheory**, **Object**		
Inherited tasks	Inherited attributes	Inherited methods
creation		
Tasks	Attributes	Methods
(1) managing attributes	numberOfNodes spaceDimension problemDimension jacobianMatrix	replaceYourselfUsingDictionary. aDictOfFunct transpose
(2) code creation	arrayOfConstants arrayOfConstantsInstanciation *equationsCollection* listOfUnknowns listOfTerms numericalIntegrationScheme	assembleLhs assembleRhs *assembleYourEquations* buildMatrixCorrespondingToBodyLoads buildMatrixCorrespondingToSurfaceLoads buildMatrixCorrespondingToUnknown createArrayOfConstantsForElement. createCPlusGaussPointsInstanciationMethodsIn: forElement createCPlusHeaderFileEndIn: forElement createCPlusHeaderFileIn: forElement. createCPlusSourceFileIn. forElement createInstanciationMethodIn createMethodsIn createNewCPlusElement createNewElement findAllUnknowns giveCPlusIncludeSourceFileName: *giveGaussPointArrayFor.* giveGaussWeightArrayFor: isSystem transpose

4.1.7. *Class Term and its subclasses*

4.1.7.1. *Class* ***Term***. As described in the first part, this class implements the smallest entity of an *expression. The choice made here is to have this general class for all sorts of terms, a gradient of a* vector field, e.g. $u_{i,j}$, a number, e.g. 5, or a constant, e.g. *E*. Another possible approach, would have been to create specific classes for each type of term, each of them implementing its own behavior. As the need of specific behaviors appears late and locally, it seems better, at this state of the development, to have only one class which describes all the terms. This implies giving the class Term the methods to analyze itself. The second part of the tasks this class implements is the discretisation of the term. Finally, as the *term is involved in the manipulation of expressions*, some methods will implement this behavior (Table 12).

Attributes

Class **Term** has six attributes. The term is described by its **name**, its **indices**, its **derivationIndices** and its **timeDerivationIndices**. All of them are instances of class **String**. As shown, they make it possible to describe all the terms needed in the first and the second steps of the procedure.

Another characteristic of the term is the **field**. The term may know if it is an unknown or not, and if its type is «admissible field» or «zero admissible field». The attribute **field** is an instance of class **Field**.

Table 12
Class **Term**

Inherits from **FEMTheory**, **Object**

Inhibited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow. term
Tasks	Attributes	Methods
(1) manipulation	–	+, addDist., *, multDist, = addYourselfToSumListTo aSum existIn: prodList expand substituteTerm: term1 with: term2 computeMatrixLawDimension getMMatrix printString deriveWithRespectToVariable i
(2) analysis	name derivationIndices indices field timeDerivationIndices	getYourNature isEquivalentTo aTerm isDivTerm is0KAF isKAF isTimeConstant isTimeFirstDerivative isAnUnknown
(3) discretization	discretization discretizationInfosArray	buildBSecondMatrixWithCharacteristics charact dimensions: point buildCMatrixWithCharacteristics: charact dimensions point buildNMatrixWithCharacteristics, charact dimensions point buildUnknownMatrixWithCharacteristics charact dimensions point getDiscretizedForm getScalarDiscetization getScalarDerivedDiscretization getVectorDiscretization

Finally, the term knows its discretization, attribute **discretization**, and the array giving the characteristics of the discretization (an array of dimension 4, containing the name of the shape functions, the name of the nodal unknown, the start and the end of the numbering).

Task description

(1) As the term is the basic entity which forms an expression, it is given methods to operate on itself during manipulations. Three types of behavior appear here: derivation, expansion, substitution

In the derivation operations, the term is able to derive itself. The variable of derivation may be a term, an integer, or an index (a lower case).

The derivation of a term with respect to a term answers *0* if the variable of derivation is not appropriate or answers *1* if it is. For example, deriving *X* with respect to variable *Y* yields *0* and with respect to variable *X* yields *1*.

If a term is derived with respect to an index, here a lower case or a number, the string

derivationIndices is added the lower case index. Deriving term N_2 with respect to *1* yields $N_{2,1}$. The process is the same for the time derivation.

Elementary operations needed for expanding an expression are the mathematical operations + and *, two methods of operation on lists and the method *expand* which returns the expanded form of the term, i.e. the term itself. Adding or multiplying a term by another object, term or expression, amounts to building a simple expression with it and to adding, or multiplying it by the object as an expression. The method which does it is *asExpression*. It instanciates a ***ProdList***, a **SumList**, an **Expression**, and adds the term to it. (see class **Expression**).

The term can determine if it is in a list (here an instance of **ProdList**) and can be added to an instance of **SumList**. This last method requires the creation a **ProdList**, to add itself to it, and add the product to the sum.

(2) In the manipulation operation, the object works on its description attributes: **name**, **indices**, **derivationIndices**, **timeDerivationIndices**. The analysis of the term is based on the reading of the different indices. The method *getYourNature* tells if the term is a scalar, a vector, the divergence or the gradient operator applied to a vector field, a second- or fourth-order tensor. So it gives the field's type (scalar, vector, tensor) and the operator applied to it. This method returns a string which says: 'scalar', 'scalarDerived', or 'divergence', . . . For example, the operator divergence applied to a vector field is recognized because the size of both strings **derivationIndices** and **indices** is equal to one and their attributes are identical as shown in Fig. 3.

The same principle makes it possible to recognize the other fields and the operators applied to them. The term can also determine how many times it is derived with respect to variable 't', the *time index*. *It only* computes the length of the string **timeDerivationIndices**.

One of the problems is to determine if two terms are equivalent or not. With the above tools it is just necessary to verify if they have the same name and if the method *getYourNature* returns the same string for both instances. For example, the term $u_{i,j}$ is equivalent to $u_{k,l}$, but not to u_i or $v_{i,j}$.

Finally, the term is able to identify its field, instance of **Field**. It can find if the field is a zero admissible field (virtual field) or not, or an unknown of the problem or not.

(3) One of the most important tasks the term has to do, is to discretize itself. It knows how to do and how to find the information if needs.

The first part of the method *getDiscretizedForm* consists in analyzing the term using method *getYourNature* and according to the answer send itself the message corresponding to the type of term. The result is a product of matrices (instances of **DiscretizationMatrix**) which contains the corresponding symbolic forms. For example the term $u_{i,j}$ is a gradient, the method which builds the discretisation is *getGradDiscretization*. This method returns the product *Bd* which is an instance of **ProdList** containing two instances of **DiscretizationMatrix**. The term in this method builds the instances *B* and *d*. The method *buildBMatrixWithCharacteristics: array4 dimension: point* builds the matrix B (see [16]) and the method *buildUnknownMatrixWithCharacteristics: array4 dimension: dim* builds the matrix corresponding to the nodal unknowns. The principle is the same for all types of terms. Note that a field may be a data of the problem such as the body loads, or an unknown, such as *U*, the displacement field for example. For a vector field, the method *getVectorDiscretization* checks if the field is an unknown of the problem or not, the

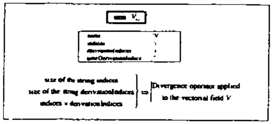

$V_{i,i}$
V
i
i
size of the string indices
size of the string derivationIndices
indices = derivationIndices
⇒
Divergence operator applied
to the vectorial field V

Fig 3 Divergence operator

equation knows its unknowns, and sends the appropriate message *self getVectorDiscretizationUnknown* or *self getVectorDiscretizationKnown*. All the information needed in order to obtain the components are asked of the attribute **hierarchicParent** if the term does not have them. This behavior is inherited from the class **FEMTheory**. This behavior introduces a new class, subclass of **Term**, the class **DiscretizationMatrix**.

4.1.7.2. Class ***DiscretizationMatrix***. As the term is the basic entity of the expression, the elementary matrix is the basic entity of the discretized expression (Table 13).

Table 13
Class **DiscretizationMatrix**

Inherits from: **Term**, **FEMTheory**, **Object**

Inherited tasks	inherited attributes	inherited methods
(1) creation		
(2) access to data of the hierarchic parent	hierarchicParent	giveHierarchicParent giveSpaceDimension
(3) manipulation	-	addYourselfToSumListTo. aSum existIn: prodList expand substituteTerm: term1 with: term2
(4) analysis	timeDerivedIndices	isTimeConstant isTimeFirstDerivative
Tasks	Attributes	Methods
(1) manipulation	transpositionIndex name elementaryMatrix	printString transpose *giveElementaryMatrix* integrateYourComponentsOnDomain. domain substituteTerm. term1 with. term2
(2) analysis	domain matrixType	isBodyMatrix isSurfaceMatrix isUnknown isConstitutiveMatrix
(3) creation of code	numericalIntegrationScheme	createCPlusGaussPointsInstanciationMethodsIn: path forElement: anElemt *createCPlusGaussPointsMethodsIn: path* forElement: anElmt createCPlusMethod: symbol in: path forElenent: an Elmt createCPlusMethodGaussianIntegration: symbol in: path forElement: anElemt createGaussPointsAttributesMethodsIn: path createGaussPointsInstanciationMethodsIn: path createGaussPointsMethodsIn. path createMethod: symbol inElement: path createMethodGaussianIntegration. symbol inElement: path createMethodWithArgument. symbol inElement. anElemt createMethodWithArgumentGaussianIntegration: symbol inElement: anElemt

Attributes

The only inherited attributes used here are **name** and **timeDerivationIndices**. Its specific attributes are **elementaryMatrix**, an instance of **Matrix**, **transpositionIndex**, which indicates if the matrix is transposed or not (if it is, this attribute is string 't'), **matrixType**, which indicates if it is a constitutive matrix (index 'L') for example, and **domain**, which indicates the domain on which the matrix is defined (surface or body matrix).

Task description

This class inherits from its superclass all the behavior concerning the manipulation in the context of an expression.

(1) The method *transpose* puts the index at 't' if it is nil, and puts it at nil if it is 't' and transposes the elementary matrix. For example, consider the matrix

$$d = \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} .$$

The message *d transpose* returns *d'* and the message *d'giveElementaryMatrix* returns

$$\begin{bmatrix} d_{11} & d_{21} \\ d_{12} & d_{22} \end{bmatrix} .$$

The attribute transpositionIndex is only useful to indicate the modification of the matrix which has been transposed.

The method *substituteTerm: aTerm by: anExpression* substitutes *aTerm* in all the components of the elementary matrix. This method plays on the polymorphism for the considered selector.

(2) The **DiscretizationMatrix** class has an attribute called **matrixType**. It can be the string 'L' or the string 'U', which respectively mean 'unknown' and 'constitutive law'. The methods **isUnknown** and **isConstitutiveMatrix** answers **true** or **false** depending on the value of the index.

The matrix can determine if the components of its elementary matrix are evaluated on the domain or on its boundary. This operation is useful to determine if a load vector is a surface load vector or a body load vector. This method asks its components to give back the domain on which *they are applied, whatever their* class is (**Integral**, **Expression**, . . .).

(3) The methods to create a code in Smalltalk are multiple. This results from the fact that in an instance of **DiscretizedEquation** the identification of different elemental matrices which are implicated in code creation (M^e, K^e, f^e) cannot be made with the matrix. For example, the mass matrix is identified through the product $Md^{\cdot\cdot}$, which contains a second time derivative. So the general method *createMethod: aSelector inElement: elmt* gives the choice of the selector, and the methods *createBodyLoadMethodInElement: elmt* and *createSurfaceLoadMethodInElement: elmt* create specific methods. The method *createMethodsInElement: elmt* asks each component of the **elementaryMatrix** to create its own parts of the code. As usual, a similar scheme exists for the C++ version.

4.2. *Class* ***OrderedCollection*** *and subclasses*

4.2.1. *Class* ***FEMTheoryOrderedCollection***

The class **FEMTheoryOrderedCollection** plays the same role as the class **FEMTheory**. It groups all lists used in the new environment and is never instanciated. These lists are chosen as subclasses of **OrderedCollection**. The reasons for this choice are that objects must be stored in a given order (either because the user is waiting for the terms in the same order he has introduced them, or the expression does not respect the commutativity of the product, as for example the expression of matrix $d^tKd + d^tMd_{,tt}$) and that the same object may be stored more than once (could not be a subclass of class **Set** which only allows one occurrence of an element). No instance of this class is expected. But the behavior of the superclass may be particularized. For example the method *add:anObject* adds *anObject* to the receiver if *anObject* is not *nil* (method *add:* of the super class); otherwise it does nothing (Table 14).

Table 14
Class **FEMTheoryOrderedCollection**

Inherits from: **Ordered Collection**, **Indexed Collection**, **Collection**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
Tasks	Attributes	Methods
(1) particularizing tasks		add anObject addIfAbsent: anObject addAllIfAbsent. anObject addLast: anObj removeObject. obj
(2) attributes managing	hierarchicParent	findAllUnknowns giveArrayOfConstants giveArrayOfConstantsInstanciation giveCPlusIncludeSourceFileName· giveHierarchicParent giveJacobianMatrix giveNumberOfGeometricNodes giveNumberOfNodes giveNumericalIntegrationScheme giveProblemDimension giveSpaceDimension giveTimeDerivationIndices giveUnknownsFEMCol hierarchicParent: occurrencesOf anObj

Attributes

As the class **FEMTheory** this class is given the attribute **hierarchicParent**. It will be inherited by all subclasses and particularly classes implied in the building of expressions.

Task description

(1) This class is given the behavior attached to the attribute **hierarchicParent**. The subclasses inherit from it the behavior corresponding to the management of this attribute, method *giveHierarchicParent* and instanciation method *hierarchicParent: anObject*.

(2) It is also given the capability to remove an object from the collection and to count the occurrences of an object in the collection. The tasks of adding objects to a collection are particularized by the fact that the attribute **hierarchicParent** of the object added is instanciated with the collection.

*4.2.2. Class **EquationsCollection***

This class represents the list of equations needed for class **System** (Table 15). This class has no particular attributes.

Task description

Most tasks are inherited from the super class, in particular the behavior linked to the attribute **hierarchicParent**.

The only behavior is decentralizing the tasks of transposition and the invocation of the linear independence of each coefficient of the equations.

*4.2.3. Class **ExpressionsLists** and its subclasses*

*4.2.3.1. Class **ExpressionsLists**.* This class regroups both lists used in the representation of expressions. No instance of this class is expected. Subclasses, are **SumList** and **ProdList** (Table 16).

Table 15
Class **EquationsCollection**

Inherits from: **FEMTheoryOrderedCollection**, **OrderedCollection**, **IndexedCollection**, **Collection**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) particularizing tasks		add: anObject addIfAbsent: anObject addLast
(2) managing attributes	hierarchicParent	
Tasks	Attributes	Methods
decentralizing tasks to the instances of class **Equation**		invokeLinearIndependence transpose

Table 16
Class **ExpressionsLists**

Inherits from: **FEMTheoryOrderedCollection**, **OrderedCollection**, **IndexedCollection**, **Collection**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) particularizing tasks		add: anObject addIfAbsent: anObject addLast
(2) managing attributes	hierarchicParent	giveHierarchicParent hierarchicParent: anObject occurrencesOf: anObject removeObject: anObject
Tasks	Attributes	Methods
access to data of the hierarchic parent		computeMatrixLawDimension getDiscretizationInfosForTerm. term getListOfTerms giveSpaceDimension knowsAsUnknow: term

Attributes

These inherited from its superclass the attribute **hierarchicParent**.

Task description

The only proper behavior, which is inherited by its subclasses, concerns the access to data which is expected to be sent back by the attribute **hierarchicParent**. This is the general behavior of each class which cannot answer directly but asks its own **hierarchicParent** to return it.

*4.2.4.3. Class **ProdList**.* This class is concerned with the modelling of the products needed to build expressions (Table 17).

General remarks

(a) This collection can be implied in expressions, functionals, discretised expressions. It is given all the behaviors for those contexts. There is not any anticipation of the content or context (instances of **Integral** for **Functional** or instances of **Term** for **Expression**). As these lists have no particularly complex tasks to realize, just specialized list processing (arrangement, research of objects, . . .), only one type of list has been created to implement the behavior of products.

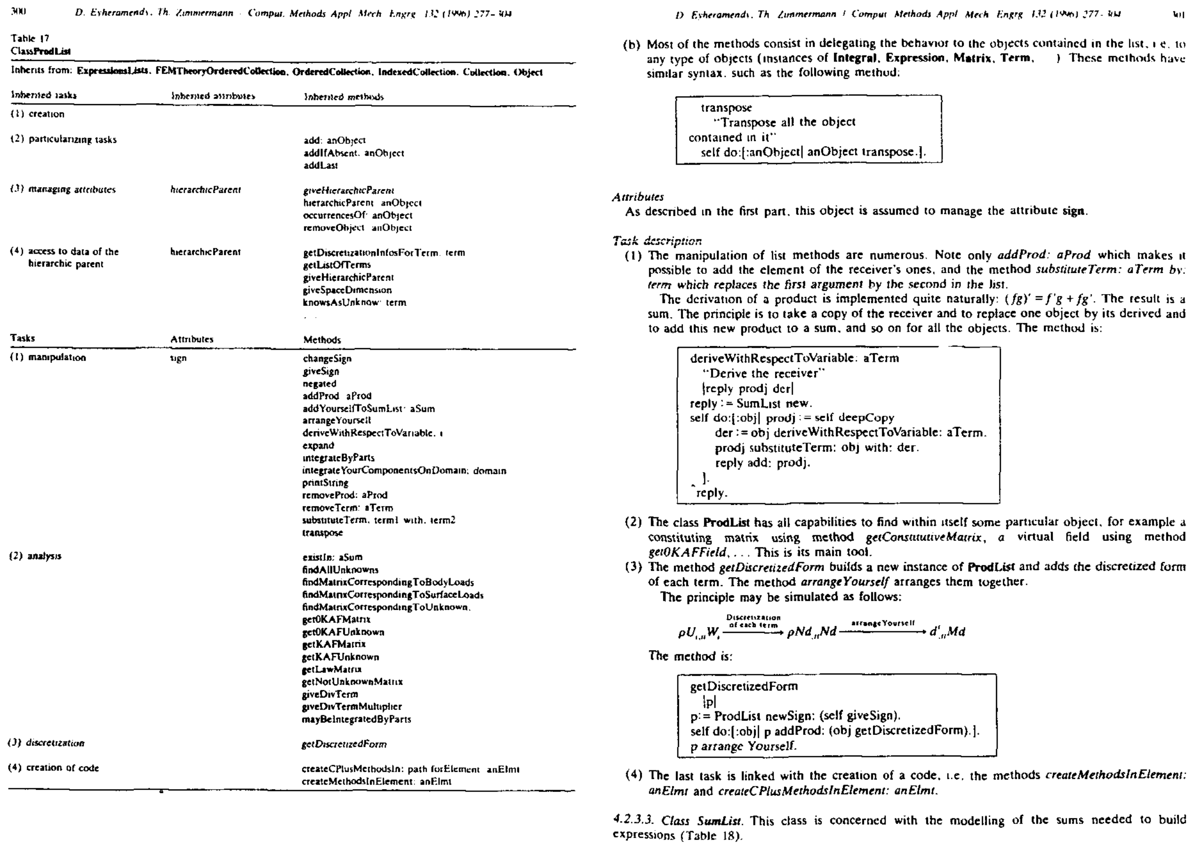


Table 17
Class**ProdList**

Inherits from: **ExpressionsLists**, **FEMTheoryOrderedCollection**, **OrderedCollection**, **IndexedCollection**, **Collection**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) particularizing tasks		add: anObject addIfAbsent. anObject addLast
(3) managing attributes	*hierarchicParent*	*giveHierarchicParent* hierarchicParent anObject occurrencesOf· anObject removeObject anObject
(4) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm. term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow· term . . .

Tasks	Attributes	Methods
(1) manipulation	sign	changeSign giveSign negated addProd aProd addYourselfToSumList· aSum arrangeYourself deriveWithRespectToVariable. t expand integrateByParts integrateYourComponentsOnDomain: domain printString removeProd: aProd removeTerm: aTerm substituteTerm. term1 with. term2 transpose
(2) analysis		existIn: aSum findAllUnknowns findMatrixCorrespondingToBodyLoads findMatrixCorrespondingToSurfaceLoads findMatrixCorrespondingToUnknown. get0KAFMatrix get0KAFUnknown getKAFMatrix getKAFUnknown getLawMatrix getNotUnknownMatrix giveDivTerm giveDivTermMultiplier mayBeIntegratedByParts
(3) discretization		*getDiscretizedForm*
(4) creation of code		createCPlusMethodsIn: path forElement anElmt createMethodsInElement: anElmt

(b) Most of the methods consist in delegating the behavior to the objects contained in the list, i.e. to any type of objects (instances of **Integral**, **Expression**, **Matrix**, **Term**,) These methods have similar syntax, such as the following method:

```
  transpose
    "Transpose all the object
contained in it"
  self do:[:anObject| anObject transpose.].
```

Attributes

As described in the first part, this object is assumed to manage the attribute **sign**.

Task description

(1) The manipulation of list methods are numerous. Note only *addProd: aProd* which makes it possible to add the element of the receiver's ones, and the method *substituteTerm: aTerm by: term which replaces the first argument* by the second in the list.

The derivation of a product is implemented quite naturally: $(fg)' = f'g + fg'$. The result is a sum. The principle is to take a copy of the receiver and to replace one object by its derived and to add this new product to a sum, and so on for all the objects. The method is:

```
deriveWithRespectToVariable: aTerm
  "Derive the receiver"
  |reply prodj der|
reply := SumList new.
self do:[:obj| prodj := self deepCopy
    der := obj deriveWithRespectToVariable: aTerm.
    prodj substituteTerm: obj with: der.
    reply add: prodj.
  ].
^reply.
```

(2) The class **ProdList** has all capabilities to find within itself some particular object, for example a constituting matrix using method *getConstitutiveMatrix*, *a* virtual field using method *get0KAFField*, . . . This is its main tool.

(3) The method *getDiscretizedForm* builds a new instance of **ProdList** and adds the discretized form of each term. The method *arrangeYourself* arranges them together.

The principle may be simulated as follows:

$$\rho U_{i,ii} W_i \xrightarrow{\text{Discretization of each term}} \rho Nd_{,ii} Nd \xrightarrow{\text{arrangeYourself}} d'_{,ii} Md$$

The method is:

```
getDiscretizedForm
  |p|
p:= ProdList newSign: (self giveSign).
self do:[:obj| p addProd: (obj getDiscretizedForm).].
p arrange Yourself.
```

(4) The last task is linked with the creation of a code, i.e. the methods *createMethodsInElement: anElmt* and *createCPlusMethodsInElement: anElmt*.

4.2.3.3. *Class* ***SumList***. This class is concerned with the modelling of the sums needed to build expressions (Table 18).

Table 18
Class **SumList**

Inherits from: **ExpressionsLists**, **FEMTheoryOrderedCollection**, **OrderedCollection**, **IndexedCollection**, **Collection**, **Object**

Inherited tasks	Inherited attributes	Inherited methods
(1) creation		
(2) particularizing tasks		add: anObject addIfAbsent: anObject addLast
(3) managing attributes	hierarchicParent	giveHierarchicParent hierarchicParent: anObject occurrencesOf: anObject removeObject: anObject
(4) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm: term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow: term

Tasks	Attributes	Methods
(1) manipulation		addYourselfToSumList: aSum deriveWithRespectToVariable: t expand expandYourExpressions integrateByParts integrateYourComponentsOnDomain: domain negated printString replace: exp1 by: exp2 substituteTerm: term1 with: term2 transpose
(2) analysis		findAllUnknowns findMatrixCorrespondingToBodyLoads findMatrixCorrespondingToSurfaceLoads findMatrixCorrespondingToUnknown: aTerm giveDivTerm giveDivTermMultiplier mayBeIntegratedByParts
(3) discretization		getDiscretizedForm
(4) creation of code		createCPlusMethodsIn: path forElement: anElmt createMethodsInElement: anElmt

General remarks

This class has no attributes. Its existence justifies itself through its behavior. Most of the methods consist in delegating part of the tasks to the objects contained in the list, i.e. to instances of **ProdList**. The list simply does what it has to do and nothing else. Take the example of the method *expand*:

```
expand
  "Expand a sum"
self do:[: prod | prod expand.]
self expandYourExpressions.
```

The product is asked to expand itself (behavior decentralized to the product) and the sum performs an operation on its own. This is an illustration of how pleasant object oriented programming can be.

Task description

The behavior is split in the same way as the class ProdList.

(1) First come methods which manipulate the list. The method *addYourselfToSumList: aSum* adds all the elements of the receiver to the argument. In method *expandYourExpressions*, the products are asked to add the sum they contain, if any, in a sum. This method enters the expansion process and can be illustrated as follows:

$$((ax+b)+(cy+d)) \xrightarrow{\text{expandYourExpression}} (ax+b+cy+d)$$

where the brackets highlight the sums of products. It is interesting to note that the method *deriveByVariable: aTerm* is programmed in a quite natural way, i.e. as a sum: the derivative of a sum is the sum of the derivatives. *In this method the product is asked its derivatives, and is added* to the reply if not nil. The method:

```
deriveWithRespectToVariable: aTerm
  |reply der|
reply := SumList new.
self do:[:plist| der := plist deriveWithRespectToVariable: aTerm.
      (der isNul)
            ifFalse:[ reply add: der.].
 ].
^reply
```

The methods of integration by parts is decentralized to the product such as transposition.

(2) In this part all the behavior is decentralized to the products. The check method *mayBeIntegratedByParts* answers true or false after verification in product and methods *giveDivTerm* and *givedivTermMultiplier* returns the first operator divergence found.

(3) The method *getDiscretizedForm* forwards the same request to the products.

(4) Code generation is decentralised to the product lists contained in the sum.

4.3. *Methods added to an existing class: class **String***

The class **String** is given methods to build expressions by itself. The user has only to write a string representing the expressions, respecting the following convention: an upper case represents the name of a field, a lower case an index. For example the string 'Sij,j + Fi' represents the expression $S_{ij,j} + F_i$. Then the method *formYourExpression* builds the corresponding instance of **Expression**, i.e. instances of **ProdList** and **SumList**. The principle is to cut the string in pieces representing terms and to instanciate **Term** using the method *getYourTerm* and adding it to a list. At this stage of the development expressions can be formed easily and then used to generate complex structures: integrals, functionals equations, . . .

5. Concluding remarks

The new environment described here makes it possible, starting from differential forms, to derive matrix formulations using a Galerkin approximation for the Finite Element Method, and finally to generate a new class corresponding to a new element in an object-oriented code, presently in Smalltalk or C++. The generation of C++ code permits to achieve an improved efficiency.

In the structure described here objects appear which are manipulated in a quite natural way, supporting the idea that Smalltalk is a natural extension of thinking. It shows moreover that symbolic capabilities in Smalltalk can enhance fast prototyping.

The proposed approach has been tested on simple examples of derivation: thermal problems, elastodynamic problems, dynamic uniaxial bars, Navier–Stokes flow. These examples have highlighted the potential of this approach and will be discussed in a companion paper. The weak link of the system is the lack of performance of the symbolic computation of shape functions at least so far. The expressions which are manipulated become very large and the lack of efficiency of Smalltalk could become a handicap for some problems. The aim was not to build an efficient environment for symbolic computations.

The environment can however be extended to solve other forms of the Finite Element Method and the same approach can easily be extended to alternative formulations such as boundary element methods, collocation, etc.

Acknowledgment

This research work was supported by the Swiss National Science Foundation under Grant No. 21-402090.94.

References

[1] Y. Dubois-Pèlerin, P Bomme and Th. Zimmermann, Object-oriented finite element programming concepts, Proc European Conference on New Advances in Computational Structural Mechanics, P Ladevèze and O C Zienkiewicz, eds (Elsevier Science Publishers, Amsterdam, 1991) 95–101

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

[3] Y. Dubois-Pèlerin, Th Zimmermann and P. Bomme, Object-oriented finite element programming: II A prototype program in Smalltalk, Comput Methods Appl. Mech. Engrg. 98 (1992) 361–397.

[4] Y. Dubois-Pèlerin and Th. Zimmermann, Object-oriented finite element programming: III. An efficient implementation in C++, Comput. Methods Appl. Mech. Engrg. 108 (1993) 165–183.

[5] Th. Zimmermann and D Eyheramendy, Object-oriented finite elements. I Principles of symbolic derivations and automatic programming, Comput. Methods Appl Mech Engrg. 132 (1996) 259–276

[6] N.S. Bardel, The application of symbolic computing to the hierarchical finite element method, Int J Numer Methods Engrg. 28 (1989) 1181–1204.

[7] M.M. Cecchi and C. Lami, Automatic generation of stiffness matrices for finite element analysis, Int J Numer Methods Engrg. 11 (1977) 396–400

[8] A.R. Korncoff and S.J. Fenves, Symbolic generation of finite element stiffness matrices, Comput. Struct. 10 (1979) 119–124.

[9] S.V. Hoa and S. Sankar, A computer program for automatic generation of stiffness and mass matrices in finite-element analysis, Comput. Struct. 11 (1980) 147–161.

[10] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in structural mechanics-progress and potential, Comput. Struct. 10 (1979) 95–118

[11] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in nonlinear finite element analysis, Comput. Struct. 13 (1981) 379–403.

[12] P.S. Wang, FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis, J. Symbolic Computation 2 (1986) 305–316.

[13] G. Masini, A. Napoli, D Léonard and K. Trombe, Les langages à objets (InterEditions, 1990)

[14] Smalltalk for Win32, Reference guide (Digitalk Inc , 1993)

[15] Smalltalk for Win32, Encyclopedia of classes (Digitalk Inc., 1993)

[16] T.J R Hughes, The Finite Element Method (Prentice-Hall, Englewood Cliffs, NJ, 1987)

[17] Th. Zimmermann, Y. Dubois-Pèlerin and P Bomme, Object-oriented finite element programming Theory and Smalltalk V implementation for FEM_Object$_{\mathrm{st}}$™001 (Elmepress International, 1992).

[18] Y Dubois-Pèlerin and Th. Zimmermann, Object-oriented finite element programming: Theory and C++ Implementation for FEM_Object$_{C++}$™ 001 (Elmepress International, 1992)

INFORMATION FOR CONTRIBUTORS

Manuscripts should be sent in triplicate to one of the Editors All manuscripts will be refereed Manuscripts should preferably be in English They should be typewritten, double-spaced first copies (or clear Xerox copies thereof) with a wide margin Abstracts, footnotes and lists of references should also be double-spaced All pages should be numbered (also those containing references tables and figure captions) Upon acceptance of an article, author(s) will be asked to transfer copyright of the article to the publisher This transfer will ensure the widest possible dissemination of information

Abstracts
The text of a paper should be preceded by a summary in English This should be short, but should mention all essential points of the paper

Figures and tables
The drawings for the figures must be originals, drawn in black India ink in large size and carefully lettered, or printed on a high-quality laser printer The lettering as well as the details should have proportionate dimensions, so as not to become illegible or unclear after the usual reduction by the printers, in general, the figures should be designed for a reduction factor of two or three Mathematical symbols should be entered in italics where appropriate Each figure should have a number and a caption, the captions should be collected on a separate sheet The appropriate place of a figure should be indicated in the margin. Tables should be typed on separate sheets. Each table should have a number and a title The appropriate places for the insertion of tables should be indicated in the margin Colour illustrations can be included and will be printed in colour at no charge if, in the opinion of the Editors, the colour is essential If this is not the case, the figures will be printed in black and white unless the author is prepared to pay the extra costs arising from colour reproduction

Formulae
Displayed formulae should be numbered and typed or clearly written by hand. Symbols should be identified in the margin, where they occur for the first time

References
In the text, reference to other parts of the paper should be made by section (or equation) number, but not by page number References should be listed on a separate sheet in the order in which they appear in the text

COMPLETE INSTRUCTIONS TO AUTHORS are published in every last issue of a volume, and copies can also be obtained from the Editors and the Publisher, Elsevier Science B V , P O Box 1991, 1000 BZ Amsterdam, The Netherlands

Instructions for LaTeX manuscripts
The LaTeX files of papers that have been accepted for publication may be sent to the Publisher by e-mail or on a diskette (3.5" or 5.25" MS-DOS) If the file is suitable, proofs will be produced without rekeying the text The article should be encoded in Elsevier LaTeX, standard LaTeX, or AMS-LaTeX (in document style "article") The Elsevier-LaTeX package together with instructions on how to prepare *a file, is available from the Publisher This package can also be obtained through the Elsevier WWW home page* (http //www elsevier nl/), or using anonymous FTP from the Comprehensive TeX Archive Network (CTAN). The host-names are ftp.dante.de, ftp tex.ac uk, ftp shsu edu, the CTAN directories are /pub/tex/macros/latex209/contrib/elsevier, /pub/archive/macros/latex209/contrib/elsevier, /tex-archive/macros/latex209/contrib/elsevier, respectively *No changes from the accepted version are permissible, without the explicit approval of the Editor The Publisher reserves the right to decide whether to use the author's file or not* If the file is sent by e-mail, the name of the journal should be mentioned in the "subject field" of the message to identify the paper Authors should include an ASCII table (available from the Publisher) in their files to enable the detection of transmission errors

The files should be mailed to Rob van Fucht, Elsevier Editorial Services, Mayfield House, 256 Banbury Road, Oxford OX2 7DH, UK Fax +44-1865-314990 E-mail, r fucht@elsevier co.uk

INFORMATION FOR SUBSCRIBERS
COMPUTER METHODS IN APPLIED MECHANICS AND ENGINEERING (ISSN 0045-7825) is published in eleven volumes (44 issues) a year For 1996 Volumes 129–139 are scheduled for publication Subscription prices are available upon request from the publisher Subscriptions are accepted on a prepaid basis only and are entered on a calendar year basis. Issues are sent by surface mail except to the following countries where air delivery (S A L – Surface Air Lifted) is ensured: Argentina, Australia, Brazil, Canada, Hong Kong, India, Israel, Japan (+25% normal p.p h.), Malaysia, Mexico, New Zealand, Pakistan, P R China, Singapore, South Africa, South Korea, Taiwan, Thailand, USA For the rest of the world *airmail charges are available upon request Claims for missing issues will be honoured free of charge* within six months after the publication date of issues. Mail orders and inquiries to: Elsevier Science S A , P O Box 564, CH-1001 Lausanne, Switzerland

Special regulations for authors
Upon acceptance of an article by the journal, the author(s) will be asked to transfer copyright of the article to the publisher This transfer will ensure the widest possible dissemination of information

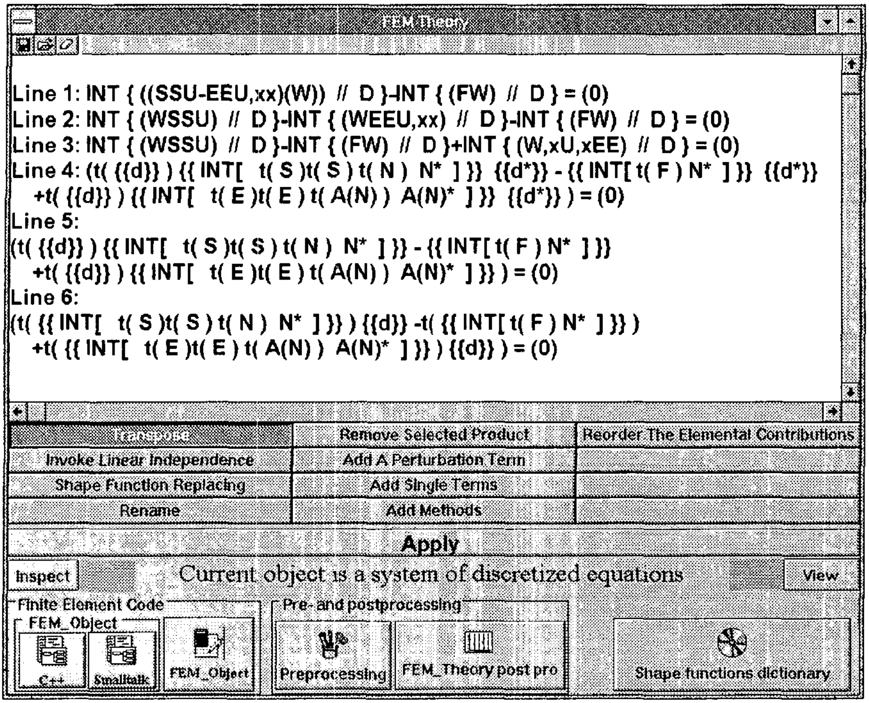
Appendix B - Analysis of a one-dimensional diffusion equation

In this part, the aim is to show a fast evaluation of different formulations on a simple scalar model equation. This part is based on the study of a stabilized formulation presented in [FRA 89]. The goal is to test different formulations on the simple linear diffusion problem $\sigma^2 u - \varepsilon^2 u_{,xx} = f$, for small ratios of ε^2 / σ^2 . First, a standard Galerkin method will be built ; then, a Galerkin Least Squares formulation; finally, a Galerkin/gradient Least Squares formulation. Linear and quadratic interpolations will be tested here.

Consider the following problem :
Given *f*, find *u(x)* with appropriate continuity conditions satisfying :
$\sigma^2 u - \varepsilon^2 u_{,xx} = f$ on $[0,1]$
with boundary conditions : $u(0) = 0$ and $u(1) = 2$
where σ and ε are real constant parameters and $u_{,xx}$ is the second derivative of u with respect to x.

Galerkin formulation

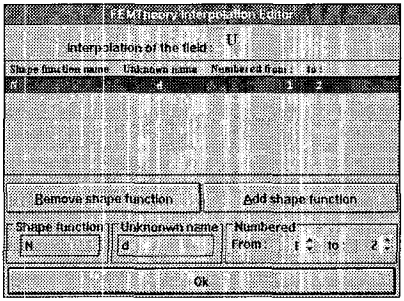
The matrix form derivation is shown in Figure 103. On line 1, the variational formulation is posted: «S» and «E» are two constants (sigma and epsilon), «F» represents the body loads, «W» is the test function, «U» is the solution and «U,xx» is the second derivative of the scalar function «U» with respect to the variable «x». The formulation is expanded on line 2, and the integral «INT{(WEEU,xx) / D» is integrated by parts, which gives the weak form of line 3. Note that the result is «-INT{(W,xU,xEE) / D», the integrated part is zero and so doesn't appear. On line 4 the discretized form of the weak form of line 3 is shown. Two different derivations are made, a first one for a piecewise linear interpolation for «U» and «W» (see Figure 104), a second one for a piecewise quadratic one (see Figure 106). Line 5 is obtained after invoking the arbitrariness of the virtual field «W», and line 6 is obtained after transposition. The choices of the shape functions corresponding to Figure 104 and Figure 106 are shown respectively in Figure 105 and Figure 107. Two elements are then introduced in the numerical code FEMTheory, Smalltalk version ; numerical efficiency is not an issue in the present case.



```
Line 1: INT { ((SSU-EEU,xx)(W))  //  D }-INT { (FW)  //  D } = (0)
Line 2: INT { (WSSU)  //  D }-INT { (WEEU,xx)  //  D }-INT { (FW)  //  D } = (0)
Line 3: INT { (WSSU)  //  D }-INT { (FW)  //  D }+INT { (W,xU,xEE)  //  D } = (0)
Line 4: (t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }}  {{d*}} - {{ INT[ t( F ) N*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}  {{d*}} ) = (0)
Line 5:
(t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }} - {{ INT[ t( F ) N*  ] }}
   +t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }} ) = (0)
Line 6:
(t( {{ INT[  t( S )t( S ) t( N )  N*  ] }} ) {{d}} -t( {{ INT[ t( F ) N*  ] }} )
   +t( {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }} ) {{d}} ) = (0)
```

Current object is a system of discretized equations

Figure 103 Derivation of the Galerkin formulation for the scalar diffusive equation



Interpolation of the field : U

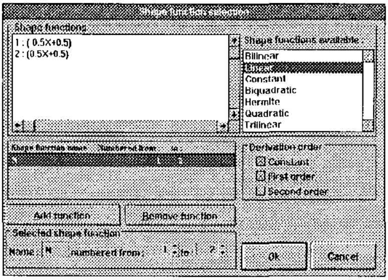
Shape function name	Unknown name	Numbered from :	to :
N	d	1	2

Shape function: N

Unknonwn name: d

Numbered From : 1 to : 2

Figure 104 Piecewise linear interpolation



Shape function selection

Shape functions

1 : (0.5X+0.5)
2 : (0.5X+0.5)

Shape function name	Numbered from :	to :
N	1	2

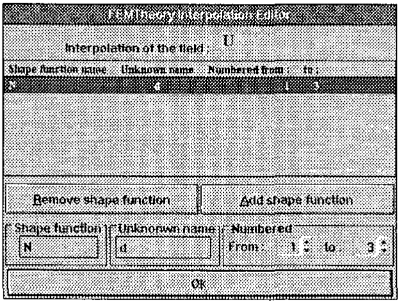
Derivation order

☒ Constant
☒ First order
☐ Second order

Selected shape function

Name : N numbered from : 1 to : 2

Figure 105 Linear shape functions



FEM theory Interpolation Editor

Interpolation of the field : U

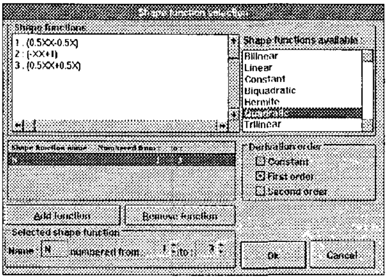
Shape function name	Unknown name	Numbered from :	to :
N	d	1	3

Shape function: N

Unknonwn name: d

Numbered From : 1 to : 3

Figure 106 Piecewise quadratic interpolation



Shape function selection

Shape functions

1 . (0.5XX-0.5X)
2 : (-XX+1)
3 . (0.5XX+0.5X)

Shape function name	Numbered from :	to :
N	1	3

Derivation order

☐ Constant
☒ First order
☐ Second order

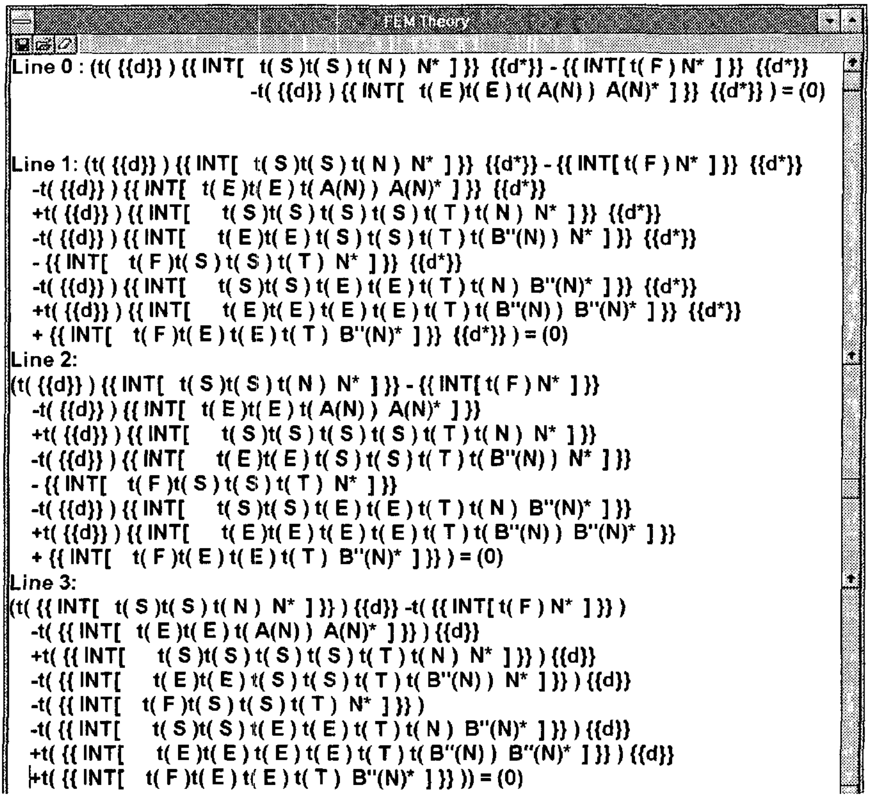
Selected shape function

Name : N numbered from : 1 to : 3

Figure 107 Quadratic shape functions

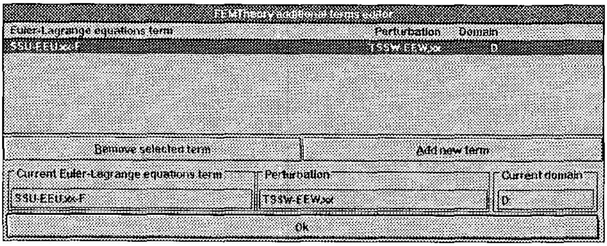
Galerkin Least-Squares formulation

The starting point of the formulation is the Galerkin weak form obtained in the preceding section (see line 4, Figure 103) The formulation is posted in Figure 108, line 0. The stabilization terms introduced are shown in Figure 109. The Lagrange equation «SSU-EEU,xx» is weighted by «TSSW-TEEW,xx» (the discretized form of this product is added to the classical Galerkin form). «T» is a stabilization parameter. The new formulation is shown on line 1. The arbitrariness of the weighting function «{{d*}}» is then invoked (line 2), and the system containing one equation is then transposed (line 3). Two studies are performed, one for linear interpolation and one for quadratic interpolation.



```
Line 0 : (t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }}  {{d*}} - {{ INT[ t( F ) N*  ] }}  {{d*}}
                                                  -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}  {{d*}} ) = (0)


Line 1: (t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }}  {{d*}} - {{ INT[ t( F ) N*  ] }}  {{d*}}
   -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[    t( S )t( S ) t( S ) t( S ) t( T ) t( N )  N*  ] }}  {{d*}}
   -t( {{d}} ) {{ INT[    t( E )t( E ) t( S ) t( S ) t( T ) t( B''(N) )  N*  ] }}  {{d*}}
   - {{ INT[   t( F )t( S ) t( S ) t( T )  N*  ] }}  {{d*}}
   -t( {{d}} ) {{ INT[    t( S )t( S ) t( E ) t( E ) t( T ) t( N )  B''(N)*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[    t( E )t( E ) t( E ) t( E ) t( T ) t( B''(N) )  B''(N)*  ] }}  {{d*}}
   + {{ INT[   t( F )t( E ) t( E ) t( T )  B''(N)*  ] }}  {{d*}} ) = (0)
Line 2:
(t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }} - {{ INT[ t( F ) N*  ] }}
   -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}
   +t( {{d}} ) {{ INT[    t( S )t( S ) t( S ) t( S ) t( T ) t( N )  N*  ] }}
   -t( {{d}} ) {{ INT[    t( E )t( E ) t( S ) t( S ) t( T ) t( B''(N) )  N*  ] }}
   - {{ INT[   t( F )t( S ) t( S ) t( T )  N*  ] }}
   -t( {{d}} ) {{ INT[    t( S )t( S ) t( E ) t( E ) t( T ) t( N )  B''(N)*  ] }}
   +t( {{d}} ) {{ INT[    t( E )t( E ) t( E ) t( E ) t( T ) t( B''(N) )  B''(N)*  ] }}
   + {{ INT[   t( F )t( E ) t( E ) t( T )  B''(N)*  ] }} ) = (0)
Line 3:
(t( {{ INT[  t( S )t( S ) t( N )  N*  ] }} ) {{d}} -t( {{ INT[ t( F ) N*  ] }} )
   -t( {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }} ) {{d}}
   +t( {{ INT[    t( S )t( S ) t( S ) t( S ) t( T ) t( N )  N*  ] }} ) {{d}}
   -t( {{ INT[    t( E )t( E ) t( S ) t( S ) t( T ) t( B''(N) )  N*  ] }} ) {{d}}
   -t( {{ INT[   t( F )t( S ) t( S ) t( T )  N*  ] }} )
   -t( {{ INT[    t( S )t( S ) t( E ) t( E ) t( T ) t( N )  B''(N)*  ] }} ) {{d}}
   +t( {{ INT[    t( E )t( E ) t( E ) t( E ) t( T ) t( B''(N) )  B''(N)*  ] }} ) {{d}}
   +t( {{ INT[   t( F )t( E ) t( E ) t( T )  B''(N)*  ] }} )) = (0)
```

Figure 108 Galerkin Least Squares formulation for the 1-D diffusion equation



Euler-Lagrange equations term	Perturbation	Domain
SSU-EEUxx-F	TSSW-EEWxx	D

Current Euler-Lagrange equations term: SSU-EEUxx-F

Perturbation: TSSW-EEWxx

Current domain: D

Figure 109 Stabilization terms added for the Galerkin Least Squares formulation

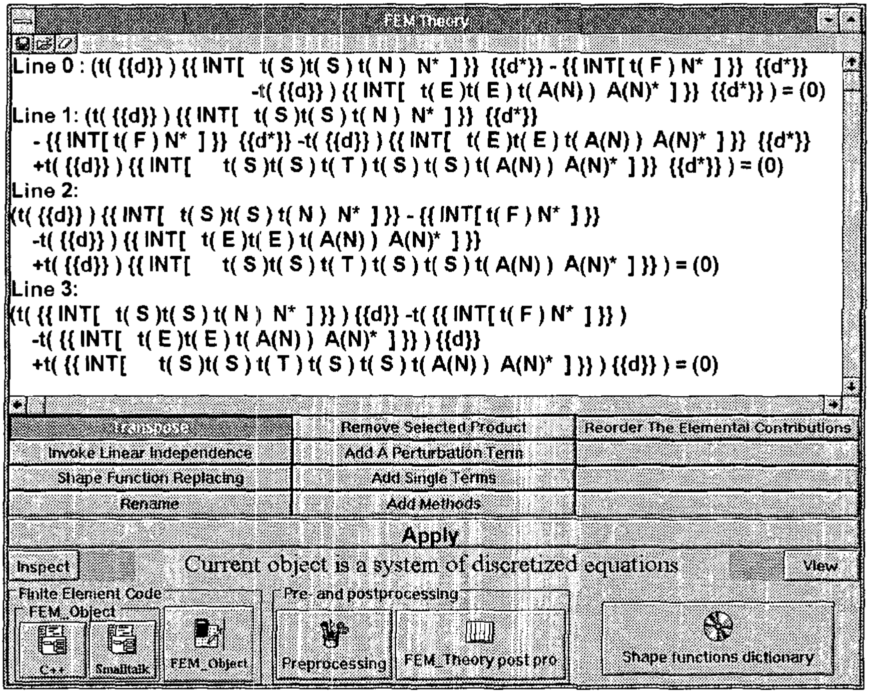
Galerkin / gradient Least Squares formulation

This formulation is presented in [FRA 89]. The starting point is also the classical Galerkin formulation (line 0 of Figure 110). In the theory, the gradient of the Lagrange equation is taken in place of the Lagrange equation (see Galerkin Least Squares method). The term added to the classical Galerkin approximation is then :

$$\sum_{\Omega^e \in \Omega_h} \left[\int_{\Omega^e} (\sigma^2 u^h - \varepsilon^2 u^h_{,xx} - f)_{,x} \tau (\sigma^2 w^h - \varepsilon^2 w^h_{,xx})_{,x} dv \right]$$

where τ is the stabilization parameter.

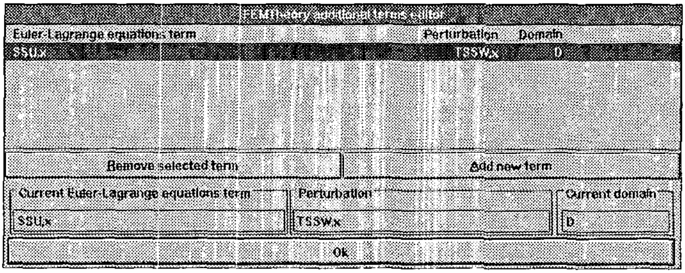
In FEM_Theory, «SSU,x-EEU,xxx-F,x» is weighted by «TSSW,x-TEEW,xxx». «T» is the stabilization parameter. As we want only to test piecewise linear and quadratic interpolations, the third derivatives with respect to the variable «x» are zero are not introduced in the formulation (see Figure 111). The body loads are not shown here and are just introduced by hand into the numerical code.



```
Line 0 : (t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }}  {{d*}} - {{ INT[ t( F ) N*  ] }}  {{d*}}
                              -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}  {{d*}} ) = (0)
Line 1: (t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }}  {{d*}}
   - {{ INT[ t( F ) N*  ] }}  {{d*}} -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}  {{d*}}
   +t( {{d}} ) {{ INT[     t( S )t( S ) t( T ) t( S ) t( S ) t( A(N) )  A(N)*  ] }}  {{d*}} ) = (0)
Line 2:
(t( {{d}} ) {{ INT[  t( S )t( S ) t( N )  N*  ] }} - {{ INT[ t( F ) N*  ] }}
   -t( {{d}} ) {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }}
   +t( {{d}} ) {{ INT[     t( S )t( S ) t( T ) t( S ) t( S ) t( A(N) )  A(N)*  ] }} ) = (0)
Line 3:
(t( {{ INT[  t( S )t( S ) t( N )  N*  ] }} ) {{d}} -t( {{ INT[ t( F ) N*  ] }} )
   -t( {{ INT[  t( E )t( E ) t( A(N) )  A(N)*  ] }} ) {{d}}
   +t( {{ INT[     t( S )t( S ) t( T ) t( S ) t( S ) t( A(N) )  A(N)*  ] }} ) {{d}} ) = (0)
```

Current object is a system of discretized equations

Figure 110 Gradient/Galerkin-Least Squares formulation for the 1-D diffusion problem



Euler-Lagrange equations term	Perturbation	Domain
SSU.x	TSSW.x	D

Current Euler-Lagrange equations term: SSU.x

Perturbation: TSSW.x

Current domain: D

Figure 111 Stabilization terms added for the Gradient/Galerkin-Least Squares formulation

Numerical results for the one-dimensional diffusion equation

The numerical test is the same as the one presented in [FRA 89]. The domain is [0,1]. The boundary conditions are $u(0)=0$ and $u(1)=2$. The body loads are linear on the domain, i.e. $f=X$. The parameters of the equation are : $\varepsilon=10^{-8}$ and $\sigma=1$. One can notice that the ratio ε^2/σ^2 is severe. For the piecewise quadratic approximation, various values of τ, the stabilization parameter, are tested. The numerical results are shown in Figure 112.

The important point for stabilized formulations is the choice of the stabilization parameter. Various methods of defining stabilization parameters exist for the Galerkin Least Squares method. But here the choice of the parameter has no stabilizing effect on the formulation (see piecewise linear interpolation in Figure 112). For the Galerkin / gradient Least Squares method and quadratic approximation, fluctuations around the value advocated in [FRA 89] were studied to see the influence of the parameter of stabilization. The optimal value was suggested in [FRA89]; a uniform mesh is used here, and $\tau=h^2/6\sigma^2=3.4e-3$ (see [FRA 89]). We see that for small values ($\tau\leq 1e-4$) the stabilizing effect is not important enough. But in this example, the value $\tau=5e-4$, which is in the "vicinity" of the optimal value $\tau=3.4e-3$, gives acceptable results. The value $\tau=1e-3$ is already too big, and the solution shows too much diffusion.

In this example, various formulations were evaluated on a simple scalar equation model.

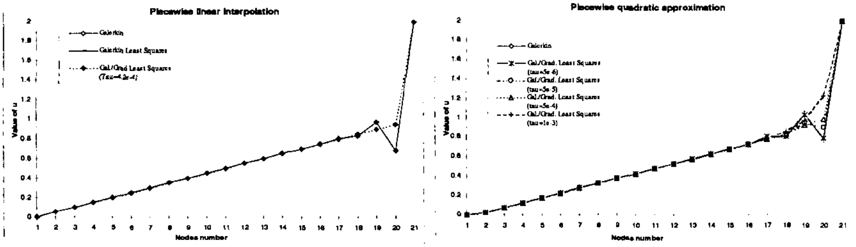


Figure 112 Numerical results for the 1-D diffusion equation

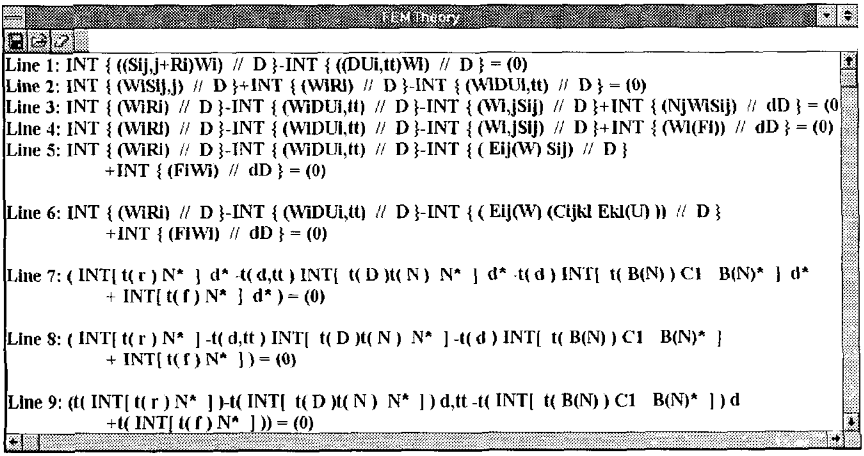
Appendix C - Linear elastodynamics

The mathematical formulation of this problem and the classical formulation of this problem are presented at length in chapter 2.

Derivation of the formulation and automatic programming

In this section the different steps of the derivation are presented. The successive steps can be seen on the screen shown in Figure 113. Each line is briefly discussed.

Line 1 : The variational formulation, instance of **IntEquation**, is created from instances of **String**. «Sij» represents the strain tensor, «Ri» the body load components, «Wi» the virtual displacement field components, «D» the density and «Ui» the solution field. This instanciation is made through a window which permits the introduction of a user defined set of differential equations, with access to a predefined dictionary of problems.
Line 2 : The button *'Expand'* was pushed. The integrands of each integral are developed (instance of Expression) and the instances of Integral apply the linearity property. The result is shown on this line.



```
Line 1: INT { ((Sij,j+Ri)Wi)  //  D }-INT { ((DUi,tt)Wi)  //  D } = (0)
Line 2: INT { (WiSij,j)  //  D }+INT { (WiRi)  //  D }-INT { (WiDUi,tt)  //  D } = (0)
Line 3: INT { (WiRi)  //  D }-INT { (WiDUi,tt)  //  D }-INT { (Wi,jSij)  //  D }+INT { (NjWiSij)  //  dD } = (0
Line 4: INT { (WiRi)  //  D }-INT { (WiDUi,tt)  //  D }-INT { (Wi,jSij)  //  D }+INT { (Wi(Fi))  //  dD } = (0)
Line 5: INT { (WiRi)  //  D }-INT { (WiDUi,tt)  //  D }-INT { ( Eij(W) Sij)  //  D }
         +INT { (FiWi)  //  dD } = (0)

Line 6: INT { (WiRi)  //  D }-INT { (WiDUi,tt)  //  D }-INT { ( Eij(W) (Cijkl Ekl(U) ))  //  D }
         +INT { (FiWi)  //  dD } = (0)

Line 7: ( INT[ t( r ) N*  ]  d* -t( d,tt ) INT[  t( D )t( N )  N*  ]  d* -t( d ) INT[  t( B(N) ) C1   B(N)*  ]  d*
         + INT[ t( f ) N*  ]  d* ) = (0)

Line 8: ( INT[ t( r ) N*  ] -t( d,tt ) INT[  t( D )t( N )  N*  ] -t( d ) INT[  t( B(N) ) C1   B(N)*  ]
         + INT[ t( f ) N*  ] ) = (0)

Line 9: (t( INT[ t( r ) N*  ] )-t( INT[  t( D )t( N )  N*  ] ) d,tt -t( INT[  t( B(N) ) C1   B(N)*  ] ) d
         +t( INT[ t( f ) N*  ] )) = (0)
```

Figure 113 Derivation of the elastodynamics problem in FEMTheory

Line 3 : On *Line 2*, the integral «INT{(WiSij,j) // D}» has been selected and the button 'Integrate by parts' pushed. The selected integral is replaced by two instances of Integral with appropriate integrands built from the initial one.
Line 4 : On *Line 3*, the integral «INT{(NjWiSij) // D}» has been selected and the button '*Substitute*' pushed. A prompter permits to replace instances of terms of the integrand, here «NjSij», by an expression, here «(Fi)» (the natural boundary condition).
Line 5 : The same operation is carried out on the integral «INT{(Wi,jSij) // D}». As the tensor «Sij» is symmetric, the following equality is verified : «SijWi,j=SijEij(W)» where «Eij» represents the strain tensor. So, the term represented by «Wi,j» is replaced by «Eij(W)».
Line 6 : In the integral «INT{(Eij(W)Sij) // D}», the term «Sij» is replaced, using the constitutive law, by «CijklEkl(U)». It is the same operation as described above.
Line 7 : The object shown on *Line 6* represents a weak form of the problem. This form can be the basis for the Galerkin approximation. So the button '*Discretize*' has been pushed ; this implies replacement of the tensor notation by the vector notation, Galerkin approximation, discretization of the domain, approximation of the different fields on an element. This scheme is actually the only one implemented, but alternative schemes could easily be implemented. Note that the notation employed to name an instance of **DiscretizationMatrix** shows how it has been built ; for example the string «INT[t(B(N))C1B(N)*]» represents the stiffness matrix, the well known $\int_{\Omega} B^{t}DBdv$ (see [HUG 87] for details). It is shown that the operator B (coming from the discretization of «Eij») is applied to shape function matrix N.
Line 8 : The equation of *Line 7* is verified for every d^{*}, so the coefficient of this term must be zero. This operation is done with the button '*Invoke linear independence*'. Note that the result of this operation is an instance of **System** and that, for a mixed formulation, more than one equation would be obtained (see the example of stokes flow in Chapter 5).
Line 9 : To obtain the final form, the equation is transposed (button '*Transpose*'). Then the shape functions are replaced by their expression, and the code corresponding to the new formulation is created in FEMObject, in Smalltalk.

Test of the element

For completeness, a test is performed to check the newly created element.

Description of the problem

The numerical problem which is proposed is the analysis of an impact of a rectangular block on a rigid surface. The problem is described in, the mesh and the boundary conditions are shown in Figure 115.

The data are :

density	$D = 0.01\,\text{kg.m}^{-3}$
Young modulus	$E = 1000\,\text{N.m}^{-2}$
Poisson's coefficient	$\nu = 0.03$

The parameters for the explicit predictor-corrector algorithm are :

$$\gamma = 0.5 \qquad \beta = 0.25$$

The time step of the integration scheme is : $\Delta t = 2.10^{-4}$ s.
The initial velocity of the body is : $v_0 = 1\ m.s^{-1}$

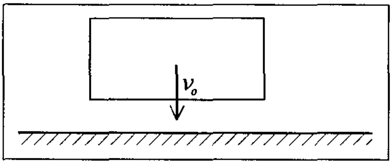


Figure 114 Description of the impact problem

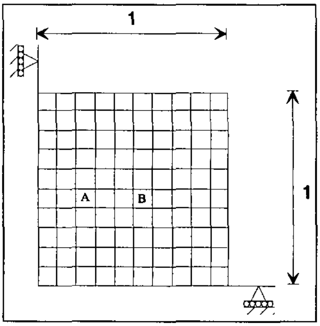


Figure 115 Finite element mesh for impact problem

Numerical results

The exact solution, shown in Figure 116 at $t = 7\,\Delta t$, is characterized by a dilatational wave front emanating from the impact. The circular wave front results from the reflection of the boundary condition. The center of the circle is the right bottom corner, the wave velocity is given by :

$$c = \left(\frac{E(1-\nu)}{(1+\nu)(1-2\nu)\rho} \right)^{\frac{1}{2}} = 366.9\ m.s^{-1}.$$

This value makes it possible to determine the time step. The deformed mesh is presented at $t = 7\,\Delta t$ in Figure 117 and agreement with theoretical solution is obtained. In Figure 118 the stress time-history of elements A and B are compared (see the finite element mesh in Figure 115) with the ones obtained by Hughes and al. in [HUG 76]. This simulation has permitted to obtain in a few minutes the same results as the ones obtained in [HUG 76] and an evaluation of this formulation on alternative numerical problems could now be performed.

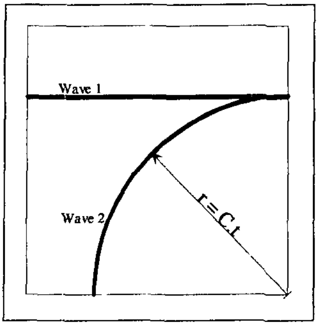


Figure 116 Exact solution for impact problem

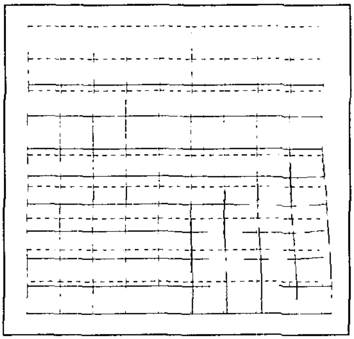


Figure 117 Deformed mesh for impact problem

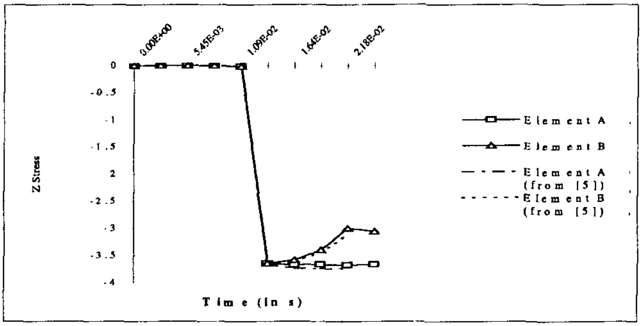


Figure 118 Stress time-history for impact problem

Curriculum vitae

Dominique EYHERAMENDY
French citizen
Male
Born in 1968
Single

EDUCATION

Sep. 89 - Jun. 92 **Scholarship at the Ecole Normale Supérieure de Cachan :**
Department of Mechanics and Technology
Ecole Normale Supérieure de Cachan, France
Sep. 89 - Jun. 90 : **License ès Sciences (Univ. Paris VI)**
Sep. 90 - Jun. 91 : **Master of Science (Univ. Paris VI)**
Sep. 91 - Jun. 92 : **«Agrégation» of Mechanics**

Sep. 86 - Jun. 89 **Scholarship at the Lycée Gustave Eiffel :**
Classes de Mathématiques Supérieures et Spéciales de Technologie
Bordeaux, France

Jun. 86 **Certificate of Secondary education**
Scientific (Mathematics and physics)
Lycée Jules Supervielle, Oloron, France

EXPERIENCE

Sep. 92 – Dec. 97 **Research Assistant**
Swiss Federal Institute of Technology - Laboratory of Structural and Continuum Mechanics
Research work:
-1- Extension of concepts developed at the LSC in the Object-Oriented formulation applied to the Finite Element Method. Developments towards O.O automatic coding.
-2- Research and developments for Debris flows simulation.
Teaching work: Lectures on "Structural Mechanics" and "Solid Mechanics".

Sep. 95 – Sep. 97 **Teaching Assistant**
Ecole supérieure d'Ingénieurs d'Annecy, Annecy, France
Teaching work: Lectures on "Structural Mechanics" and "Technology".

Sep. 95 – Mar. 96 **Visit at the University of Minnesota and at the Minnesota Supercomputer Institute, Minneapolis, USA**
Research work: Study and development of an object-oriented computer code for the simulation of free-surface flows (in relation with Prof. T.E. Tezduyar).

PUBLICATIONS

International journals :

D. Eyheramendy and Th. Zimmermann *Fonctionnalité d'un environnement orienté objet pour le développement de code éléments finis*, Submitted to Revue Européenne des éléments finis (special issue), (1997).

D. Eyheramendy and Th. Zimmermann, *Intégration d'une approche variationnelle pour la méthode des éléments finis dans un environnement orienté objet : Application à un problème de convection non-linéaire*, Submitted to Revue Européenne des éléments finis (special issue), (1997).

D. Eyheramendy and Th. Zimmermann, *Object-oriented finite elements : III. Theory and application of automatic programming*, To appear in Comput. Methods Appl. Mech. Engrg., (1997).

D. Eyheramendy and Th. Zimmermann, *Object-oriented finite elements : II. A symbolic environment for automatic programming*, Comput. Methods Appl. Mech. Engrg., 132 (1996) pp. 277-304.
Th. Zimmermann and D. Eyheramendy, *Object-orientedfinite elements : I. Principles of symbolic derivation and automatic programming*, Comput. Methods Appl. Mech. Engrg., 132 (1996) pp. 277-304.
D. Eyheramendy and Th. Zimmermann, *Object-oriented Finite Element Programming : An interactive environment for symbolic derivations, Application to an Initial Boundary Value Problem*, Advances in Engineering Software 27 (1996) 3-10.
D. Eyheramendy and Th. Zimmermann, *Programmation orientée objet appliquée à la méthode des éléments finis : dérivations symboliques, programmation automatique*, Revue Européenne des éléments finis, vol. 4 (1995) pp. 327-360.

Internal reports :
D. Eyheramendy and Th. Zimmermann, *Object-Oriented finite element programming Development of an environment for symbolic derivations and automatic programming*, Rapport interne LSC-EPFL 94/5, April 1994.
D. Eyheramendy and Th. Zimmermann, *Développement d'un concept orienté objet pour un préprocesseur d'éléments finis*, Rapport de Maîtrise ENS de Cachan et Rapport interne LSC-EPFL 91/17, Août 1991.

Conferences :
** D. Eyheramendy and Th. Zimmermann, *Symbolic derivations and automatic generation of finite elements in an object-oriented environment*, Proceedings of the 4th US National Congress on Computational Mechanics, San Francisco, (1997).
R. Frenette, D. Eyheramendy and Th. Zimmermann, *Numerical modeling of dam-break type problems for Navier-Stokes and granular flows*, 1st international Conference on Debris-flow hazards mitigation : mechanics, prediction, and assessment, San Francisco, Aug. 7-9 1997, Ed. C.L. Chen, (1997) pp. 586-595.
** D. Eyheramendy and Th. Zimmermann *Dérivations symboliques pour code éléments finis - Application à un problème d'élasticité*, Actes du 3ième Colloque national en calcul des structures de Giens, vol. 2 (1997) pp. 553-558.
** D. Eyheramendy and Th. Zimmermann *Fonctionnalité d'un environnement orienté objet pour le développement de code éléments finis*, Actes du 3ième Colloque national en calcul des structures de Giens, vol. 2 (1997) pp. 837-842.
** Th. Zimmermann, D. Eyheramendy, P. Bomme, S. Commend and R.S. Arruda, Object-oriented finite element programming : Languages, Symbolic derivations, Reasoning capabilities, Proceedings of NAFEM 97- Stuttgart, vol. 1 (1997) pp. 652-663.
Th. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier and S. Commend, *Object-Oriented Finite Element Techniques : Towards a general purpose environment*, Proceedings of CST 95 Cambridge (1995).
Th. Zimmermann and D. Eyheramendy, *Symbolic object-oriented Finite Element Programming - Application to incompressible viscous flow*, Proceedings of IECS 95 Hawaii, vol. 1 (1995) pp. 21-26.
** D. Eyheramendy and Th. Zimmermann, *Génération automatique de Code Eléments Finis dans un environnement Orienté Objet*, Actes du 2nd Colloque national en calcul des structures de Giens (1995) pp. 717-722.
Th. Zimmermann and D. Eyheramendy, *Object-oriented finite element programming : Automatic programming*, Proceedings of the Third Congress on Computational Mechanics (IACM), Chiba Japan, vol. II (1994) pp. 1527-1528.
** D. Eyheramendy and Th. Zimmermann, *Object-oriented finite element programming : Beyond fast prototyping*, Proceedings of CST 94 , Athens Greece, vol. Artificial intelligence and object oriented approaches for structural engineering, Civil Comp Press, (1994) pp. 121-127.

*N.B. : the presentations are highlighted using the symbol***