Titre: Title:	Développement d'un logiciel de calcul par éléments finis fondé sur les formes différentielles
Auteur: Author:	Ludovic Pénet
Date:	1999
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pénet, L. (1999). Développement d'un logiciel de calcul par éléments finis fondé sur les formes différentielles [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8710/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8710/
Directeurs de recherche: Advisors:	François Trochu
Programme: Program:	Unspecified

UNIVERSITÉ DE MONTRÉAL

DÉVELOPPEMENT D'UN LOGICIEL DE CALCUL PAR ÉLÉMENTS FINIS FONDÉ SUR LES FORMES DIFFÉRENTIELLES

LUDOVIC PÉNET
DÉPARTEMENT DE GÉNIE MÉCANIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE MÉCANIQUE)
DÉCEMBRE 1999

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DÉVELOPPEMENT D'UN LOGICIEL DE CALCUL PAR ÉLÉMENTS FINIS FONDÉ SUR LES FORMES DIFFÉRENTIELLES

présenté par : PÉNET Ludovic
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment lu et accepté par le jury d'examen constitué de :

M. GAUVIN Raymond, D.Sc.A., président
M. TROCHU François, Ph.D., membre et directeur de recherche
M. BELIVEAU Alain, Ph.D., membre

REMERCIEMENTS

Je souhaite remercier les membres du jury :

Monsieur Raymond Gauvin, professeur au département de génie mécanique de l'École Polytechnique de Montréal pour avoir accepté de présider ce jury.

Monsieur François Trochu, professeur au département de génie mécanique de l'École Polytechnique de Montréal, pour m'avoir accueilli dans son laboratoire et m'avoir laissé une grande liberté dans mon travail.

Monsieur Alain Béliveau, Ph.D. en mathématiques appliquées, pour ses précieux conseils et pour avoir accepté de faire partie de ce jury.

Un remerciement particulier pour Monsieur Jean-François Remacle, assistant de recherche au Rensselaer Polytechnic Institute, pour m'avoir encadré durant la majeure partie de cette thèse.

Je remercie les autres étudiants du laboratoire : Tan Pham, Élaine Bohr, Éric Béchet.

Je remercie les nombreux relecteurs de ce document pour leurs judicieuses remarques tant sur le fond que sur la forme : Guillaume Lapierre et Régis Vinciguerra.

Je salue également les autres personnes que j'ai rencontrées ici et qui ont contribué à rendre mon séjour au Québec agréable, notamment Chloé Galipeau.

Merci enfin à Éric Cartman pour m'avoir soutenu moralement tout au long de cette maîtrise.

RÉSUMÉ

Ce travail traite de la conception d'un moteur de calcul par éléments finis généraliste. La méthode des éléments finis est une méthode numérique de résolution des équations aux dérivées partielles utilisée aujourd'hui dans la plupart des génies. La conception de moteurs de calcul par éléments finis est un problème ancien, ayant connu différents avatars selon les paradigmes en vogue de l'époque.

Dans un premier temps, nous étudions le code élément fini du laboratoire, LCMFlot, et passons en revue différentes publications. Nous procédons également à une étude mathématique de la méthode des éléments finis, en l'abordant du point de vue de la géométrie différentielle. Cette partie de l'étude permet de définir des outils originaux basés sur la géométrie différentielle et justifie que l'on désigne cette dernière comme la clé mathématique de cette conception.

L'originalité de ce travail réside également dans l'utilisation d'une méthode d'analyse encore peu utilisée, l'approche multi-paradigmes, et la justification de l'utilisation de la programmation dite générique. Cette dernière sépare systématiquement algorithmes et données, mais d'une manière différente de celle de la programmation procédurale. La « STL » d'Alexander Stepanov est un exemple d'utilisation de cette programmation.

Le résultat est un logiciel d'éléments finis rapide, compact, facile à utiliser et à étendre. Le développement d'une formulation Navier-Stokes prend moins d'une journée. Il est de plus capable, grâce aux diverses généralités, tant mathématiques qu'informatiques, introduites, de traiter, à partir d'une unique formulation variationelle, un problème en une dimension quelconque, ce pour des problèmes simples, comme la résolution d'un Laplacien, pour des problèmes complexes, comme la résolution de l'équation de

transport par la méthode de Lesaint-Raviart en passant par des classiques, comme Navier-Stokes.

ABSTRACT

This work is about the conception of a general finite element computation engine. The finite element method is a partial derivatives équations numerical resolution technique. It is used nowadays in most scientific fields. Conception of finite element computation engine is an old problem. Different solutions, conceived with various programming parading were proposed.

Whe begin with the study of LCMFlot, the finite element code of the laboratory, and some related publications. We follow by the study of the finite element method mathematics, from the differential geometry point of view. This part of the work makes the definition of original tools possible. Differential geometry is thus the mathematical key of this conception.

Multi-paradigm design is another innovation of this work, and justificatifies the use of generic programming as our implementation of the finite element method. In a manner quite different from the old procedural approach, generic programmaing systematically splits algorithms and data. Alexander Stepanov's STL is an example of package implemented using this programming style.

The result is a fast, compact, easy to use and effortlessly extendable finite element engine. Using this finite element package, developpement of the Navier-Stokes equations takes less than one day. It is also capable, due to the introduction of mathematical and design generalisation, of solving a problem using a single, dimension free, variational formulation. This, for simple examples, such as the Laplace equation, as for more complex problem, like the resolution of the transport equation using the Lesaint-Raviar scheme.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

ACRONYMES

COO : Conception Orientée Objet

UML : Unified Modeling Language ou langage unifié pour la modélisation objet. Il permet de décrire de façon standard un système informatique à base d'objets. Ce n'est pas une méthodologie objet ainsi UML est souvent associé à des méthodes classiques telles qu'OMT.

LCM : Liquid Composit Molding. Procédé de fabrication de pièces composites par injections dans un renfort fibreux.

EDP : Équation aux dérivées partielles.

MEF : Méthode des éléments finis. Méthode de résolution d'équations aux dérivées partielles.

STL Standard Template Library. Librairie d'algorithmes et de structures de données intégrée au standard C++.

DDL : Degré De Liberté

INTRODUCTION

La méthode des éléments finis (MEF) est une méthode numérique de résolution d'équations aux dérivées partielles (EDP). Elle a bénéficié depuis environ un demi-siècle de développements constants, essentiellement théoriques et mathématiques. Dans les années 70, accompagnant les progrès fulgurants de l'informatique, tant au niveau de la puissance de calcul disponible que dans les techniques de conception et de développement de logiciel, de nombreux canevas, le plus souvent commerciaux, comme par exemple NASTRAN, de résolution d'EDP par la MEF ont été développés. Le paradoxe est que, bien que tant sur les plans mathématiques et informatiques les choses aient beaucoup changées, les codes phares restent basés sur des technologies datant d'au moins 20 ans, comme la programmation procédurale par exemple. Aussi, certaines personnes ont décidé de développer d'autres canevas intégrant les nouveautés en matière de philosophie de conception et de programmation. Le langage C++ est généralement retenu, du fait de la généralisation de son utilisation. Or, les designs résultant de ces approches volontaristes sont, comme on le verra, souvent, et paradoxalement, limités par les dogmes de l'orienté objet. Aussi, après avoir étudié le code élément fini du laboratoire, LCMFlot, ainsi que diverses publications traitant de ce sujet, nous attellerons-nous à l'utilisation de paradigmes autres que l'orienté objet ainsi que de certains outils mathématiques rendant possible l'utilisation de ces paradigmes, comme les formes différentielles et un système de gestion des degrés de liberté original, pour concevoir et implémenter un moteur de calcul d'une nouvelle espèce, d'une généralité et d'une extensibilité à notre connaissance inégalées : FEMView.

Chapitre 1 INTRODUCTION À LA MODÉLISATION ORIENTÉE OBJET ET AU C++

1.1 Une brève présentation de la conception orientée objet

La conception orientée objet (COO) avec son pendant logique la programmation orientée objet est l'approche généralement retenue depuis environ une décennie pour concevoir des projets complexes, de grande envergure. On présente ici une version délibérément simplifiée de cette approche, tout en respectant l'essence et les principes.

1.1.1 Le modèle statique

1.1.1.1 Classes, spécialisation et héritage

Le modèle statique cherche uniquement à représenter les structures de données (les **classes**). Prenons l'exemple « simple » d'un être humain. On peut choisir de le définir selon un ensemble limité de caractéristiques : couleur des yeux, des cheveux, de la peau, taille et poids. Le choix des caractéristiques retenues dépend du domaine à modéliser. On peut vouloir ensuite définir des sous-ensembles d'êtres humains, ayant certaines caractéristiques supplémentaires selon, par exemple, leur occupation. Si l'on modélise une université, il peut ainsi être intéressant de distinguer les professeurs des élèves. Le

fait de définir un sous-ensemble d'entités ayant des caractéristiques supplémentaires constitue dans la terminologie orientée objet une **spécialisation**. Elle se traduit en C++ par un **héritage**. Dans le diagramme suivant, le triangle indique qu'Élève et Professeur sont des spécialisations de la classe de base.

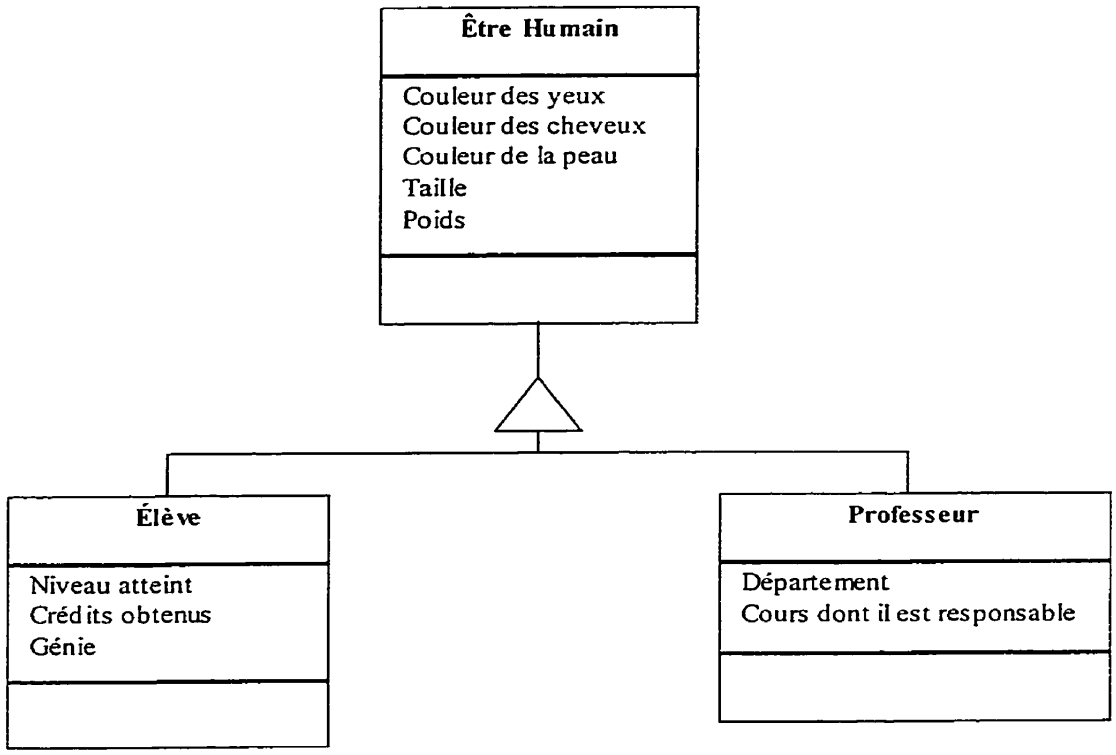


Figure 1.1- Exemple d'héritage (de spécialisation)

1.1.1.2 Composition

Certaines relations ne sont pas des spécialisations mais plutôt des **compositions**. Un exemple simple est une maison composée de portes, fenêtres, etc. On ne peut pas définir une maison comme un type particulier de porte ou de fenêtre particulier. Ceci implique

que la relation ne peut pas être modélisée par une spécialisation. La relation entre la maison, les portes et les fenêtre est dite de **composition**. Les compositions sont notées comme suit, des variables apparaissant selon la cardinalité :

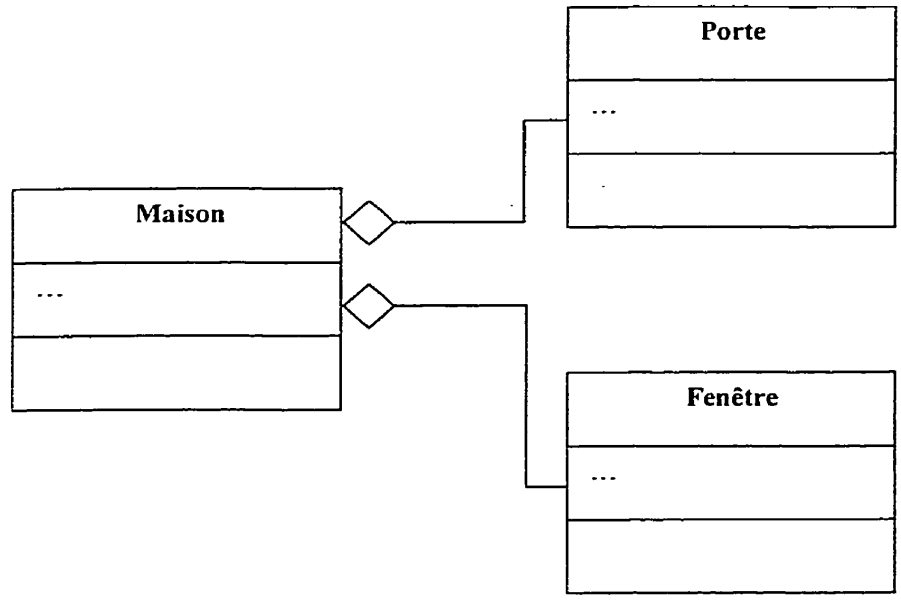


Figure 1.2 - Exemple de composition

La cardinalité est l'expression du nombre de quantité de chaque côtés de la relation. On pourrait ainsi préciser qu'une fenêtre ne peut appartenir qu'à une seule maison ou encore qu'une maison peut comporter N fenêtres.

Il est bien évident qu'il peut exister des types particuliers de maison, porte et fenêtre; des spécialisations de ces concepts généraux. On indique alors une composition entre classes de base, signifiant ainsi que, par exemple, une maison est constituée de portes dont le type peut être le type de base ou l'une de ses spécialisations.

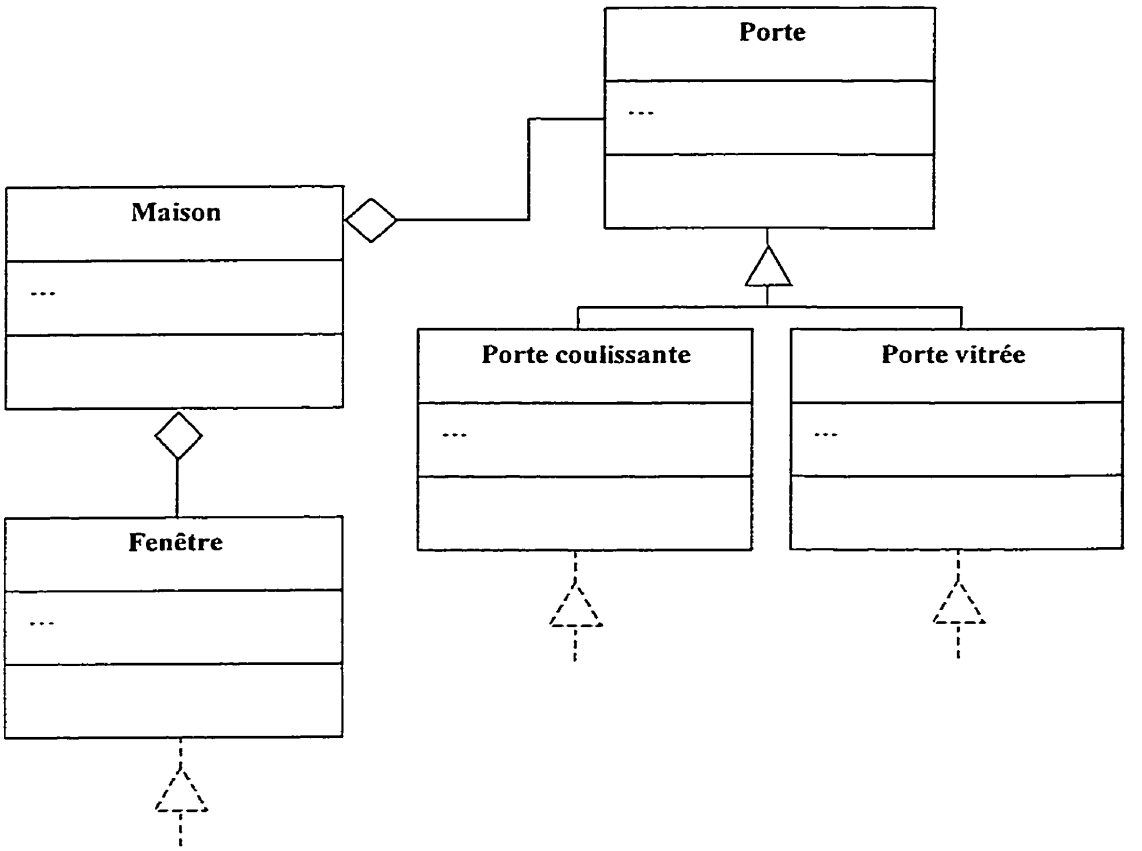


Figure 1.3 - Composition entre hiérarchies de classes

1.1.1.3 Association

Le dernier concept utilisé lors de la modélisation statique est celui d'**association**. Une association est définie formellement comme une **relation de dépendance entre classes**. Revenant sur notre premier exemple où intervenaient des élèves et des professeurs, on peut imaginer une association très simple, comme *enseigne à*. La notation des associations reflète la diversité de cardinalités pouvant exister dans une association. Elle se ramène cependant le plus souvent à une simple ligne entre deux classes. Les extrémités de la ligne pouvant être pourvues de petites boules pour indiquer des

subtilités de cardinalité, dont nous ne nous préoccupons pas ici. Des associations peuvent exister entre plus de deux classes, elles peuvent également être qualifiées, c'est à dire que l'on définit une classe contenant les attributs et les comportements de cette association. Le cas ne s'étant pas présenté lors de notre étude, on évite délibérément d'approfondir ces deux aspects.

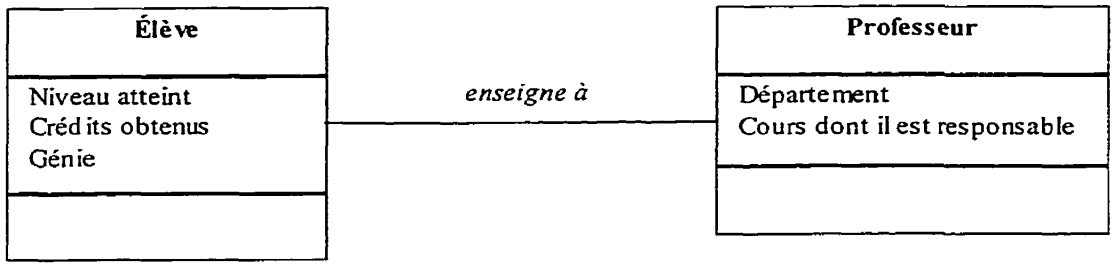


Figure 1.4 - Exemple d'association

1.1.2 Le modèle dynamique

L'objectif du **modèle dynamique** est de **décrire les aspects temporels et les comportements** du système et des objets. Il utilise pour cela des **scénarios**. Ils peuvent être très simples et ne pas nécessiter de commentaires particuliers. Ils peuvent également être particulièrement complexes, comme un algorithme d'emploi du temps par exemple. Les scenarios sont composés d'**événements**. Les évènements peuvent être définis comme étant les stimuli externes (à une classe ou à un système). Chaque classe fait l'objet d'un **diagramme d'état**, décrivant précisément les changements d'état d'un objet selon les stimuli externes. Il est également courant d'utiliser un **diagramme d'événements**, représentant le déroulement temporel des événements.

Lors de sa modélisation dynamique, on décrit le comportement des instances d'une classe soumise à certains événements. Ces événements peuvent provoquer des **transitions vers d'autres états** ainsi que des **actions** et des **activités**. Une activité est distinguée d'une action en ce qu'elle dure alors qu'une action tend à être ponctuelle. On peut voir ci-dessous le diagramme d'état d'une hypothétique classe modélisant un lecteur de disques compacts.

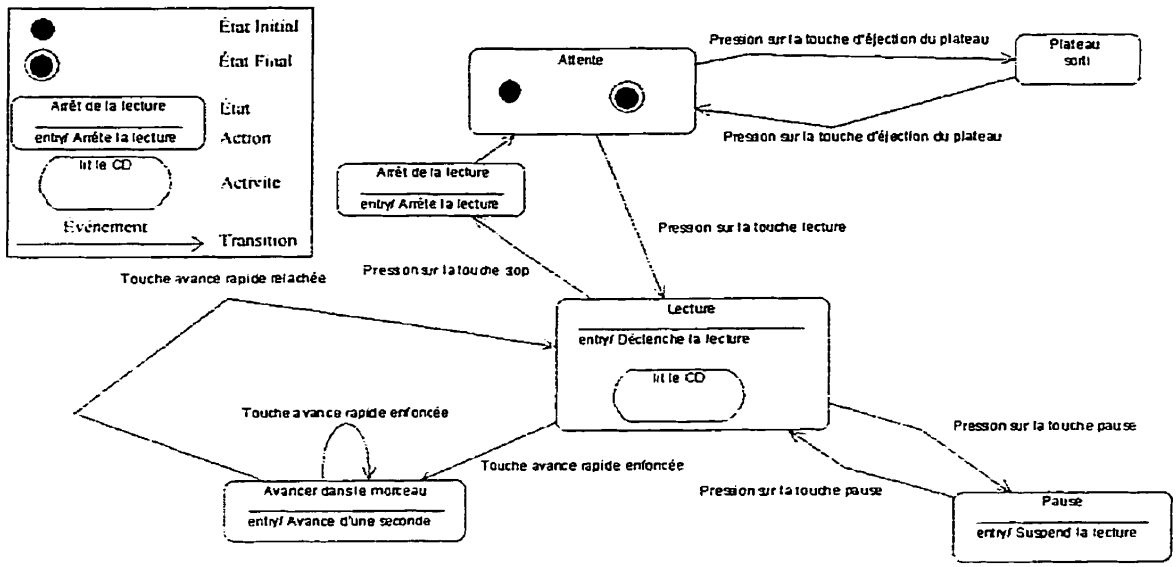


Figure 1.5 - Exemple simplifié de diagramme d'état: le lecteur de disques compacts

Le diagramme ci-dessous représente la séquence d'événements se produisant lors de la recherche d'un document à l'aide du protocole de recherche bibliographique Z39.50.

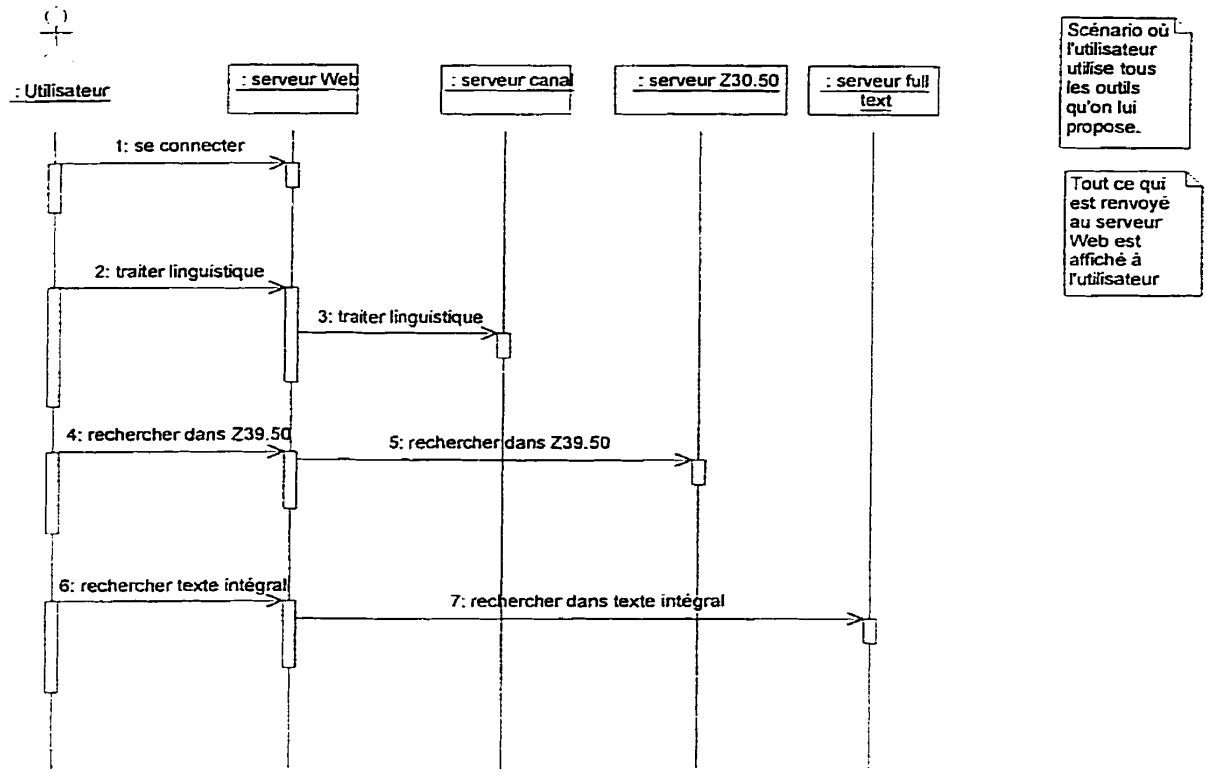


Figure 1.6 - Exemple simplifié de diagramme d'événements

Le modèle dynamique étant surtout utile dans le cas de systèmes interactifs et un moteur de calcul par éléments finis étant par définition assez peu interactif, nous ne nous attardons pas plus sur cet aspect de la conception orientée objet.

1.1.3 Le modèle fonctionnel

Le modèle fonctionnel décrit le **traitement des données** sans tenir compte de l'ordonnancement des dits traitements, ni de la structure des objets. Il met en évidence les **dépendances** et les **relations entre les valeurs**. On représente graphiquement le traitement des données par le biais d'un **diagramme de flots de données**. Sur ce diagramme on note les algorithmes avec leurs données d'entrées et leurs données de

sortie. Les algorithmes simples peuvent être directement spécifiés sur le schéma. Dès que l'algorithme devient complexe, on préfère utiliser un document annexe définissant dans le détail l'algorithme. Le modèle fonctionnel a été supprimé lors de la définition d'UML (« *Unified Modeling Language* »), la norme pour ce qui est de la *notation* (et non pas la *méthode*) à employer dans une approche orientée objet. C'est bien regrettable lorsque l'on modélise un logiciel essentiellement algorithmique, comme un moteur de calcul par éléments finis.

1.1.4 Ajouter les opérations

Une fois toutes ces modélisations effectuées, il convient d'ajouter les **opérations**, également nommées **méthodes** ou **fonctions membres**, aux classes. Cela se fait principalement par l'analyse des accès nécessaires en lecture/écriture aux attributs et par l'ajout des comportements apparus dans le modèle dynamique. Il est à noter que des attributs peuvent également apparaître, et donc venir compléter le modèle statique, lors de l'analyse dynamique et de l'analyse fonctionnelle.

1.2 Présentation du langage C++

On présente dans cette partie le langage C++. Afin de complémenter la partie précédente et de faciliter la compréhension du lecteur, on redéveloppe succinctement la méthodologie orientée objet pour les besoins de la présentation de la syntaxe C++ de quelques uns de ses concepts clés.

1.2.1 Introduction

L'un des problèmes majeurs auxquels le développeur et l'architecte logiciel sont confrontés dans le développement d'un projet de grande envergure en utilisant un langage procédural comme le langage C est la séparation des algorithmes et des

données. En effet, l'absence de relation imposée entre les données et les différentes opérations à effectuer sur ces données conduit dans l'immense majorité des cas à la non-utilisation des fonctions existantes, à l'existence de plusieurs versions du même algorithme réalisées par différents programmeurs, ignorant souvent que le travail a déjà été fait. On a donc une perte de cohérence de la structure et une perte de temps, puisque le travail est effectué inutilement plusieurs fois. De plus, cette situation pose de gros problèmes sur les plans de la maintenance (modifications visant à corriger des bogues ou à effectuer des changements mineurs) et l'évolution (ajout de nouvelles fonctionnalités visant à répondre à un nouveau besoin).

Un exemple de maintenance est la correction de bogues liées à l'an 2000 ; un exemple d'évolution est l'ajout du traitement de la monnaie unique européenne, l'euro, qu'ont dû effectuer les banques et les entreprises européennes.

Ce sont principalement ces considérations qui ont amené au développement de nouvelles méthodes de conception et d'implémentation afin de raccourcir ces deux processus et donc d'en diminuer les coûts, et de faciliter en même temps l'entretien et l'évolution du logiciel. Il est intéressant de noter que ces développements ultérieurs à la conception et à l'implémentation initiale représentent 80% du coût actuel d'un logiciel et qu'il est donc primordial d'en maîtriser les coûts.

Après avoir réintroduit l'héritage et expliqué le concept d'encapsulation, nous présentons dans cette section le langage C++, sa syntaxe et comment traduire les concepts présentés dans ce document en ce langage.

1.2.2 Concepts

1.2.2.1 Classes et objets

La notion de classe en C++ représente une évolution de la notion de structure. Comme pour les structures, on définit des **classes qui sont l'abstraction d'un objet de la vie réelle ou d'un concept en termes de caractéristiques (attributs) et de comportements (fonctions membres). C'est uniquement lorsque l'on va prendre une instance de cette classe (un objet) que l'on va concrétiser les valeurs de ces attributs.**

En C++ tout (ou presque) est classe. Ainsi, les types de base, comme le type de nombre flottant double précision, sont pour le compilateur des classes comme les autres, sauf que leurs définitions sont figées, «internes» au compilateur. Une variable de type `double` est une instance de la classe `double`.

De manière analogue aux structures, on peut définir une classe `Eleve` en considérant qu'un élève est composé d'un nom, d'un prénom et d'un matricule. C'est uniquement lors de la concrétisation de cette classe, lors de l'instanciation d'une variable de type `Eleve` que l'on va garnir les attributs.

Lorsque l'on représentera les classes, on adoptera toujours la représentation suivante :

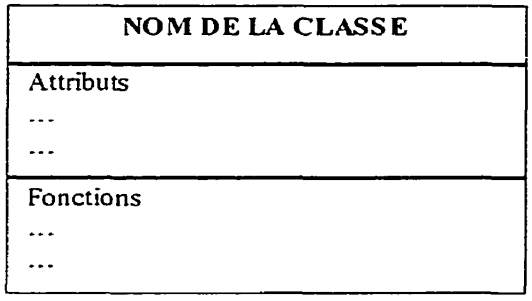


Figure 1.7 – Mode de représentation d'une classe

La classe élève se représente de la manière suivante ici:

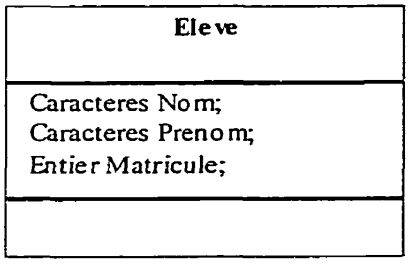


Figure 1.8 – Exemple de classe

Une instance d'une classe est représentée de la manière suivante :

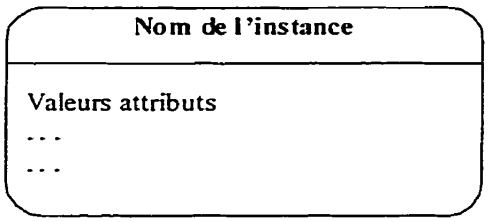
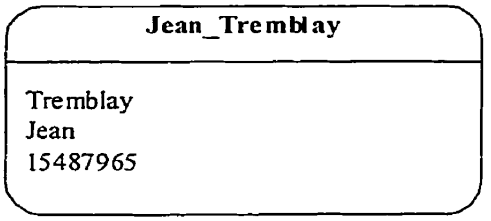


Figure 1.9 – Exemple d'instance de classe

Ainsi l'objet Jean_Tremblay, instance de la classe élève est représenté comme suit:



1.2.2.2 Héritage

Comme on l'a vu précédemment, l'héritage est un mécanisme permettant de spécialiser une classe, en lui ajoutant au besoin des attributs et fonctions. Prenons par exemple des véhicules roulants, volants et flottants. Tous ont des caractéristiques communes comme le poids, un nombre de passager ou un numéro de série. Les véhicules roulants ont en plus un nombre de roues, les véhicules volants ont en plus une envergure et les véhicules flottants on en plus un type de propulsion (à voile ou à moteur). Sans le mécanisme de l'héritage, on est obligé de faire trois classes différentes en dupliquant les attributs communs inutilement.

Avec l'héritage, on définit une classe, dite **classe de base**, de véhicule ayant les attributs et les fonctions membres communs et trois classes de véhicules spécialisés **dérivées de la classe de base** ayant chacune les attributs particuliers à leur type.

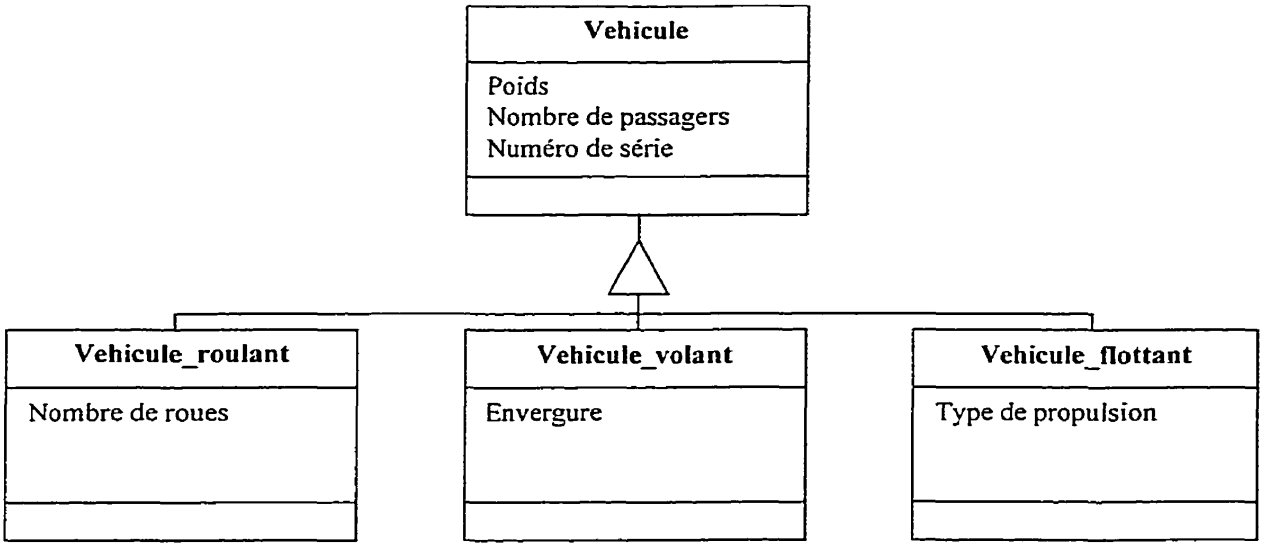


Figure 1.10 – Exemple d'héritage

Les classes dérivées ont tout ce qui est dans la classe de base (attributs, fonctions membres, etc....) en plus des attributs et fonctions qui leur sont particuliers.

1.2.2.3 Principe de l'encapsulation

L'encapsulation est le principe visant à interdire l'accès aux attributs d'une structure de données ceci afin d'imposer l'utilisation d'un unique jeu de fonctions dûment identifiées et documentées. Ainsi on complète le regroupement données-opérations par une garantie d'unicité des opérations. Mais ce concept permet également autre chose : il permet de définir un objet quelconque de la vie réelle via ses attributs et opérations tout en cachant le détail d'implémentation de ces opérations et attributs.

Imaginons par exemple une classe de Date. Cette classe a un attribut : la date, codée sur six caractères en format JJ/MM/AA ainsi les méthodes permettant d'accéder à cette date

et de la modifier : DonneJour, DonneMois, DonneAnnee et les méthodes de modification correspondantes.

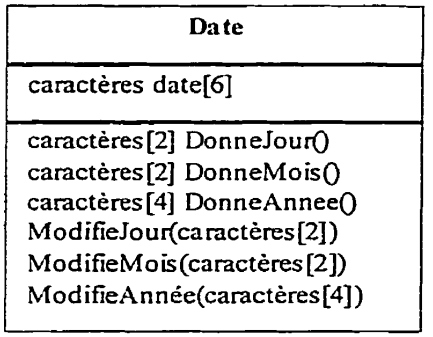


Figure 1.11 – classe date

On remarque que bien que l'année retournée par DonneAnnee et celle à spécifier à ModifieAnnée soient codées sur quatre caractères, la représentation utilisée par la classe ne comporte que deux chiffres. Lors du passage à l'an 2000, cette façon de faire pourrait poser problème : comment déterminer si l'année 00 est 1900 ou 2000 ?

Du fait de l'encapsulation des attributs, on peut facilement remédier à cela en représentant l'année non avec deux mais avec quatre caractères et en modifiant les méthodes DonneAnnee et ModifieAnnee. La modification de l'implémentation est totalement invisible hors de la classe.

En résumé, on a pu maintenir une classe en modifiant la représentation interne de ses attributs à un coût minimal tout en ayant qu'un nombre limité de fonctions à modifier : celle de l'interface d'accès. C'est à la fois l'objectif de l'encapsulation et son avantage.

1.2.3 Les classes en C++

Cette section présente l'application des concepts abordés dans la partie précédente en ajoutant successivement à la structure classiquement utilisée en C la plupart des mécanismes fournis par le C++. Les mots clés class et struct seront utilisés indifféremment. La différence est uniquement dans le mode de protection par défaut et sera explicitée dans la partie encapsulation.

1.2.3.1 Fonctions membres

1.2.3.1.1 Syntaxe en C++

On a besoin de regrouper les fonctions et les données : ceci est fait en C++ en déclarant les fonctions dans le corps de la structure :

```
struct Date{
private :
      char date[8] ;
public :
      char* DonneJour();
char* DonneMois();
char* DonneAnnee();
char* ModifieJour(char param[2]);
char* ModifieMois(char param[2]);
char* ModifieAnnee(char param[4]);
};
```

Pour appeler les fonctions membres de la classe, on utilise la même syntaxe que pour les attributs à savoir :

```
struct Date an2000;
an2000.ModifieJour(«01»);
an2000.ModifieMois(«01»);
an2000.ModifieAnnee(«20000»);
```

Si la date est un pointeur on utilise la syntaxe :

```
struct Date* an2000  = new Date;
an2000->ModifieJour(«01»);
an2000->ModifieMois(«01»);
an2000->ModifieAnnee(«2000»);
. . .
delete an2000;
```

On remarque dans le dernier exemple l'apparition des opérateurs **new** et **delete**. Ces opérateurs accomplissent les tâches d'allocation/destruction et plus encore comme nous le verrons dans la section suivante, après avoir présenté des fonctions membres particulières : les constructeurs et destructeurs ainsi que la surcharge des fonctions membres.

1.2.3.1.2 Cas particulier : les constructeurs et destructeurs

Il est fondamental de connaître l'état d'une instance d'une classe dès son initialisation. Imaginons en effet le code suivant :

```
struct Date an2000;
printf(« jour: %s »,an2000.DonneJour());
```

On accède là à un attribut non-initialisé et pouvant donc contenir absolument n'importe quoi.

Afin de contrôler cette initialisation, on dispose d'une fonction membre particulière : le constructeur. C'est une fonction portant le même nom que la classe et qui sera appelée automatiquement lors de l'instanciation de la classe. Comme il peut être souhaitable de pouvoir définir plusieurs manières différentes d'initialiser une instance, il est possible de définir plusieurs différentes versions d'un constructeur en suivant les règles de la surcharge, abordées dans la sous-section du même nom.

On aura donc la syntaxe suivante :

```
struct Date{
      char date[8] ;
      char* DonneJour();
      char* DonneMois();
      char* DonneAnnee();
      char* ModifieJour(char param[2]);
      char* ModifieMois(char param[2]);
      char* ModifieAnnee(char param[4]);
      Date(); //constructeur par défaut
      Date(char dateParam[8]) ; //autre version du constructeur
};
```

Il peut être également intéressant de définir la manière dont doit être détruite un objet à la fin de sa vie (c'est à dire lors de sa destruction explicite pour un objet alloué sur le tas, lorsque le pointeur de programme sort de sa portée dans le cas d'une variable automatique). Cela peut servir à désallouer un attribut alloué dynamiquement dans le constructeur, à fermer un fichier ouvert dans le constructeur et toujours ouvert ou toute autre chose.

Toutes ces opérations sont regroupées dans une unique fonction membre particulière le destructeur. Le destructeur porte le même nom que la classe, précédée d'une tilde. On aura donc la syntaxe suivante :

```
struct Date{
      char date[8] ;
      char* DonneJour();
      char* DonneMois();
      char* DonneAnnee();
      char* ModifieJour(char param[2]);
      char* ModifieMois(char param[2]);
      char* ModifieAnnee(char param[4]);
      Date(); //constructeur
      ~Date() ; //destructeur
};
```

Il est important de noter que le destructeur ne prend jamais de paramètres et qu'il ne peut pas être surchargé (i.e., qu'on ne peut pas en définir plusieurs versions en surchargeant la méthode, selon les règles édictées plus loin).

1.2.3.2 Encapsulation

1.2.3.2.1 Syntaxe

L'encapsulation se fait en C++ par l'utilisation des spécificateurs de protection **private**, **protected** et **public** dans le corps de la déclaration d'une structure ou d'une classe.

Ces mots clés permettent de spécifier la visibilité (c'est à dire le domaine duquel on peut accéder à la variable) de tout ce qui suit dans la déclaration de la classe (ou de la structure), jusqu'à la fin de la classe (ou de la structure) ou jusqu'au prochain spécificateur de protection.

Si une variable (ou toute autre chose) est déclarée **private** (privée), elle est invisible pour les fonctions hors de la classe. Seules les fonctions membres de la classe peuvent donc y accéder. Le spécificateur **protected** rend également la variable (ou toute autre chose) invisible hors de la classe. La différence avec **private** est au niveau de l'héritage : les variables protégées sont visibles dans les classes dérivées, au contraire des variables privées.

Enfin, les variables (ou toute autre chose) déclarées **public** sont accessibles de partout, d'une fonction membre de la classe, d'une classe dérivée ou d'une fonction non membre de la classe.

Exemple :

```
struct A{
private :
      int B ;
```

```
      void fnct1() ;
protected :
      int C ;
      void fnct2() ;
public :
      void fnct3() ;
} ;
```

Dans l'exemple ci-dessus, seule la fonction `fnct3` peut être appelée par une fonction non membre de la classe. En revanche `fnct3`, membre de la classe A, a, tout comme fnct2 et `fnct1`, accès à tous les attributs et fonctions de la classe.

1.2.3.2.2 Mode de protection par défaut

La différence entre les classes et les structures est uniquement au niveau du mode de protection par défaut : il est **public** dans les structures et **private** dans les classes. C'est donc pour insister sur l'évolution depuis le C et sur la nécessité de respecter le principe de l'encapsulation que l'on utilise **class** plutôt que **struct**.

1.2.3.2.3 Amitié

L'amitié peut être brisée en C++ par la directive **friend**. Elle permet de spécifier une fonction ou une classe dont les fonctions doivent être considérées comme amies et auxquelles on accorde un accès non restreint à tous les attributs et toutes les fonctions de la classe. Ainsi la structure suivante spécifie que la structure A est son amie, ainsi que la fonction `fonction_amie` et lui accorde donc l'autorisation d'utiliser tous ses attributs et fonctions quel que soit leur mode de protection.

```
struct B{
friend class A ;
friend void fonction_amie() ;
      /*déclaration d'attributs et de fonctions…*/
};
. . .
void fonction_amie() ;
```

On remarque que le nom de la classe doit être précédé du mot clé **class** et que ce n'est pas uniquement le nom de la fonction mais son prototype complet que l'on doit spécifier.

La directive friend n'est pas affectée par les spécificateurs de mode de protection.

1.2.3.3 Surcharge des fonctions et paramètres par défaut

1.2.3.3.1 Paramètres par défaut

Il est possible en C++ de définir des valeurs par défaut pour les paramètres d'une fonction. La syntaxe est la suivante :

```
type_parametre non_parametre = valeur_par_defaut
```

Ce qui signifie donc que si une fonction lambda retournant un entier et prenant comme paramètre un entier ayant une valeur par défaut est définie :

```
int lambda(int param = 0) ;
```

Il est possible d'invoquer cette fonction de manière classique, en spécifiant une valeur pour le paramètre mais comme une valeur par défaut est spécifiée, il est possible de l'invoquer sans spécifier aucun paramètre, la valeur par défaut étant alors utilisée.

Comme un paramètre par défaut peut être ou non spécifié, il est interdit de définir des paramètres n'ayant pas de valeur par défaut après le premier paramètre ayant une valeur par défaut. Si cette règle n'existait pas, on aurait des ambiguïtés, comme dans le cas de cette déclaration incorrecte :

```
int lambda(int param1,int param2 = 0,int param3);
```

Si l'on invoque cette fonction en lui passant deux entiers, à quel paramètre correspondrait alors le second entier, param2 ou param3 ?

La déclaration suivante serait par contre correcte :

```
int lambda(int param1,int param2 = 0,int param3);
```

1.2.3.3.2 Surcharge des fonctions

La surcharge des fonctions, également appellée, entre autre, surcharge des opérateurs, est un mécanisme permettant de définir différents comportements pour un même nom de

fonction. Ce mécanisme permet donc de nommer d'une manière unique une même opération déclinée en plusieurs versions adaptées à des paramètres différents.

La surcharge d'une fonction en C++ se fait selon la règle suivante :

Deux surcharges de la même fonction doivent être différentiables uniquement selon le type des paramètres passés à cette fonction.

On remarque que le type de la valeur de retour ne peut pas être utilisé pour différencier deux surcharges.

La surcharge suivante est donc légale :

```
int lambda(int un) ;
int lambda(int un, int deux) ;
```

La surcharge suivante est par contre illégale :

```
int lambda(int un) ;
int lambda(int un, int deux = 0) ;
```

Il est en effet impossible de différencier les deux versions si seulement un entier est passé en paramètre.

1.2.3.3.3 Opérateurs

Il est également possible en C++ de surcharger les opérateurs, comme '+', '*', '-', etc....

Le mot clé **operator** précédant l'opérateur proprement dit constitue le nom de l'opérateur. Sa valeur de retour et les paramètres dépendent du type de l'opérateur et du fait que l'opérateur soit une fonction membre de la classe ou non.

Un opérateur + prenant comme paramètre un entier spécifiant un nombre de jours à ajouter peut être ajouté à la classe date.

```
class Date{
      char date[8] ;
public :
      Date(); //constructeur
      ~Date() ; //destructeur
      Date& operator+(int NbJours);
      char* DonneJour();
      char* DonneMois();
      char* DonneAnnee();
      char* ModifieJour(char param[2]);
      char* ModifieMois(char param[2]);
      char* ModifieAnnee(char param[4]);
};
```

Il est possible de surcharger les opérateurs, comme pour n'importe quelle autre fonction.

On peut surcharger tous les opérateurs sauf '.', '::', '?:', 'sizeof'.

1.2.3.4 Héritage

L'héritage en C++ se fait lors de la déclaration de la classe, selon un mode de protection.

Soit une classe A :

```
class A{
      //attributs
      //fonctions membres
} ;
```

et une classe B dérivant publiquement de la classe A :

```
class B : public A{
      //attributs
      //fonctions membres
} ;
```

On retrouve dans la classe B tous les attributs et toutes les fonctions de la classe A, comme expliqué dans la section « Héritage ».

Le mode de protection spécifié lors de l'héritage influence le mode de protection des attributs de la classe de base dans la classe dérivée selon le tableau suivant :

Tableau 1.1 – Modes d'héritages

		Mode de protection dans la classe de base		
		Public	protected	private
Mode d'héritage	public	Public	Protected	Inaccessible
	protected	Protected	Protected	Inaccessible
	private	Private	Private	Inaccessible

Le mode de protection le plus restrictif du mode de déclaration et du mode d'héritage s'impose. Naturellement, les membres privés de la classe de base sont (par définition) inaccessibles depuis la classe dérivée.

Le mode d'héritage utilisé dans la quasi-totalité des cas est l'héritage public.

1.2.3.5 Surcharge de fonctions héritées et fonctions virtuelles

1.2.3.5.1 Surcharger les fonctions membres de la classe de base

L'héritage en C++ ne propose pas seulement une manière d'organiser logiquement la composition de nos entités et de leurs spécialisations. Il est possible également de fournir dans les classes dérivées des versions des fonctions de la classe de base adaptées à la classe dérivées.

Soit le code suivant :

```
class A{
/*déclarations*/
      void fonction_exemple() ;
} ;


class B : public A{
      void fonction_exemple () ;
} ;
```

La structure B surcharge la version de la `fonction_exemple` de la classe A. Cela signifie que si l'on appelle fonction_exemple sur une instance de la classe B, ce sera la version fournie par la classe B qui sera fournie, en lieu et place de celle de la classe A.

1.2.3.5.2 Fonctions virtuelles

En l'état le mécanisme de surcharge de fonctions héritées est déjà un outil précieux. Il est complété par celui de fonction virtuelle. La déclaration d'une fonction virtuelle se fait de la même manière que pour une fonction classique, en précédant cette déclaration du mot clé **virtual** :

```
class C{
/*déclarations*/
      virtual void fonction_exemple() ;
} ;


class D : public C{
/*déclarations*/
      void fonction_exemple() ;
} ;
```

On remarque que la classe D n'utilise pas le mot clé virtual. Ce n'est en effet pas nécessaire : en C++, la première déclaration dans l'arbre d'héritages de classes fixe le caractère virtuel ou non d'une fonction. Il est cependant conseillé pour plus de clarté de réutiliser ce mot clé virtual dans les classes dérivées surchargeant une fonction virtuelle, ceci afin de rendre le code source plus lisible.

L'intérêt des fonctions virtuelles est le suivant : si une instance d'une classe faisant partie d'une hiérarchie de classes est pointée par un pointeur du type d'une classe de base de la classe de l'instance et que ce pointeur est utilisé pour invoquer une fonction virtuelle, c'est la version correspondant le mieux au type réel de l'instance pointée qui sera utilisée et non pas le type du pointeur.

Exemple :

```
C instance_C ;
C *ptr_C ;
D instance_D ;

instance_C.fonction_exemple() ; // la version de la classe C est
utilisée

instance_D.fonction_exemple() ; // la version de la classe D est
utilisée

ptr_C = &instance_C ;

ptr_C->fonction_exemple() //la version de la classe C est utilisée

ptr_C = &instance_D ;

ptr_C->fonction_exemple() //la version de la classe D est utilisée
```

Il est donc possible avec ce mécanisme de définir un ensemble de comportements (de fonctions) dans la classe de base qui n'auront de sens que dans ses dérivées. En l'état actuel de nos connaissances, cela nous obligerait à fournir une implémentation de la version de la classe de base. Hors celle-ci peut ne pas être assez spécialisée pour faire l'objet d'une implémentation pertinente. Heureusement le C++ pallie cet inconvénient en permettant qu'une fonction soit **virtuelle pure**.

1.2.3.5.3 Fonctions virtuelles pures et classes abstraites

Une fonction virtuelle pure est une fonction virtuelle n'ayant pas d'implémentation. Pour spécifier qu'une fonction est virtuelle pure il faut mettre '=0' derrière la déclaration de la fonction, juste avant le ';' de fin de ligne :

```
class A{
public :
      int virtuelle_pure(int a,int b,int c) = 0 ; //fonction virtuelle
pure
} ;
```

Une classe ayant au moins une fonction virtuelle restée pure (dont elle ou une de ses classes de base ne fournit pas une implémentation) est une classe abstraite. Cela signifie que cette classe restera une abstraction. La classe étant incomplète, elle ne pourra pas être instanciée.

Dans l'exemple suivant, A est une classe virtuelle pure, B également, mais C ne l'est plus :

```
class A{
public :
      void virtuelle_pure1() = 0 ;
      void virtuelle_pure2() = 0 ;
} ;


class B : public A{
public :
      void virtuelle_pure1() ; //une implémentation de virtuelle_pure1
est fournie.

//Il manque toujours une implémentation de virtuelle_pure2

} ;


class C : public B{
public :
      void virtuelle_pure2() ;      //implémentation de virtuelle pure
```

```
2 fournie, la classe
            //ne compte plus de méthode abstraite et peut donc être
instanciée
} ;
```

On remarque que la classe C ne fournit pas d'implémentation spécifique de virtuelle_pure1. Si l'on invoque cette méthode sur une instance de C ou par le biais d'un pointeur, c'est la version « la plus proche » en terme d'héritage qui sera utilisée, soit celle de B.

1.2.3.6 Classes et fonctions paramétrées

1.2.3.6.1 Classes

Il est possible en C++ de définir des classes et des fonctions en ne spécifiant un ou plusieurs types manipulés par cette classe ou cette fonction qu'au moment de l'instanciation. On déclare pour cela des classes ou des fonctions patrons, manipulant des types abstraits. Ces classes et fonctions sont dites **template** en C++.

La syntaxe est :

```
template <class T> class complexe{
      T partie_reelle ;
      T partie_imaginaire ;
/*. . .*/
} ;
```

La substantifique moelle réside dans la partie template <class T> de la déclaration. On spécifie là que la classe est une classe patron (couramment appelée template) et qu'elle manipule un type abstrait T qui sera précisé lors de l'instanciation. On dispose alors, dans la déclaration de cette classe, de ce type abstrait en plus des types dont on a habituellement l'usage.

Il est impératif de préciser le type réel du type abstrait lors de l'instanciation selon la syntaxe suivante :

```
complexe<double> nombre_complexe ;
```

Ainsi dispose-t-on d'un nombre complexe dont les parties réelle et imaginaire sont de type double.

Ce mécanisme est précieux lors de la définition, par exemple, d'une structure de liste chaînée : on fournit une et une seule implémentation en lieu et place d'une myriade d'implémentations différenciées uniquement par le type manipulé.

Notez bien qu'il est nécessaire d'utiliser la syntaxe suivante lors de l'écriture de l'implémentation des fonctions membres de la classe complexe dans le fichier .cpp :

```
template <class T> complexe<T>::fonction(){
. . .
}
```

1.2.3.6.2 Fonctions

La syntaxe est très similaire pour les fonctions :

```
template <class T> type_retour nom_fonction(/*arguments*/) ;
```

L'appel de la fonction ne nécessite par contre par de spécifier le type abstrait ; une fonction template s'appelle en utilisant la même syntaxe qu'une fonction ordinaire.

1.2.4 Conclusion

Le C++ est un langage riche et élégant offrant simultanément de bonnes possibilités d'abstraction tout en héritant du langage C sa concision et son efficacité. On n'a volontairement pas abordé dans ce chapitre des concepts tels que l'héritage multiple ou les « *namespaces* » (espace de nommage). Ces quelques pages ne constituent qu'une introduction et comme pour tout apprentissage de langage informatique, rien ne remplacera de longues heures de pratique.

Chapitre 2 ÉTUDE PRÉLIMINAIRE

2.1 Présentation du code élément fini du laboratoire : LCMFlot

2.1.1 Introduction

La présente étude a été réalisée par l'auteur dans le cadre de la seconde partie de son stage de fin d'études d'ingénieur. On y explique la chaîne logicielle utilisée, ce que sont les éléments finis, l'approche adoptée avant d'analyser le code de calcul du labo, LCMFlot, et de préconiser quelques modifications en vue de rendre le code plus adaptable, plus générique, plus rapide et plus facile à utiliser.

2.1.2 Chaîne logicielle

LCMFlot est un logiciel de simulation numérique d'injection de pièces par le procédé LCM (« *Liquid Composite Molding* »). De telles pièces sont largement utilisées dans les industries aéronautique et automobile.

Les étapes suivantes sont requises lorsque l'on simule numériquement un processus physique :

- discrétisation de la géométrie en un maillage
- spécification des conditions aux limites
- édition des paramètres de la simulation

- simulation
- traitement des résultats

Les trois premières étapes sont souvent appelées pré-traitement. La génération de maillage est un thème de recherche important. Le leader incontesté en ce domaine est Paul-Louis George et son mailleur GHS3D, développé dans le cadre du projet GAMMA de l'INRIA. Ce logiciel réputé comme étant le plus rapide du monde est incorporé dans un nombre important de logiciels commerciaux. Cependant cette solution est assez coûteuse et n'est donc pas la plus adéquate dans le cadre des travaux de l'équipe LCMFlot. On préfère donc utiliser le mailleur GMSH développé par Jean-François Remacle ou le mailleur Wiomesh développé par Yanick Benoît et fourni avec LCMFlot.

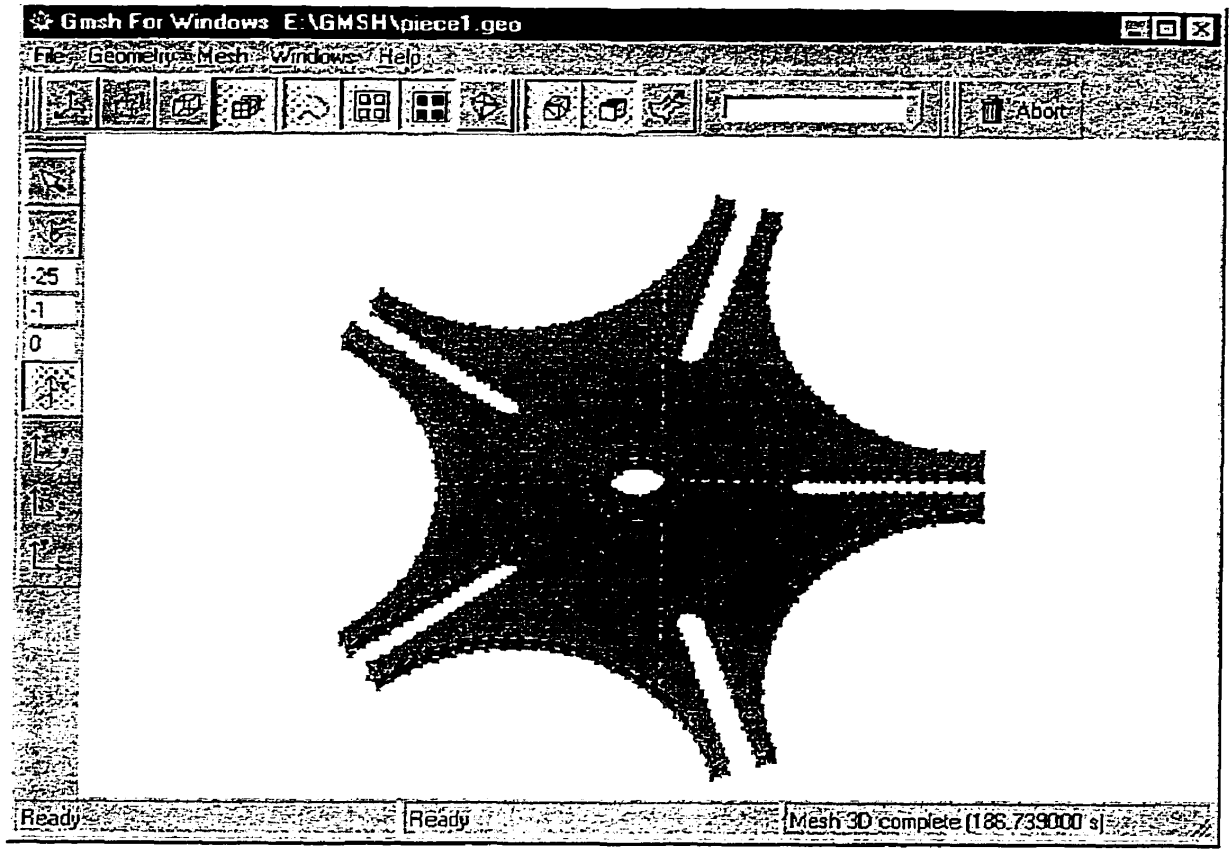


Figure 2.1 - GMSH version WIN32

Une fois la géométrie créée, il est nécessaire de définir des **ZONES** et des **GROUPS**. Les zones sont des regroupements d'éléments géométriques du maillage qui permettent de définir la nature des matériaux utilisés dans la simulation.. Les groupes permettent d'introduire les conditions aux limites sur certaines parties du maillage. Elles peuvent être, dans le cadre de LCMFlot, de nature variées (une pression ou un flux par exemple). La valeur des conditions aux limites est définie à l'aide d'un autre logiciel, **Dataflot**. Une possibilité de spécifier d'autres paramètres est également offerte à l'utilisateur : placement de capteurs, type de simulation, etc.

LCMFlot utilise ensuite toutes ces données pour :

- imposer les conditions aux limites;
- calculer les matrices élémentaires sur chaque élément;
- assembler les valeurs localement calculées dans la matrice globale du problème;
- invoquer le résoluteur;
- effectuer tout autre traitement particulier spécifique à une formulation.

Ce procédé est susceptible d'être itéré jusqu'à la satisfaction d'un critère d'arrêt. Dans le cas du remplissage d'une pièce avec de la résine par exemple, c'est typiquement un temps de remplissage maximal ou le remplissage complet de la pièce.

Les résultats sont ensuite visualisés avec un **post-processeur** (logiciel capable de transcrire les pages de chiffre résultant de la simulation en des données faciles à interpréter). On utilise la plupart du temps GMSH ou Wiomesh pour effectuer cette tâche.

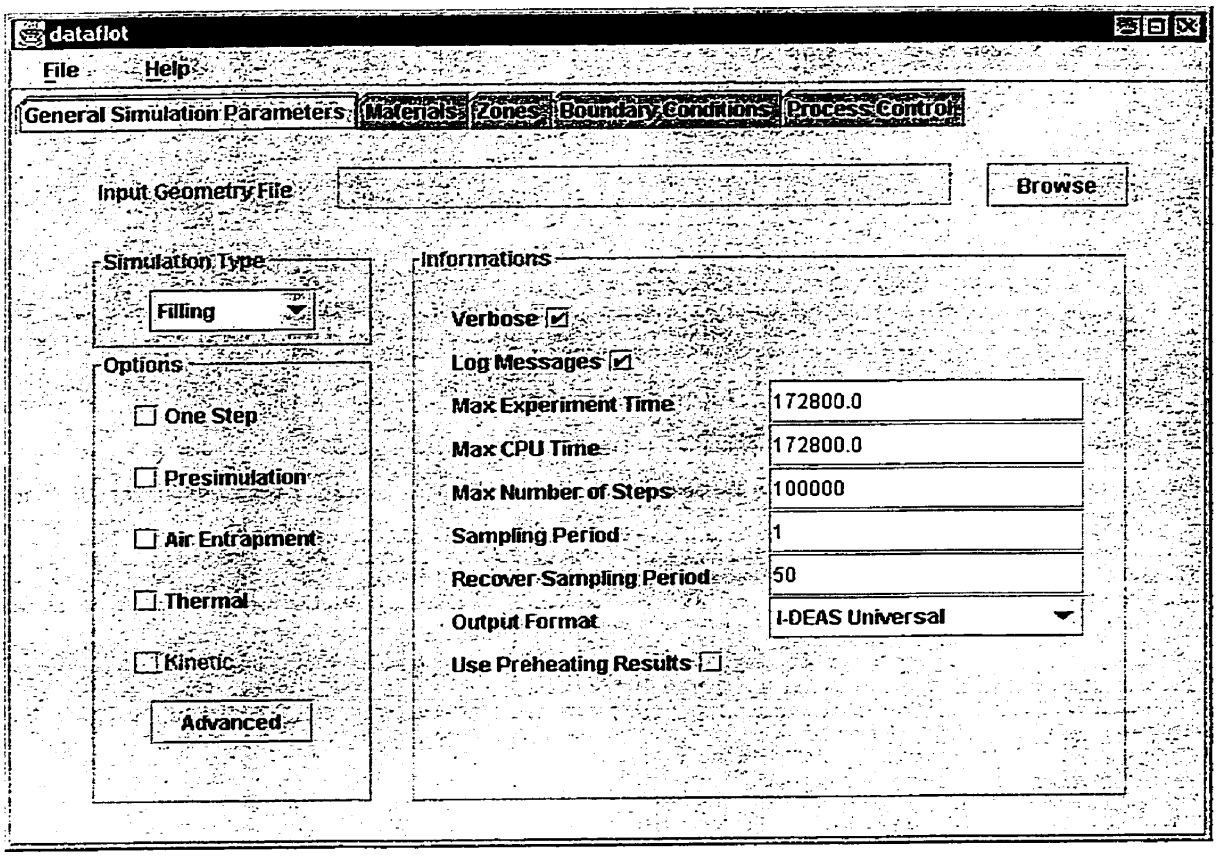


Figure 2.2 – Exemple de fenêtre Dataflot

2.1.3 Exemples

Cette section donne quelques exemples d'utilisation du logiciel LCMFlot.

2.1.3.1 Écoulement de Stokes

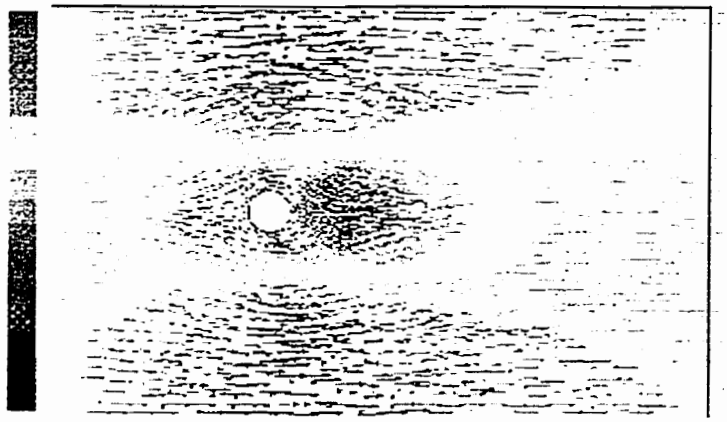


Figure 2.3 - Écoulement de Stokes – vitesses

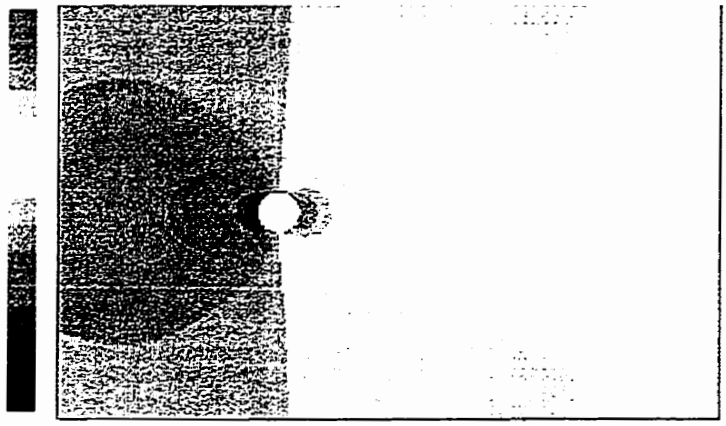


Figure 2.4 - Écoulement de Stokes – pressions

On simule l'écoulement d'un fluide visqueux autour d'un cylindre. Une méthode d'éléments finis non isoparamétrique de type mixte a été utilisée. Une formulation est dite mixte lorsqu'elle contient à la fois la variable principale et sa variable duale. Ici la variable principale est la vitesse, la duale est la pression. La divergence nulle est imposée par un multiplicateur de Lagrange. Cette formulation est habituellement appelée

formulation de Herrmann. Le multiplicateur de Lagrange possède une interprétation physique : il s'agit en fait de la pression. Des fonctions de forme de Serendipi ont été utilisées pour interpoler la vitesse, la pression étant interpolée par des fonctions de forme lagrangienne. On choisit de telles fonctions de forme afin de satisfaire la condition LBB (Ladjenskaja-Babuska-Brezzi) pour que la simulation converge.

2.1.3.2 Procédés LCM

Activité de recherche principale de l'équipe, les procédés LCM consistent à fabriquer des pièces composites en injectant de la résine dans un renfort fibreux. Les figures suivantes montrent cette injection à différents pas de temps. On utilise ici une ligne d'injection afin de réduire le temps de remplissage et éviter ainsi la formation de bulles d'air. Des fonctions de forme lagrangienne du premier ordre et une formulation très spécifique sont utilisées pour simuler l'injection et déterminer l'avance du front de résine.

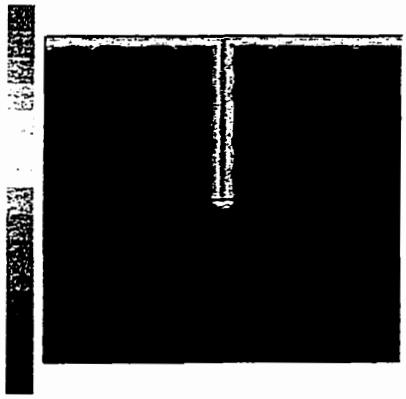


Figure 2.5 - Procédé LCM - pressions - étape 1

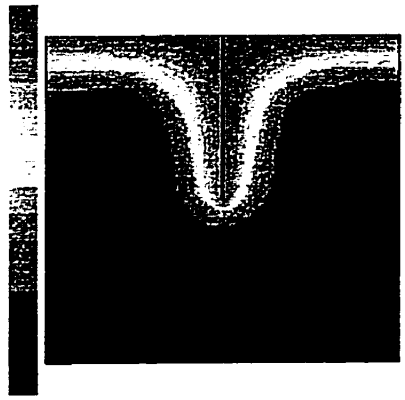


Figure 2.6 - Procédé LCM - pressions - étape 2

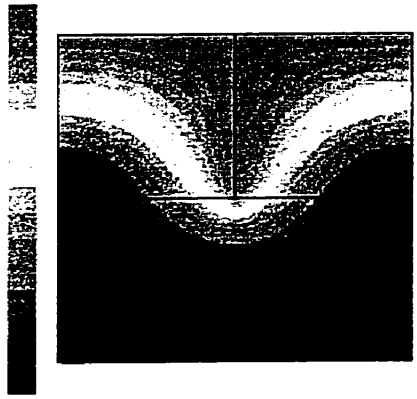


Figure 2.7 - Procédé LCM - pressions - étape 3

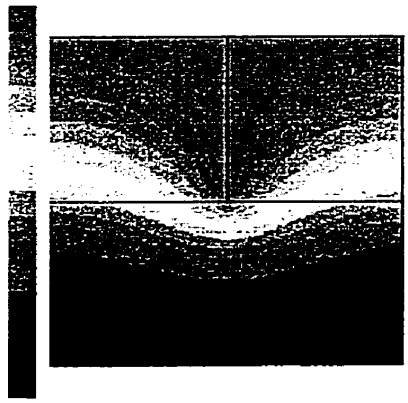


Figure 2.8 - Procédé LCM - pressions - étape 4

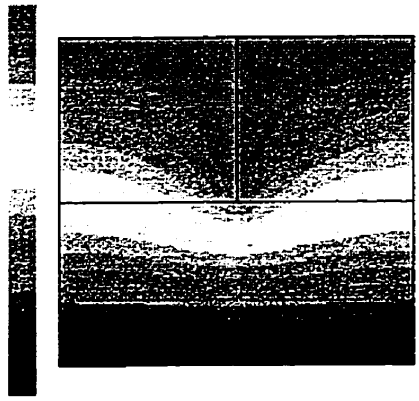


Figure 2.9 - Procédé LCM - pressions - étape 5

2.1.4 Élasticité

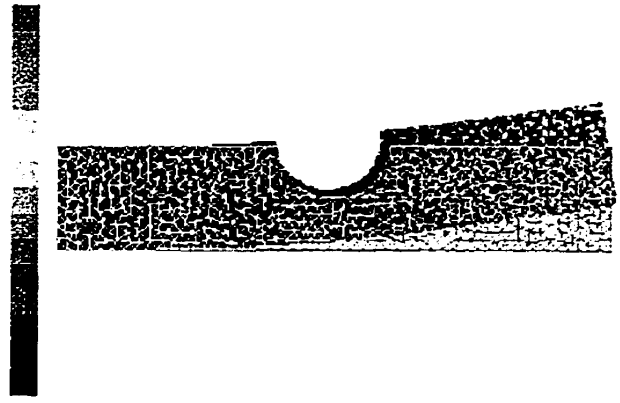


Figure 2.10 - Déformation d'une poutre d'acier

Cette image montre un problème d'élasticité : une poutre d'acier est déformée par une force sur son côté droit. Le côté gauche est encastré (le champ de déplacement est fixé à 0). Les déformations sont amplifiées afin de les rendre plus aisément visibles. Le maillage transparent montre l'état initial, le maillage plein l'état final de la poutre après déformation.

2.2 Une (très) courte présentation de la méthode des éléments finis

2.2.1 Quelques définitions

La méthode des éléments finis permet de résoudre numériquement des équations aux dérivées partielles dont on ne connaît pas de solution analytique. Il y a beaucoup de types d'éléments finis. Comme cette introduction ne veut donner qu'un aperçu de la méthode des éléments finis, on y traitera uniquement le cas des éléments finis conformes.

La méthode des éléments finis implique la définition de trois espaces fonctionnels :

- un espace de fonctions de forme,
- un espace de fonctions test,
- un espace de fonctions de forme géométrique.

2.2.1.1 Fonctions de forme

Une fonction de forme est utilisée pour interpoler la fonction inconnue en tout point d'un élément. L'ensemble des fonctions de forme définissent l'interpolation de la solution en tout point de l'élément fini. Un connecteur est une partie d'un élément géométrique (nœud, arête, face ou encore l'élément lui-même). Supposons l'utilisation d'un élément triangulaire et la définition des nœuds de ce triangle comme étant ses connecteurs.

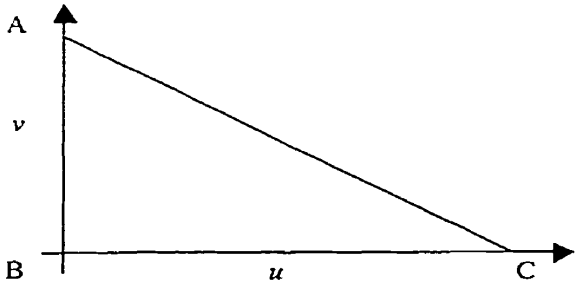


Figure 2.11 - Élément fini triangulaire linéaire

Si les trois fonctions de forme associées aux nœuds sont :

$$FF(A) = v,\ FF(B) = 1\text{-}u\text{-}v,\ FF(C) = u$$

(FF est l'abréviation de Fonction de Forme.)

Alors on définit l'interpolation de la valeur aux connecteurs en tout point X du triangle comme étant :

$$val(X) = val(A)\ FF(A) + val(B)\ FF(B) + val(C)\ FF(C)$$

Supposons maintenant que l'on introduise également des fonctions de forme sur les arêtes du triangle. On devrait alors construire une nouvelle fonction de forme pour chacun de ces nouveaux connecteurs. Un choix typique de fonctions de forme serait le produit des fonctions de forme des deux nœuds de l'arête. Il va de soit que l'on peut définir des fonctions de forme sur les faces d'un élément en 3D. Il existe également des fonctions de forme définies sur l'élément lui-même, et non une de ses parties, comme la fonction bulle.

Un espace fonctionnel doit satisfaire certaines conditions pour pouvoir être utilisé dans le cadre de la méthode des éléments finis. Le lecteur trouvera une présentation de ces conditions dans la plupart des ouvrages introduisant cette méthode numérique de résolution d'équations aux dérivées partielles.

2.2.1.2 Fonctions test

Les fonctions tests sont utilisées comme poids d'intégration durant l'intégration d'un terme sur un élément. Comprendre exactement la signification des fonctions test requiert des connaissances sur la théorie des résidus pondérés. On se contentera ici d'admettre le résultat.

2.2.1.3 Fonctions de forme géométrique

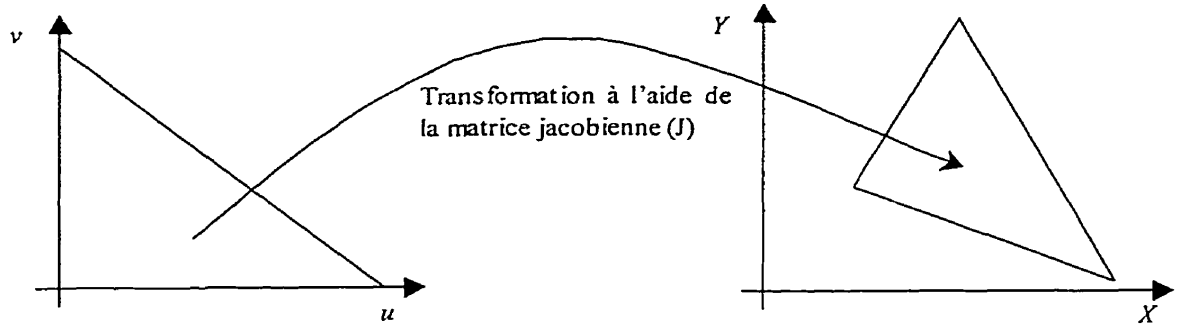


Figure 2.12 - Changement de coordonées

Les fonctions de forme sont définies dans l'espace de référence (u,v,w). Les éléments géométriques sont construits dans l'espace euclidien X, Y, Z. Nous avons à définir un changement de coordonnées de (u,v,w) en (X,Y,Z). Les fonctions de forme géométriques sont utilisées pour effectuer ce changement de variable. Pour l'appliquer on utilise classiquement une matrice jacobienne :

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial v} & \frac{\partial f_1}{\partial w} \\ \frac{\partial f_2}{\partial u} & \frac{\partial f_2}{\partial v} & \frac{\partial f_2}{\partial w} \\ \frac{\partial f_3}{\partial u} & \frac{\partial f_3}{\partial v} & \frac{\partial f_3}{\partial w} \end{pmatrix}$$

Figure 2.13 – Matrice jacobienne

2.2.1.4 Autres définitions

Lorsque les fonctions de forme et les fonctions test sont identiques, la méthode des éléments finis est dite de Galerkin. Si les fonctions de forme et les fonctions de forme géométrique sont identiques, la méthode des élément finis est dite isoparamétrique.

2.2.2 Problème

Notre problème consiste à résoudre l'équation suivante afin de calculer la vitesse v de la résine durant l'injection.

$$div(v) = 0$$

On multiplie cette équation par une fonction test et on en prend une forme faible en intégrant par parties :

$$\int_{\Omega} v\, grad(P')d\Omega - \int_{\Gamma} (v\, n)\, P'\, dS = 0, \forall P' \in H^1$$

L'équation de Darcy :

$$v = -K\, grad(P)$$

nous indique que la vitesse d'écoulement est proportionnelle au gradient de pression, la constante de proportionnalité étant la perméabilité du milieu poreux.

On trouve finalement que si **v.n** est nul sur le bord :

$$\int_{\Omega} grad^{T}(P)\, K\, grad(P')\, d\Omega = 0$$

Dans ces équations, P est la fonction de forme interpolant la pression, P' est la fonction test et K est une loi physique, donnée par une étude expérimentale.

Afin de résoudre cette équation, nous devons :

- définir les espaces fonctionnels,
- définir les degrés de liberté,
- intégrer localement sur chaque élément,
- assembler la matrice globale à partir des intégrations locales,
- résoudre le système d'équations global.

Pour les problèmes non linéaires, ces étapes peuvent être répétées jusqu'à ce qu'un critère d'arrêt soit satisfait.

2.2.2.1 Définition des espaces fonctionnels

Les espaces fonctionnels ont déjà été traités dans les sections précédentes.

2.2.2.2 Définition des degrés de liberté

Les degrés de liberté sont les variables de notre problème. Ils sont les valeurs définies aux connecteurs. Un exemple de degré de liberté peut être la pression en un nœud. Le connecteur est alors le nœud, la valeur de cette pression en ce nœud est le degré de liberté.

2.2.2.3 Intégration locale sur chaque élément

Cette étape consiste à calculer les termes de l'équation (4) localement sur un élément. Plusieurs méthodes d'intégration pourraient être fournies mais pour l'instant, seule la méthode de Gauss est utilisée. Bien sûr, il ne s'agit pas d'une intégration symbolique, mais numérique. Dans la méthode de Gauss, les points d'intégration sont définis comme étant les racines d'un polynôme de Legendre. Nous ne définissons pas ici ce qu'est un polynôme de Legendre, encore moins comment trouver ses racines. Connaissant ces points on calcule pour chaque point d'intégration une matrice dont la cellule ij est égale à :

$$Cellule_{ij} = FT_i\, FT_j$$

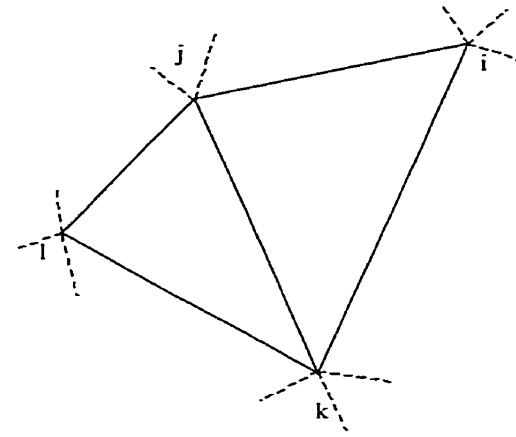
Où FT_i est la $i^{ème}$ fonction test et FF_j la $j^{ième}$ fonction de forme des bases de fonctions de forme et test. Prenons l'exemple d'un élément triangulaire sur lequel on aurait choisi des fonctions de forme et des fonctions test linéaires de Lagrange. Comme on avait défini une fonction de forme en chaque nœud, on a donc 3 fonctions de forme et 3 fonctions test et une matrice 3x3 en chaque point. Le résultat de l'intégrale est juste la somme de toutes ces matrices, pondérées par les poids associées aux points de Gauss.

$$\int_{\Omega} f(u,v,w)\, d\Omega = \sum_{Points\ de\ Gauss} f(u_{pg}, v_{pg}, w_{pg})\, poids_{pg}$$

2.2.2.4 Assemblage de la matrice globale

Une gestion efficace des degrés de liberté (ddl) est essentielle à un bon logiciel d'éléments finis. Un gestionnaire de degrés de liberté est utilisé dans LCMFlot. Sa conception fort astucieuse permet de grandement généraliser l'opération d'assemblage. Si un degré de liberté a été fixé, on soustrait sa valeur au membre de droite. Sinon, la valeur ajouté en ce ddl est ajouté à la matrice globale. La position dans la matrice est déterminée de la manière suivante : chaque connecteur génère une clé unique en tenant

compte de la fonction de forme utilisée et de la quantité physique. Ainsi, on peut obtenir aisément une clé unique pour représenter une fonction de forme en un connecteur et une autre clé unique pour représenter une fonction test en ce même connecteur. En se servant de la clé générée à l'aide de la fonction test pour déterminer la ligne et de celle qui est générée à l'aide de la fonction de forme pour déterminer la colonne, on assemble correctement les valeurs dans la matrice globale : les contributions de différents éléments en un même nœud, par exemple, sont bien assemblées au même endroit dans la matrice globale.



Deux matrices locales sont calculées, une pour le triangle (j,j,k), l'autre pour le triangle (j,k,l).

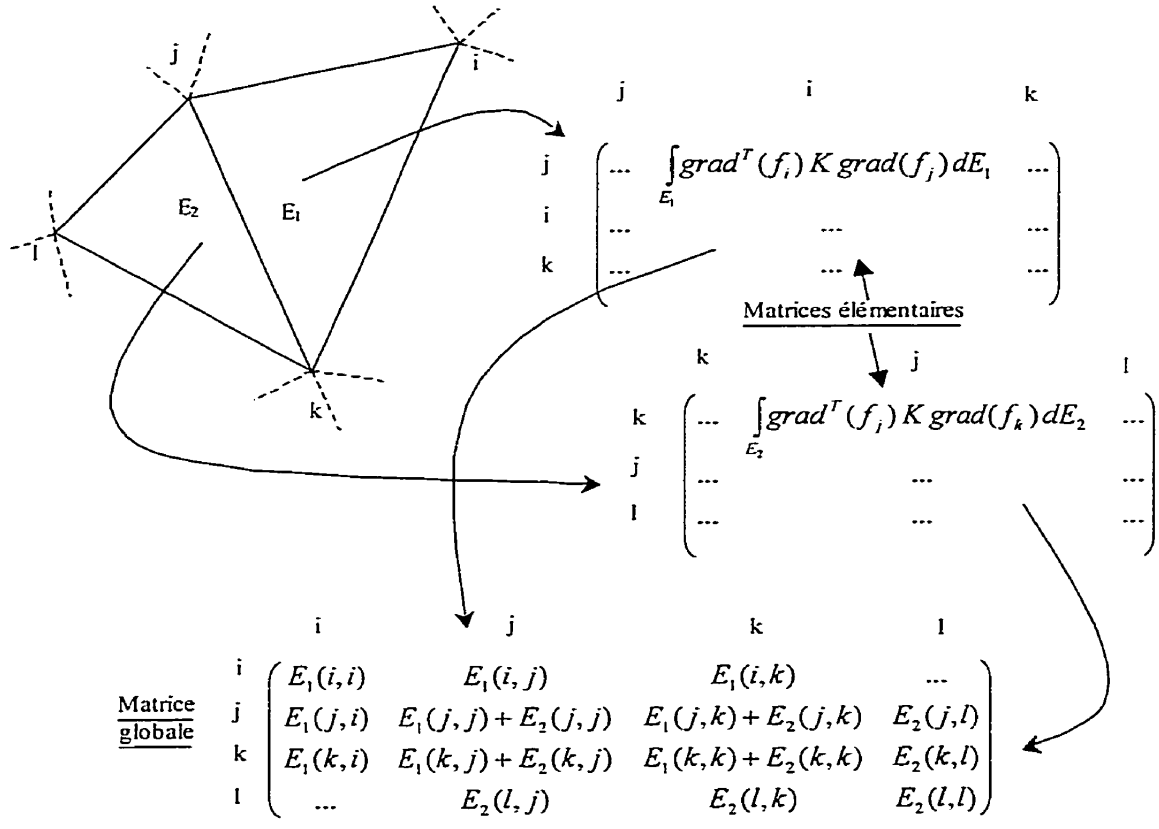


Figure 2.14 – Assemblage

2.3 Analyse du code élément fini de LCMFlot

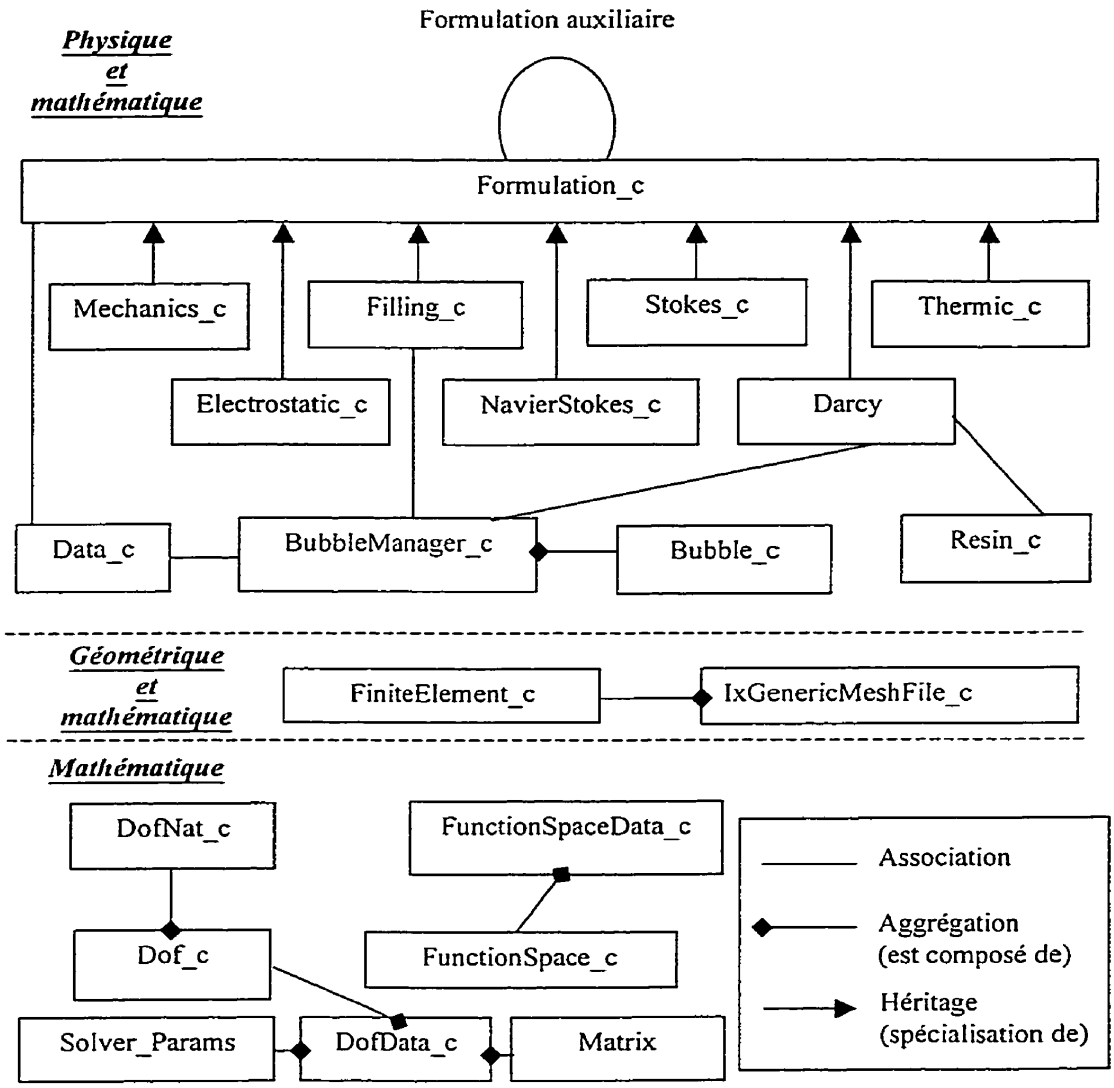


Figure 2.15 - Diagramme de classes de LCMFlot

La structure de LCMFlot tente, sans y arriver totalement, de séparer les trois aspects de la méthode des éléments finis :

- l'aspect géométrique (maillage et éléments géométrique);
- l'aspect mathématique (fonctions de forme, schéma de résolution, degrés de liberté);
- l'aspect physique (lois de comportement, traitements spécifiques effectués dans la formulation d'un problème donné).

Certes, une partie complète de LCMFLot est essentiellement algébrique et comprend la gestion des degrés de liberté et l'interface avec le résoluteur. La partie géométrique devrait être uniquement constituée des éléments géométriques et du maillage. Ici elle est mélangée à la partie mathématique du fait de l'implémentation des fonctions de forme dans l'élément géométrique, appelé *élément fini*. Enfin, la partie physique comprend également des fonctions d'assemblage de termes, qui ne sont pourtant que purement mathématiques.

Il est intéressant de remarquer que l'on retrouve dans les classes de LCMFlot la plupart des entités intervenant dans la méthode des éléments finis :

- les espaces fonctionnels (classe `FunctionSpace_c`),
- les formulations (classe de base `Formulation_c` et ses dérivées),
- le maillage (classe `IxGenericMeshFile`),
- les éléments finis (classe `FiniteElement_c`),
- la matrice globale (classe `Matrix`),

- les degrés de liberté (classe `Dof_c`) et leur gestionnaire (classe `DofManager_c`),

- quelques entités spécifiques au procédé LCM : les bulles (classe `Bubble_c`) et leur gestionnaire (classe `BubbleManager_c`) ainsi que la résine (`Resin_c`).

On se propose maintenant de discuter des deux classes les plus importantes : celle qui définit les formulations et celle qui définit les éléments finis.

2.3.1 Formulation_c

La classe formulation est une classe abstraite (qui ne peut être instanciée) définissant une interface de fonctions virtuelles pures communes à toutes les formulations. Elle fournit également quelques méthodes utilitaires. Les fonctions virtuelles pures sont :

- MaterialValid

- DiffusiveCharacteristicScalar

- DispersiveCharacteristic

- ConvectiveCharacteristic

- DirichletCharacteristic

- RobinCharacteristic

- TreatmentOfGroups

- TreatmentOfFormulation

Le rôle de la fonction `MaterialValid` est d'indiquer si les éléments recouvrant le matériau courant doivent être calculés. On peut, par exemple, ne pas souhaiter calculer la pression de l'air ambiant lors d'une injection de résine dans un renfort. Les fonctions `XXXXCharacteristic` calculent toutes des caractéristiques (le plus souvent, d'un matériau) différentes. La fonction `TreatmentOfGroups` impose les conditions aux limites. Trois types de conditions aux limites sont actuellement supportées par LCMFlot :

- condition de Dirichlet (imposition directe de la variable),
- condition de Neumann (imposition de la dérivée de la variable),
- condition de Robin (imposition de la variable et de sa dérivée).

On laisse le détail de l'imposition des conditions aux limites à la charge des formulations, bien que cela puisse être généralisé.

Enfin la méthode `TreatmentOfFormulation` constitue le cœur de la formulation en précisant quelles conditions aux limites doivent être imposées, quels termes doivent êtres assemblés et quel schéma de résolution doit être utilisé (linéaire ou non par exemple).

2.3.2 FiniteElement_c

Une unique classe d'élément fini implémente tous les types d'éléments offert par LCMFlot. Il contient par conséquent une quantité impressionnante de code « switch/case » pour évaluer la bonne fonction de forme sur un élément. Ceci en rend la lecture malaisée. La classe d'éléments finis implémente également un intégrateur de Gauss, le changement de coordonnées par le biais de l'utilisation d'une matrice

jacobienne, les différentes fonctions de formes implémentées comme autant de fonctions membres.

Chapitre 3 ÉTUDE BIBLIOGRAPHIQUE

3.1 Étude bibliographique – Revue de quelques designs

On se propose dans cette partie d'étudier les atouts et inconvénients inhérents à chaque design. Il est à noter que la majorité des publications présentées ici sont postérieures au début de cette thèse de maîtrise. C'est une bonne indication de la jeunesse de ce thème et de l'intérêt nouveau qu'il suscite chez les concepteurs de logiciel de calcul par éléments finis.

3.1.1 « Object Oriented programming in non linear finite element analysis »

Cet article de MM. Yves Dubois-Pèlerin et Pierre Pegon, du centre de recherche de l'Union Européenne à Ispra (Italie) est focalisé sur la modularité accrue obtenue en adoptant l'approche orientée pour modéliser la méthode des éléments finis. La hiérarchie de classes proposée est la suivante :

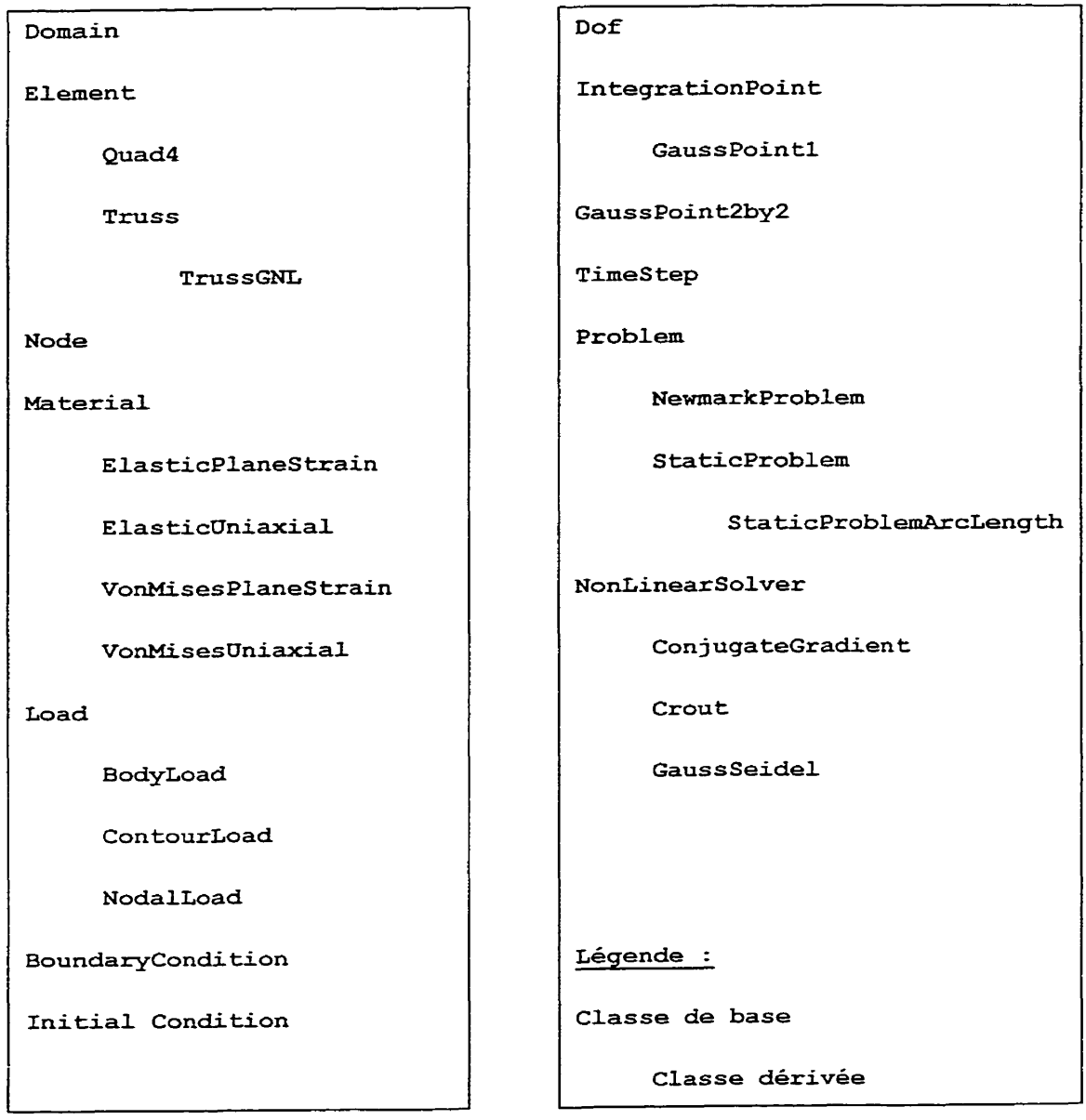


Figure 3.1 - Hiérarchie de classes proposée par le premier article

Plusieurs aspects importants de cette modélisation doivent être soulignés. Il est à noter, tout d'abord que les aspects géométriques, mathématiques et physiques sont peu ou pas séparés. Un élément géométrique a connaissance, par exemple, de la fonction de forme

qu'il doit utiliser (et qui n'est donc pas aisément modifiable) ainsi qu'une liste de points d'intégration. Ensuite, la physique est mal séparée de l'élément fini, malgré l'apparition, certes intéressante, d'une hiérarchie de matériaux. Un élément fini sera responsable de l'assemblage local d'une matrice de rigidité. Il est étonnant de remarquer que, et cela contraste singulièrement avec ce qui vient d'être analysé, les algorithmes de résolution non linéaire sont séparés du reste. Cette approche est intéressant car elle rend les algorithmes interchangeables.

3.1.2 "Aspects of an object oriented finite element environment"

Cet article de Th. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier et S. Commend de l'École Polytechnique Fédérale de Lausanne (Suisse) présente un sujet beaucoup plus ambitieux que cette thèse, comme l'indique son titre. On restreint donc ici l'étude à la partie concernant notre étude, soit la modélisation du cœur de calcul par éléments finis.

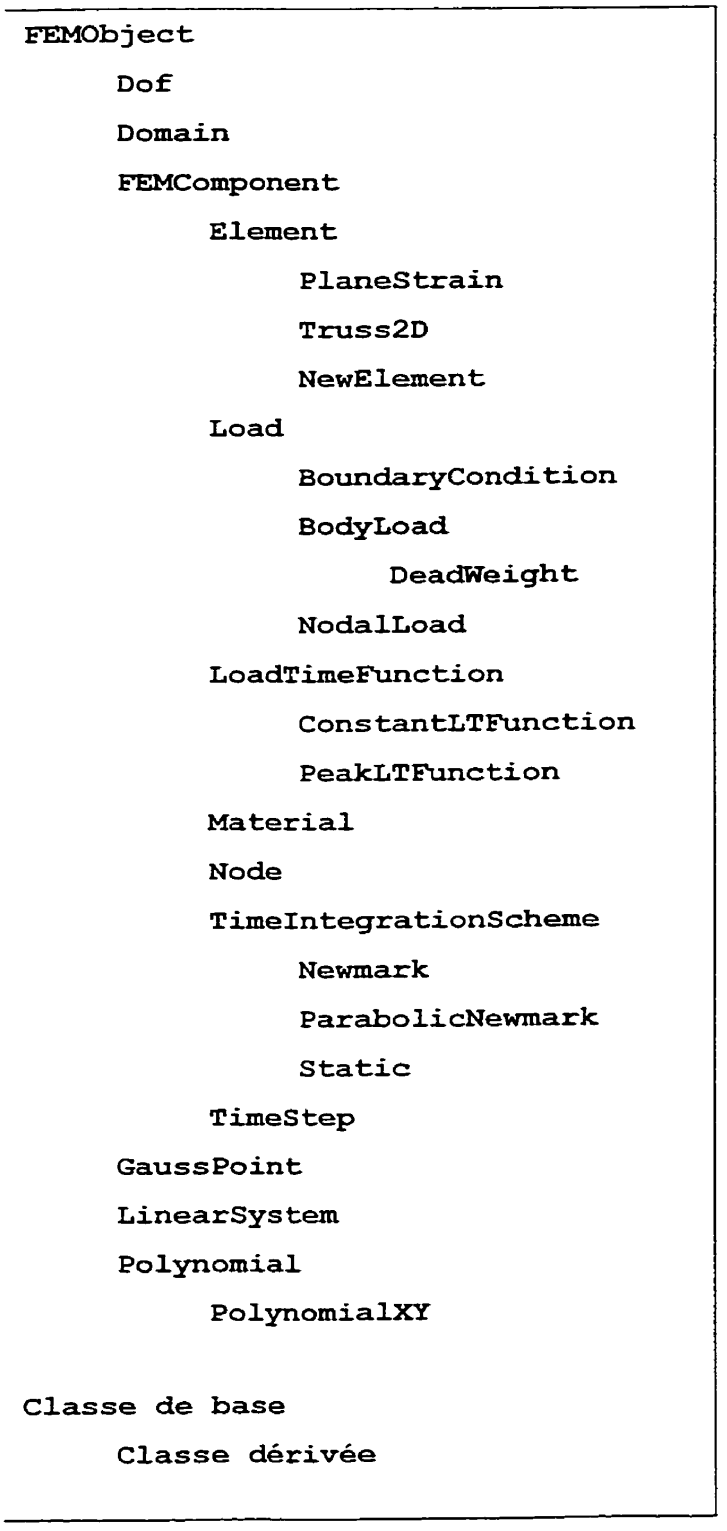


Figure 3.2 - Hiérarchie de classes proposée par le second article

La première critique concernant cet article est qu'il traduit un souci de modéliser, sans se préoccuper des conséquences que cette modélisation peut avoir sur l'efficacité. Cela n'est certes pas l'objet de leur travail, qu'ils présentent comme étant principalement un outil de prototypage d'applications éléments finis, mais cet aspect est primordial dans notre cas, puisque nous prétendrons fournir en résultat de notre travail une base solide permettant l'écriture rapide d'applications aux performances comparables à celles de logiciels commerciaux. Cette perte d'efficacité est principalement dûe à l'utilisation du langage « SmallTalk » qui, bien qu'excellent en ce qui concerne la modélisation purement objet, n'en reste pas moins un langage interprété et notoirement plus lent que le C++, qui est lui compilé. La justification du choix de « SmallTalk », soit le souci d'avoir une approche purement orientée objet, motive une seconde critique, étayée par cet extrait de la publication :

« In fact, when the set of data to encapsulate and the associated methods are easy to identify, for example when a physical correspondence exists, then the chances of building a class which is inappropriate are small »

que l'on peut traduire par :

« En fait, quand le jeu de données à encapsuler et les méthodes qui leur sont associées sont faciles à identifer, par exemple quand une correspondance physique existe, alors les chances d'avoir conçu une classe inappropriée sont très réduites »

Ce qui signifie donc que l'on se condamne, en suivant l'un des dogmes de l'orienté objet qui est le regroupement des données et des opérations, à ne JAMAIS séparer données et algorithmes, ce qui implique que l'on ne pourra donc pas, par exemple, obtenir l'élégante séparation obtenue par les auteurs de l'article précédent. Il est cependant intéressant de noter que, bien que ce principe soit explicitement exprimé par les auteurs,

des classes d'algorithmes apparaissent naturellement, résistent au carcan de l'orienté objet. On note par exemple une classe « Domaine », équivalente à la classe « Formulation » de FEMView et qui implémente une formulation. Il est à souligner que ce n'est pas l'encapsulation que nous condamnons, procédé au demeurant très sain et permettant d'éviter que des accès anarchiques aux attributs d'un objet ne soit effectués. On s'attaque plutôt au postulat que cette encapsulation implique nécessairement un regroupement algorithme/données. La non séparation des éléments géométriques des fonctions de forme et de l'aspect physique provoque l'apparition d'une kyrielle d'éléments finis spécialisés. L'utilisation d'un élément quadrangulaire avec une fonction de forme lagrangienne pour faire de l'élasticité fait l'objet par exemple d'une classe particulière. On remarque donc immédiatement que le comportement asymptotique de cette approche, si le nombre d'éléments finis, de fonctions de forme et de spécificités physiques tend vers N, est de provoquer l'apparition de N^3 classes. Alors que la séparation de ces composantes limiterait le nombre de classes nécessaires à 3N.

3.1.3 An adaptable finite element modelling kernel

Cet article traite exactement du même sujet que cette thèse : la modélisation d'un noyau de calcul par élément fini générique. Cette article se distingue des deux précédents par un aspect très important : la séparation de la physique et des mathématiques y est excellente et fait l'objet de deux hiérarchies de classes complètement indépendantes, ce qui semble être une caractéristique essentielle de tout moteur de calcul suffisamment générique. Plus important encore, la modélisation proposée dans cet article effectue un traitement spécial des degrés de liberté (ddl), les associant à des connecteurs, autrement dit, des parties de l'élément (nœud, arête, face, ou même tout l'élément) servant de support. Cette approche, complétée par le fait que, lors de l'assemblage, tous les ddls semblent être traités comme ce qu'ils sont, soit de simples valeurs scalaires auquelles le connecteur et surtout l'interpolation donnent un sens, induit de grands avantages au niveau de la facilité de traitement de fonctions de forme variées et pouvant avoir

n'importe quel type de support (pouvant reposer sur n'importe quel ensemble de connecteurs).

Autre élément très intéressant : les auteurs ont décidé de ne pas faire un moteur de calcul strictement orienté objet, mais un moteur de calcul *généraliste*. Ils ne cherchent donc tout naturellement pas à réprimer l'apparition de hiérarchies de classes uniquement algorithmiques. Il est par contre dommage de n'avoir vraiment retenu cette approche que pour les algorithmes de résolution et de ne pas l'avoir étendu au reste du noyau de calcul. Il n'y a pas d'exposition claire et complète de la hiérarchie de classe proposée dans cette article, aussi invitons-nous le lecteur intéressé à s'y reporter pour plus de détails.

3.1.4 Alexander Stepanov, la STL et la programmation générique

3.1.4.1 Pourquoi la programmation générique?

La programmation orientée objet est devenu de nos jours une connaissance de base indispensable à tout informaticien et son succès est dû à de réels apports à l'implémentation de la modélisation de systèmes complexes. Ce succès ne fut d'ailleurs possible que grâce au développement de méthodologies de COO telles qu'OMT. Cependant, si les méthodes orientées objet excellent dans la modélisation de données et de flux de données, elles pêchent dès qu'il s'agit de réaliser des logiciels scientifiques ou toute partie purement algorithmique d'un logiciel. Le terrain de la modélisation efficace des relations entre les données à traiter et les algorithmes sans utiliser de MCOO est cependant loin d'être inoccupé. Des travaux remarquables ont été menés en ce domaine par M. Alexander Stepanov. Une des réalisations les plus connues de M. Stepanov, que nous allons (trop) brièvement aborder afin d'expliciter les concepts clés de la généricité, est la Standart Template Library (STL).

3.1.4.2 La programmation générique par l'exemple: la STL

3.1.4.2.1 Présentation de la STL

La STL est un ensemble de structures de données et d'opérations sur ces structures de données. On distingue, par exemple, des structures de vecteurs, des listes chaînées, des maps, etc. Côté algorithmes, on distingue surtout des algorithmes de tri, comme le "binary search", l'algorithme de recherche binaire. Une conception purement objet d'une telle bibliothèque de classes aurait immanquablement donné une hiérarchie de classes, certainement à héritage multiple, et entraînant donc la pénalité de performance inhérente à la résolution des surcharges lorsque l'on utilise l'héritage multiple, réunissant dans une même classe les données et la surcharge des méthodes de l'interface pour cette classe.

L'approche utilisée dans la STL est toute autre: on sépare radicalement **données** et les **algorithmes** et on définit leurs interactions en termes d'**itérateurs**.

3.1.4.2.2 Itérateurs

On distingue plusieurs classes d'itérateurs: "forward iterator", "reverse iterator", "random iterator", etc. Pourquoi définir plusieurs types d'itérateurs? Supposons que l'on se contente de définir un unique type d'itérateur permettant l'accès aléatoire aux données (un "random iterator" donc). Cet itérateur conviendrait parfaitement aux utilisations potentielles d'une map ou encore d'un vecteur. Par contre, l'utilisation d'un random iterator sur une liste chaînée entraînerait des pénalités de performances proprement catastrophiques si l'on fait réellement des accès aléatoires dans cette liste chaînée, une liste chaînée étant intrinsèquement une structure séquentielle. C'est donc pour exprimer les limites de chaque type de structure de données qu'ont été créés différents types d'itérateurs. Une liste doublement chaînée comprendra donc, par exemple, un forward iterator (itérateur permettant de parcourir séquentiellement la liste du premier au dernier noeud) et un reverse iterator (itérateur permettant de parcourir séquantiellement la liste

du dernier au premier nœud) mais pas de random iterator. De manière similaire, un flux ne sera doté que d'un forward iterator.

Aborder les flux nous permet d'aborder une deuxième classification, orthogonale à la première, des itérateurs. Il est évident, pour le lecteur averti, qu'il existe plusieurs types de flux. Principalement des flux d'entrées, de sorties et d'entrées/sorties. Afin de modéliser correctement cette réalité, il est donc nécessaire d'introduire une classification selon que l'accès itératif aux éléments d'une structure de données sera en lecture seule, en écriture seule ou en lecture écriture.

Ces deux classifications orthogonales sont définies en termes de spécification d'interfaces. Cependant, le respect de ces interfaces ne se fait pas par héritage d'une classe de base virtuelle pure mais par implémentation de toutes les méthodes définies dans chacune des classifications auxquelles appartient l'itérateur. Ceci afin d'éviter la pénalité, minime dans le cas de l'héritage simple, importante dans le cas de l'héritage multiple, due à la résolution de la version de la méthode virtuelle à utiliser.

3.1.4.3 Synthèse POO - Programmation Générique

Avant de conclure sur la programmation générique et sur la manière de la combiner à l'approche objet pour retirer le meilleur de ces paradigmes, il est intéressant d'étudier l'extrait d'une interview d'Alexander Stepanov, retranscrit ci-dessous :

Question: *I think STL and Generic Programming mark a definite departure from the common C++ programming style, which I find is almost completely derived from SmallTalk. Do you agree?*

Réponse: Yes. STL is not object oriented. I think that object orientedness is almost as much of a hoax as Artificial Intelligence. I have yet to see an interesting piece of code that comes from these OO people. In a sense, I am unfair to AI: I learned a lot of stuff from the MIT AI Lab crowd, they have done some really fundamental work: Bill Gosper's Hakmem is one of the best things for a programmer to read. AI might not have had a serious foundation, but it produced Gosper and Stallman (Emacs), Moses (Macsyma) and Sussman (Scheme, together with Guy Steele). I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras - families of interfaces that span multiple types. I find OOP philosophically unsound. It claims that everything is an object. Even if it is true it is not very interesting - saying that everything is an object is saying nothing at all. I find OOP methodologically wrong. It starts with classes. It is as if mathematicians would start with axioms. You do not start with axioms - you start with proofs. Only when you have found a bunch of related proofs, can you come up with axioms. You end with axioms. The same thing is true in programming: you have to start with interesting algorithms. Only when you understand them well, can you come up with an interface that will let them work.

Question: *Can I summarize your thinking as "find the [generic] data structure inside an algorithm" instead of "find the [virtual] algorithms inside an object"?*

Réponse: Yes. Always start with algorithms.

-

Ces paroles peuvent sembler hérétiques aux tenants du dogme orienté objet et leur auteur aurait sûrement été brûlé vif en place publique ou aurait dû les abjurer (comme Galilée) pour avoir osé les proférer s'il n'avait pas apporté la preuve de ses affirmations en proposant la STL. La STL possède en effet la même efficacité que le code assembleur équivalant. Quel logiciel développé avec une approche purement orientée objet peut se

targuer d'une telle efficacité en ne brimant pas ses capacités d'extension? Bref, pour ce qui est de l'écriture de logiciels avant tout algorithmiques, la preuve est faite que d'autres paradigmes que le paradigme purement objet doivent être utilisés pour obtenir un résultat satisfaisant, tant sur le plan de la généralité que sur le plan de l'efficacité.

Le logiciel développé durant cette thèse, FEMView, est essentiellement algorithmique en ce qu'il ne doit pas répondre à des sollicitations de l'usager et de son environnement autres que ses paramètres initiaux. L'approche générique a été largement utilisée dans ce développement en ce que l'on a systématiquement séparé les algorithmes des structures de données. On a, par exemple, éclaté l'élément fini en un support géométrique et des espaces fonctionnels de fonctions de forme, tests et forme géométrique. On définit le calcul local sur chacun des éléments de la décomposition en sous-domaines comme une itération sur chacun de ces sous-domaines. On définit également l'intégration numérique sur chacun de ces sous-domaines comme une itération sur les points d'intégration de cet élément. Cependant, certaines fonctionnalités offertes par la programmation orientée objet ont été conservées parce qu'elles complétaient la programmation générique ou rendaient plus simple la modélisation et l'implémentation de certains concepts. On pense notamment à la définition de hiérarchies de classes homomorphes ainsi qu'aux interfaces de « double dispatch ». Ces deux notions sont présentées dans la partie « Notions avancées de C++ » du présent document.

Chapitre 4 SYNTHÈSE DE L'ÉTUDE PRÉLIMINAIRE ET DE L'ÉTUDE BIBLIOGRAPHIQUE

L'étude préliminaire et l'étude bibliographique font toutes deux ressortir le besoin d'une autre approche pour modéliser et implémenter un moteur de calcul par éléments finis. On voit clairement dans les incohérences et la séparation incomplète des différentes parties les limites de l'approche purement orientée objet dès lors que l'on s'attelle à la modélisation d'un logiciel essentiellement algorithmique. Ce n'est pas étonnant : l'approche orientée objet a été conçue pour créer des systèmes d'informations, pas pour exprimer efficacement des algorithmes. Il est frappant de constater que l'exemple typique des cours de modélisation objet est un distributeur bancaire, exemple où l'on voit effectivement apparaître différentes entités et où l'approche objet trouve sa pleine justification.

Les éléments finis font eux aussi apparaître plusieurs entités, mais une différence fondamentale existe entre les objets d'un distributeur bancaire et les objets d'un moteur de calcul par éléments finis : les objets du moteur de calcul par éléments finis sont, pour une large partie, essentiellement algorithmiques. Ils ne sont pas caractérisés par un état propre, mais par l'effet qu'ils produisent sur d'autres objets. Les fonctions de forme fournissent un très bon exemple pour illustrer cette affirmation : elles sont uniquement algorithmiques (calcul de fonctions) et leur évaluation produit un effet sur la valeur de la matrice élémentaire correspondant à l'intégration locale d'un terme sur un élément.

D'autres composantes sont essentiellement algorithmiques, comme les intégrateurs par exemple. Il semble donc essentiel d'utiliser une approche prenant en compte d'une manière appropriée cette spécificité d'un moteur de calcul par éléments finis et, plus généralement, de tout logiciel avant tout algorithmique.

La programmation générique, tant prisée par Alexander Stepanov et dont l'approche est validée par la réussite de la STL, est une piste intéressante, à explorer. Elle permet en effet par une utilisation raisonnée des spécificités du langage d'implémentation, d'obtenir une efficacité optimale des algorithmes proposés. Il faut par contre se garder de tomber dans l'excès inverse, consistant à rejeter en bloc l'approche orientée objet et à ne vouloir de nouveau se limiter qu'à un seul paradigme. La STL illustre notre propos. Elle est le résultat de l'application stricte des principes de la programmation générique et sépare complètement et systématiquement algorithmes et données. Le mécanisme d'itérateur utilisé par la STL pour régir les interactions entre les algorithmes et les données est suffisant pour les cas simples que traite la STL, comme le tri « quicksort » par exemple. Il est par contre insuffisant pour évaluer une fonction de forme sur un élément quelconque : il faut alors transmettre bien plus d'information que de simples considérations sur la comparaison entre deux éléments : il faut savoir avec exactitude quelles sont les natures de l'élément géométrique et de la fonction de forme afin d'évaluer correctement cette dernière. On voit donc qu'il est nécessaire d'adopter une approche combinant les avantages de différents paradigmes pour espérer réussir la synthèse d'un moteur de calcul par éléments finis compact et généraliste : une approche dite « multi-paradigmes ».

Chapitre 5 APPROCHE MULTI-PARADIGMES

5.1 Introduction

L'approche multi-paradigmes consiste principalement à reconnaître les « familles » d'un « domaine d'application » donné, tout en conservant une description très abstraite et en ne faisant aucune hypothèse sur le paradigme d'implémentation. L'approche multi-paradigmes, comme toutes les techniques permettant à l'homme de mieux cerner un domaine est donc fondée sur une abstraction du problème. Lorsque l'on effectue cette analyse, on commence par reconnaître un « domaine ». Ce domaine peut-être, par exemple, la gestion de transactions bancaires ou encore l'automobile. Une fois le domaine identifié, on reconnaît les différentes composantes de ce domaines, les différentes « choses » le composant. On insiste ici particulièrement sur l'utilisation du mot « chose » pour bien souligner que l'on ne cherche pas à repérer des objets, mais des éléments du domaine. Ces éléments peuvent donc être des objets, mais peuvent également être autre chose, comme la simple définition d'un comportement commun à un ensemble d'algorithmes. Prenons l'exemple d'un tri. Le tri peut être une composante du domaine. Afin de pouvoir l'implémenter efficacement, il est nécessaire de tenir compte de tous les paramètres entrant en considération non seulement lors de la définition de l'architecture logicielle, mais également dans son implémentation. On peut par exemple reconnaître à l'étape de modélisation certain comportements usuels (« *patterns* ») et, à l'étape d'implémentation, décider d'utiliser le mécanisme de classe de type patron (« *template* ») que propose le C++. Une fois tout cela défini (cela va du mode d'interaction entre l'algorithme et les données à trier jusqu'aux stockages de ces

données), on peut reconnaître les algorithmes de tri respectant les contraintes de ce cadre abstrait. On peut par exemple imaginer que le tri rapide « *quicksort* » soit une possibilité et que le tri en monceau « *heap sort* » en soit une autre. En résumé, pour utiliser la terminologie définie par James O. Coplien dans « Multiple-Paradigm Design for C++ » [1], après avoir abstrait au maximum les éléments du « domaine » pour reconnaître des « communautés » suffisamment fortes entre des entités qui sont alors regroupées en « famille », on s'intéresse aux « variations » entre les membres de ces familles afin d'achever de les caractériser.

L'approche multi-paradigmes décompose précisément toutes les étapes de la création d'un programme, de sa spécification jusqu'à son implémentation. Cependant, le respect de ces étapes n'apportant pas de clarté à la présentation notre étude, on préfère, après avoir présenté plus en détails les concepts de domaine, de famille, de communauté et de variabilité, s'en tenir à une définition du domaine et à une présentation complète de l'analyse points communs/variations des différentes familles composant un moteur de calcul généraliste par éléments finis.

5.2 Domaine

Un domaine représente une zone d'intérêt. C'est la zone définie par l'utilisateur du logiciel qui devient le plus souvent le domaine du programme. Dans la plupart des analyses, le domaine est séparé en sous-domaines. On adopte ainsi une approche « diviser pour régner » permettant de simplifier la conception. Cependant, certains domaines se prêtant mal à cette décomposition, celle-ci n'a rien d'obligatoire.

5.3 Famille et communautés

Comme on l'a expliqué dans l'introduction, lorsque l'on identifie les familles, on met l'emphase sur l'abstraction, sur les points communs entre éléments, en omettant

volontairement les détails. Une telle abstraction demande donc une bonne connaissance du domaine d'application et de la manière dont varient les éléments. Prenons l'exemple de voitures. Il est aisé pour l'humain de reconnaître une voiture dans la rue, bien qu'il existe une grande diversité de modèles. On reconnaît donc les points communs, les communautés entre les différents modèles (essentiellement : quatres roues, un habitacle, un coffre, un moteur ,un volant, etc.) en faisant abstraction des variations (couleur, formes des différentes composantes, etc.).

Tous les domaines ne comprennent pas de familles. Nous ne nous attarderons cependant pas sur ces cas particuliers, assez marginaux au demeurant.

5.4 Variabilité

L'essentiel des variations est généralement constitué d'ajouts à ce qui a été reconnu comme étant commun à tous les membres d'une famille. On peut citer comme exemple de variation positive, toujours dans le cas de la famille voiture, l'ajout d'un aileron à l'arrière de la voiture. Il existe également des cas, plus rares, de variabilité (ou variation) négative, c'est à dire de cas où la variation s'exprime par la soustraction d'une caractéristique commune à tous les autres membres de la famille. Prenons l'exemple de formes géométriques. On peut imaginer qu'une des caractéristiques communes à toutes les formes soit un nombre de côtés. Dans le cas d'un cercle cependant, ce nombre de côtés est sans signification, et est une variation négative.

La bonne gestion des variabilités est, comme on le voit, un des aspects principaux d'un bon design. Il est essentiel de considérer dès le départ les mécanismes utilisés pour permettre ces variabilités, pouvant aller de la simple définition de nouvelles classes à la spécification dynamique de structures de données durant l'exécution du programme. On oppose ici une liaison statique des variabilités (tout est fixé lors de l'écriture du programme) à une liaison dynamique (prise en compte des variabilités lors de

l'exécution). La souplesse et l'efficacité de ces mécanismes sont importantes et, malheureusement, l'obtention d'une grande souplesse s'obtient généralement au prix d'une importante pénalité de performance et réciproquement. Prenons l'exemple d'un algorithme parfaitement connu et maîtrisé. On peut décider de l'écrire directement en assembleur afin d'obtenir une efficacité maximale. On interdit par contre alors toute modification dynamique de cet algorithme. À l'opposé, définir entièrement dynamiquement un algorithme est certes très souple, mais pénalise la performance de l'application.

Lors de l'application de l'analyse communautés/variations des familles de la méthode des éléments finis, on mettra systématiquement en tête de la section correspondant à chaque famille un tableau résumant les variations et les points communs entre ses membres. La plupart des variations étant positives, on considérera toutes les variations comme l'étant par défaut. Les variations négatives seront signalées au moyen d'une signe (-).

Chapitre 6 DESCRIPTION MATHÉMATIQUE DE LA MÉTHODE DES ÉLÉMENTS FINIS

6.1 Interpolation

Dans toute méthode numérique, il est nécessaire d'interpoler des champs en utilisant des espaces fonctionnels de dimension finie, l'ordinateur ayant par définition une capacité mémoire limitée. Soit un champ scalaire quelconque v de V un espace fonctionnel de dimension finie défini sur un espace Ω. Soit une base $B=(v_1, \ldots, v_n)$ de V où les v_i sont les fonctions de base (on peut toujours trouver une base aux espaces fonctionnels de dimension finie). Par définition d'une base, un champ v de V peut s'écrire de façon univoque comme :

$$v = \sum v_i c_i .$$

Les quantités ci sont uniques. Le vecteur colonne $[\mathbf{c}]_B$ est le vecteur des coordonnées de v en base B. Par exemple, l'espace fonctionnel formé des fonctions $B = (1, x, x^2, x^3)$ sur l'intervalle [0,1] est une base d'un espace de dimension 4. C'est l'espace des polynômes de degré inférieur à 4 pour la variable x de $\Re$. Une fonction quelconque de cet espace s'écrit :

$$v(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

On peut interpréter les coefficients c_i de la façon suivante :

$$c_i = \frac{1}{i!}\frac{d^i v}{dx^i}\bigg|_{x=0}, \; i = 0,...,3$$

Donc, en spécifiant toutes les dérivées de v à l'origine jusqu'à l'ordre 3, on caractérise complètement la fonction c'est-à-dire qu'on choisit un élément particulier de l'espace fonctionnel. En choisissant une autre base pour le même espace, on facilite l'interprétation des coefficients c_i. Par exemple, la base des polynômes de Lagrange de degré 4 :

$$B = \left(\frac{\prod_{i \neq j} (x - x_j)}{\prod_{i \neq j} (x_i - x_j)}, \; i = 0,...,3 \right)$$

confère une interprétation particulière aux coefficients c_i :

$$c_i = v(x_i) \;\; i = 0,...,3$$

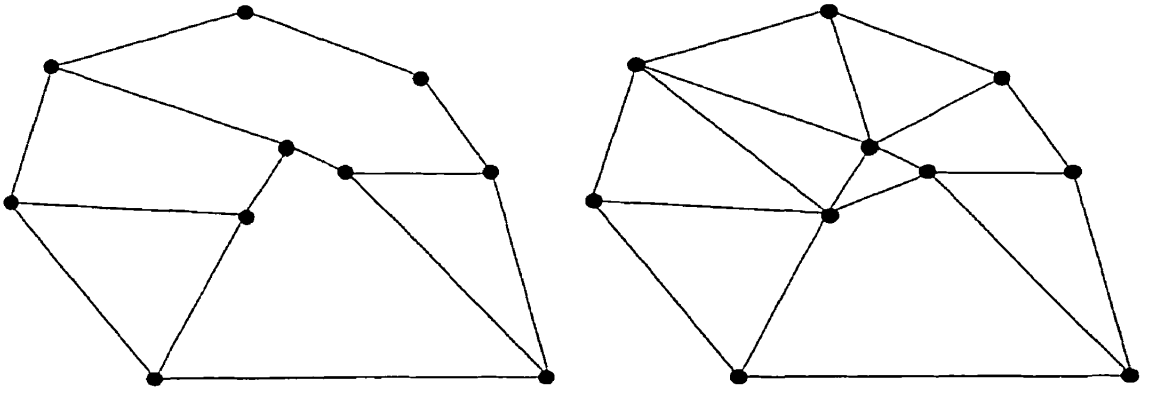
L'interprétation des c_i peut s'avérer importante dans le contexte des méthodes numériques même si on peut toujours s'en passer. Contradictoire ? Pas tellement. La plupart des ouvrages sur les éléments finis parlent d'une méthode utilisant un type bien particulier de fonctions de base dont les valeurs des coefficients c_i possèdent une interprétation simple. Cela permet de simplifier certaines procédures, notamment l'imposition de conditions aux frontières. Dans le cas d'une interpolation de Lagrange, fixer une valeur $v=v_i$ en un x_i revient simplement à imposer que $c_i=v_i$.

6.2 Maillage

Imaginons maintenant que l'espace Ω soit divisé en entités géométriques e_i appelés éléments géométriques. Pour que cette division soit un maillage, il faut qu'elle respecte les conditions suivantes :

- $\Omega = \bigcup_{i=1}^{N} e_i$,
- L'intersection de deux éléments est de mesure nulle.

Par exemple, la figure suivante montre deux maillages.



Maillage 1 *Maillage 2 au sens des éléments finis*

Figure 6.1 - Maillage au sens des éléments finis

Un maillage au sens des éléments finis est composé de formes géométriques simples (lignes, triangles, quadrangles, tétraèdres, prismes ou hexaèdres) dont les intersections ne peuvent se produire qu'en un sommet, une arête ou une face. Le maillage 1 n'est pas

un maillage au sens des éléments finis « classiques ». Le maillage 2 est un exemple de maillage utilisable pour une approximation au sens des éléments finis (voir figure ci-dessus).

6.3 Connecteurs

On a indiqué plus haut comment interpoler des champs dans des espaces fonctionnels de dimension finie. Si nous possédons un maillage, celui-ci peut servir de support à notre interpolation. Montrons tout d'abord comment interpoler linéairement un champ sur un triangle :

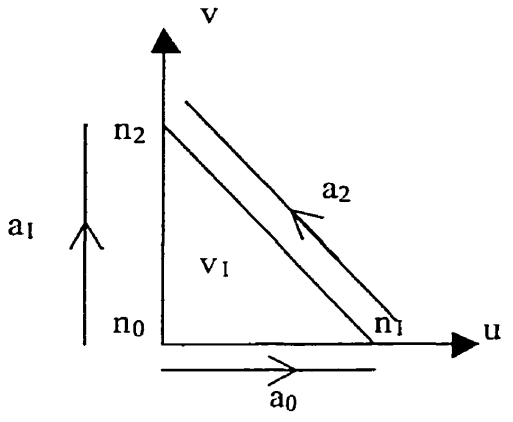


Figure 6.2 – Triangle de référence

Une base de l'espace des fonctions de u et v linéaires sur le triangle est $B_1 = (1,u,v)$. Une fonction v quelconque de cet espace peut s'écrire $v = c_0 + c_1 u + c_2 v$. L'interprétation des c_i est la suivante :

$$c_0 = v(n_0)\,;\, c_1 = v(n_1) - v(n_0)\,;\, c_2 = v(n_2) - v(n_0)$$

Il est plus simple de redéfinir la base de la façon suivante : B_1 = (1-u-v,u,v) ce qui permet une interprétation plus simple des c_i :

$$c_0 = v(n_0)\,;\, c_1 = v(n_1)\,;\, c_2 = v(n_2)$$

Ce type de fonction de base est usuellement appelé fonction de base nodale car la signification des coefficients c_i est simplement la valeur du champ au nœud i. Le nœud i est appelé **connecteur relatif à la fonction de base c_i** dans le sens où la fonction de base est non nulle au connecteur c_i et nulle aux autres connecteurs. On voit aussi que le connecteur est commun aux éléments connexes au nœud i et que le support de la fonction de base est simplement l'ensemble des éléments connexes au nœud i. Pour les fonctions de base du premier ordre tels que nous venons de les décrire, la fonction de base f_i relative au nœud i est la fonction « chapeau » :

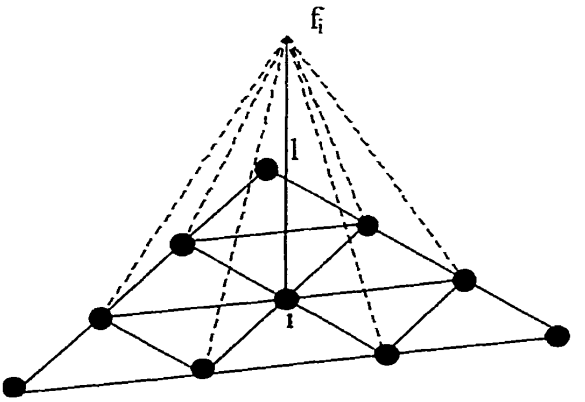


Figure 6.3 - Fonction chapeau

Dans un maillage composé de N nœuds, un champ quelconque v(x,y) s'interpole comme suit :

$$v(x,y) = \sum_{i=1}^{N} v(n_i)\, f_i(x,y)$$

Notons que les nœuds ne sont pas les uniques connecteurs possibles. Les arêtes peuvent elles aussi connecter. Si par exemple on désire compléter la base des fonctions du premier ordre jusqu'au second ordre, on peut écrire :

$$B_2 = B_1 \oplus B_{12}$$

avec $\oplus$ la somme directe et :

$$B_{12} = \left(\frac{(1-u-v)\,u}{4}, \frac{(1-u-v)\,v}{4}, \frac{uv}{4} \right)$$

Un champ v s'interpole maintenant comme suit :

$$v(x,y) = \sum_{i=1}^{3} v(n_i)\, f_i(x,y) + \sum_{i=4}^{6} v(\tfrac{1}{2} a_i)\, f_i(x,y)$$

où les $v(\frac{1}{2} a_i)$ sont les valeurs du champ au milieu de l'arête a_i. Les arêtes sont manifestement des connecteurs de cet élément. Notons que ce type de construction des fonctions de base peut se généraliser: on peut compléter la base B_2 pour obtenir une base pour les polynômes d'ordre 3 avec $B_3 = B_2 \oplus B_{23}$. On crée ainsi une suite d'espaces de fonctions de base dite **hiérarchique**. On verra plus tard l'avantage de ce type d'éléments sur des éléments classiques.

Il n'est pas obligatoire de définir des fonctions de bases continues. Les champs physiques ne sont pas tous continus intrinsèquement, c'est loin d'être le cas. Et même si c'était vrai, il est parfois utile de discrétiser un champ de façon discontinue.

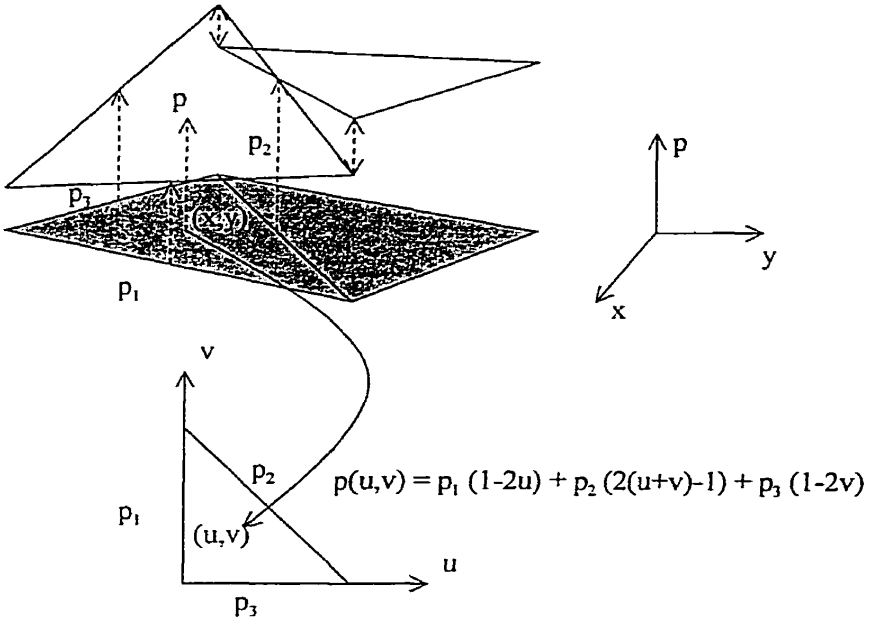


Figure 6.4 - Éléments finis semi-discontinus

Par exemple, ces fonctions de forme discontinues peuvent être choisies comme étant égales à 1 sur la face à laquelle elles sont attachées (connecteur de face) et égales à –1 au nœud qui n'appartient pas à la face. L'interpolation telle que décrite est discontinue (sauf aux centres des faces) comme on le voit sur la figure précédente. Notons que l'espace fonctionnel couvert sur un élément est strictement le même pour les fonctions continues et discontinues. Par contre, les fonctions discontinues constituent un espace plus riche sur un maillage. Pour s'en convaincre, il suffit de se rendre compte que le nombre de faces d'un maillage est beaucoup plus élevé que le nombre de nœuds. Il est en fait plus coûteux d'interpoler une fonction discontinue qu'une fonction continue.

Une autre question peut se poser quand on désire interpoler des champs sur un maillage : comment interpoler des champs vectoriels **v**? On peut se servir de 3 espaces fonctionnels

discrets pour interpoler des champs à trois composantes. Pour un champ **v** interpolé aux nœuds, on écrit :

$$\mathbf{v}(x,y) = \sum_{i=1}^{N} \mathbf{v}(n_i)\, f_i(x,y)$$

avec :

$$\mathbf{v}(n_i) = \left[v_x(n_i),\, v_y(n_i), v_z(n_i)\right]$$

Un vecteur construit à partir de trois composantes scalaires continues (c'est-à-dire avec des fonctions de forme f_i continues) par exemple aura toutes ses composantes continues. La continuité d'un champ de vecteurs se définit de façon différente de celle d'un champ scalaire. Il est en effet peu vraisemblable que la composante x d'un vecteur soit continue alors que les autres composantes ne le soient pas. Par contre, il existe des champs de vecteurs dont uniquement la composante tangentielle est continue, d'autres où c'est la composante normale qui l'est. La sous-section sur les formes différentielles nous renseignera sur la réelle nature de ces vecteurs. En ce qui concerne l'interpolation de tels champs, il paraît naturel de choisir les arêtes du maillage comme connecteur pour les champs à composantes tangentielles continues et les faces pour ceux à composante normale continue. La façon adéquate d'interpoler un champ de vecteurs dont certaines composantes uniquement sont continues est d'utiliser un autre type d'interpolation. Au lieu de construire à partir d'un espace de fonctions scalaires V un espace vectoriel V^3, on peut définir un espace de fonctions à valeurs vectorielles comme base à l'interpolation de nos vecteurs. Si on reprend l'exemple précédent des cubiques sur [0,1], l'ensemble des fonctions vectorielles (2 composantes pour rester simple) à composantes cubiques sur [0,1] a comme base : $B = ([1,0], [x,0], [x^2,0], [x^3,0], [0,1], [0,x], [0,x^2], [0,x^3])$. N'importe quelle fonction à valeur vectorielle cubique s'écrit comme une combinaison linéaire des 8 vecteurs de base. Si $\mathbf{e}_x$ et $\mathbf{e}_y$ sont les vecteurs de base du plan dans lequel sont définies les fonctions, la base B s'écrit $B = ((1,x,x^2,x^3)\mathbf{e}_x, (1,x,x^2,x^3)\mathbf{e}_y)$. On peut

interpoler le champ composante par composante à partir d'espaces fonctionnels scalaires. Si par exemple on choisissait comme base $B = ([1,0],[x,x],[x^2, x^2],[x^3, x^3])$, la notion même de composante du vecteur est inutilisable car ces composantes sont liées. L'espace est de dimension 4 et il faut 4 connecteurs pour interpoler un vecteur, ces connecteurs n'ayant pas la signification d'une composante.

Reprenons l'exemple du triangle, on peut choisir une base vectorielle pour les fonctions à interpoler telle que la composante tangentielle du vecteur construit soit continue aux interfaces entre les éléments. Ces fonctions de base appelées « fonctions de base d'arêtes de Whitney » permettent d'interpoler judicieusement des champs dont la composante tangentielle est continue. On a :

$$B = \begin{pmatrix} [(1-u-v)\text{grad}(u) - u\ \text{grad}(1-u-v)], \\ [(1-u-v)\text{grad}(v) - v\ \text{grad}(1-u-v)], \\ [u\ \text{grad}(v) - v\ \text{grad}(u)] \end{pmatrix} = (\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3).$$

$$\mathbf{v}(u,v) = \sum_{i=1}^{N} v(a_i)\, \mathbf{f}_i(u,v).$$

Notons qu'il est possible de prouver que la circualtion de $\mathbf{f}_i$ sur a_j notée $< a_j, \mathbf{f}_i > = \delta_{ij}$.

6.4 Changement de coordonnées, fonctions de forme géométriques

On a défini des fonctions de base dans un système de coordonnées (u,v) dit de référence. Les éléments géométriques d'un maillage sont représentés dans un espace réel (x,y) et sont de forme quelconque. On se doit donc de trouver une application qui permet de passer du système de coordonnées (u,v) au système (x,y), un changement de coordonnées :

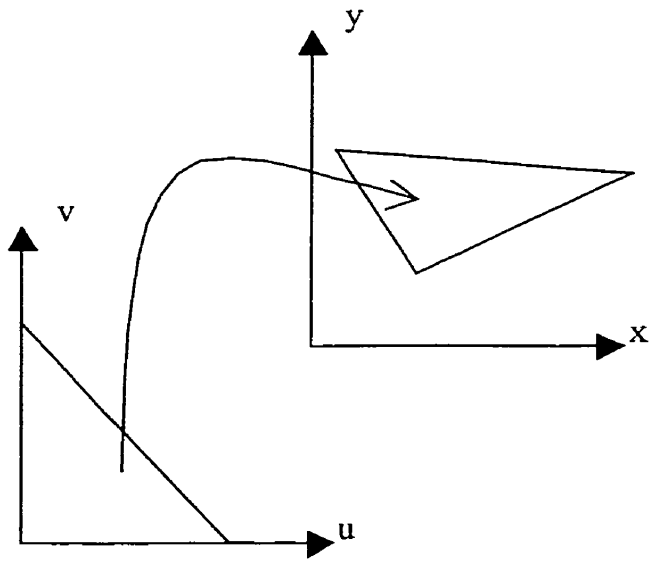


Figure 6.5 – Changement de coordonnées

On utilise habituellement les fonctions linéaires pour définir l'application qui fait passer $(u,v) \mapsto (x,y)$. Par exemple, pour le triangle à trois nœuds, on peut calculer les coordonnées d'un point quelconque (x,y) dans l'élément de référence comme suit :

$$x(u,v) = \sum_{i=1}^{3} x(n_i)\, f_i(u,v) = x(n_1)(1.-u-v)+x(n_2)\,u+x(n_3)\,v$$

$$y(u,v) = \sum_{i=1}^{3} y(n_i)\, f_i(u,v) = y(n_1)(1.-u-v)+y(n_2)\,u+y(n_3)\,v$$

exactement comme si on interpolait un champ. Il est nécessaire que les fonctions f_i soient continues il n'y a pas de déchirures dans le domaine Ω. On appelle ce type de fonctions « **fonctions de forme géométrique** ».

Soit $\{x^1_i\}, \ldots, \{x^N_i\}$ N systèmes de coordonnées successifs qui font passer des coordonnées $\{x^N_i\}$ de l'élément de référence aux coordonnées $\{x^1_i\}$ de l'élément réel

dans ses axes euclidiens. La matrice jacobienne de la transformation de coordonnées de $\{x^k_i\}$ vers $\{x^{k-1}_i\}$ est définie comme suit :

$$\left(\mathbf{J}^{k-1,k}\right)_{ij} = \frac{\partial x^k_j}{\partial x^{k-1}_i} ,$$

Cette définition montre que la matrice jacobienne est associée à deux systèmes de coordonnées successifs et n'est donc pas une caractéristique d'un seul système de coordonnées. Pour obtenir une notion intrinsèque à un seul système de coordonnées, on introduit le tenseur métrique $(g^k)_{ij}$:

$$\left(\mathbf{g}^k\right)_{ij} = \left(\left(\mathbf{J}^{k-1,k}\right)^T \mathbf{g}^{k-1} \mathbf{J}^{k-1,k}\right) , \left(\mathbf{g}^1\right)_{ij} = \delta_{ij}$$

Il apparaîtra plus tard que les équations éléments finis peuvent être exprimées de telle sorte que le tenseur métrique $(g^N)_{ij}$ apparaisse explicitement.

6.5 Formes différentielles

Les résultats de l'évalution des fonctions de forme sont considérées comme des p-formes différentielles, p = 0,1,2,3. Sans entrer trop profondément dans la théorie des formes différentielles, énonçons maintenant quelques concepts relatifs aux formes.

Les 0-formes sont des champs scalaires destinés à être évalués en un point. Ce sont des champs scalaires au sens classique, définis ponctuellement. Ils sont invariants si, à topologie égale, on applique une déformation continue à l'espace. Le « gradient » d'une 0-forme est défini. Une 0-forme est donc une fonction continue.

Les 1-formes ont une représentation vectorielle. Ce sont, par exemple, des circulations sur des arêtes. Leur évaluation a un sens sur une courbe (1-chaîne). Leur intégrale

curviligne est invariante si on déforme continûment l'espace. Plutôt que le symbole d'intégration, notons l'intégrale curviligne de la 1-forme α sur la courbe C par $<C,\alpha>$. Le rotationel d'une 1-forme est défini. On peut montrer que les 1-formes existent dans des espaces où leur composante tangentielle est continue. Le gradient d'une 0-forme est une 1-forme.

Les 2-formes ont une représentation vectorielle, ce sont, par exemple, des flux à travers une surface. Leur évaluation a un sens sur une surface (2-chaîne). Le flux $<S,\beta>$ d'une 2-forme β à travers une surface S est invariant si on déforme continûment l'espace. La divergence d'une 2-forme est définie. On peut montrer que les 2-formes existent dans des espaces où leur composante normale est continue. Le rotationnel d'une 1-forme est une 2-forme.

Les 3-formes ont une représentation scalaire. Ce sont, par exemple, des densités volumiques. Leur évaluation a un sens sur un volume (3-chaîne). L'intégrale de volume $<V,\gamma>$ d'une 3-forme γ sur le volume V est invariante si on déforme continûment l'espace. On peut montrer que les 2-formes existent dans des espaces « d'énergie finie ». La divergence d'une 2-forme est une 3-forme.

Un seul opérateur est défini pour les formes : la dérivée extérieure d. La multiplication d'une p-forme par une q-forme donne une (p+q)-forme. Dans le graphique suivant, on montre les relations existant entre les formes :

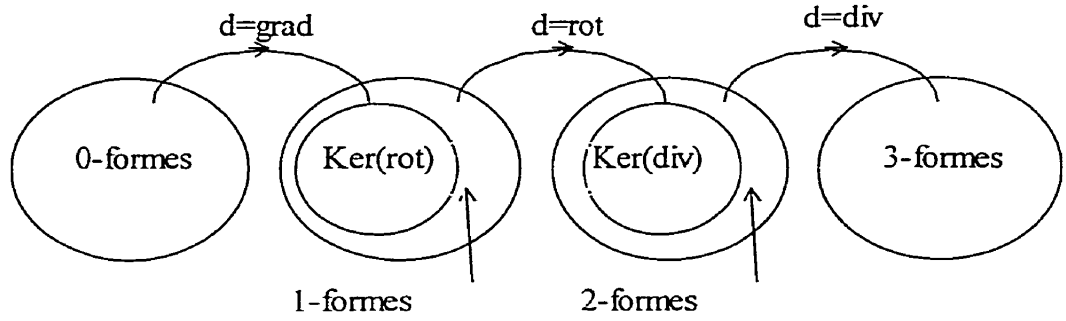


Figure 6.6 – Relations entre formes différentielles

On note que l'on a toujours d(dw) = 0. Deux applications successives de la dérivée extérieure donnent zéro. D'un autre côté, on peut définir un opérateur de bord ∂ qui associe à chaque domaine D de dimension p un domaine de dimension p-1 noté ∂D. (Soit une paramétrisation de D définie par p coordonnées locales. Le bord est l'ensemble des points dont tout voisinage dans l'espace des paramètres contient des points appartenant au domaine D et des points n'appartenant pas au domaine D). Le théorème de Stokes peut s'écrire $<D,d\omega>=<\partial D,\omega>$. Avec cette notation de produit de dualité, la dérivée extérieure apparaît comme l'**opérateur adjoint** du bord, c'est pourquoi en topologie on l'appelle également **cobord**. La dualité entre formes et domaines géométriques (chaînes) est purement topologique, elle correspond à la dualité entre **homologie** et **cohomologie**. Notons un exemple clair de la dualité entre forme et chaînes : de même qu'on a toujours d(dw) = 0, on a aussi toujours $\partial(\partial D) = 0$ c'est-à dire que l'opérateur frontière appliqué deux fois à une p-chaîne donne 0.

Allons plus profondément dans la compréhension des formes différentielles. Une 1-forme est une forme linéaire. Soit un espace euclidien à trois dimensions avec les vecteurs directeurs $(\mathbf{e}_x,\mathbf{e}_y,\mathbf{e}_z)$ définissant la base canonique habituelle. Elle s'écrit de la façon suivante :

$$w = f_x(x,y,z)\overline{dx} + f_y(x,y,z)\overline{dy} + f_z(x,y,z)\overline{dz}$$

où $\overline{dx}, \overline{dy}$ et $\overline{dz}$ sont des formes linéaires telles que :

$$\overline{dx}(\mathbf{e}_x) = 1, \overline{dx}(\mathbf{e}_y) = \overline{dx}(\mathbf{e}_z) = 0, \overline{dy}(\mathbf{e}_x) = \overline{dy}(\mathbf{e}_z) = 0, \overline{dy}(\mathbf{e}_y) = 1, \ldots$$

Ces formes linéaires définissent en fait l'étalon de comparaison dans notre espace de référence. Il ne faut pas confondre $\overline{dx}$ avec la différentielle dx, c'est d'ailleurs pour cela que nous avons légèrement modifié la notation habituelle propre à Cartan, l'inventeur génial des formes différentielles.

6.6 Formulation faible

On a défini des espaces fonctionnels car c'est dans ces espaces que sont situés les champs physiques que nous voulons calculer comme solution d'équations aux dérivées partielles. Résumons cela de façon assez abstraite. Soit H un espace de Hilbert sur $\Re$ et a une forme bilinéaire :

$$a : H \times H \to \Re$$

et soit une forme linéaire b appartenant au dual de H noté H' et à valeurs dans $\Re$:

$$b : H \to \Re$$

Les problèmes qu'on cherche à résoudre sont tous de la forme suivante. Soit $u \in H$ le champ inconnu. Il faut trouver u dans H qui vérifie :

$$a(u,v) = \langle b, v \rangle \quad \text{pour tout } v \in H.$$

La notation <.,.> est classique pour désigner la dualité entre H et H'. On aurait pu écrire b(v) à la place de <b,v>.

Les espaces fonctionnels que nous utilisons dans le cadre du calcul numérique sont des espaces de dimension finie (un nombre fini de paramètres). Ils sont entièrement caractérisés grâce au maillage du domaine (support à l'interpolation) et aux fonctions de forme. Par exemple, si on désire résoudre l'équation de Poisson ($\Delta u=f$ dans Ω avec $u=0$ sur Γ), on a :

$$a(u,v) = \int_{\Omega} \text{grad}^{t}\, u \,\text{grad}\, v \, d\Omega \;\; et \;\; b(v) = \int_{\Omega} f \, v d\Omega$$

De façon plus générale, en utilisant les formes différentielles, on peut écrire une forme bilinéaire comme suit :

$$a(u,v) = \int_{\Omega} du \wedge *dv \;\; et \;\; b(v) = \int_{\Omega} f \wedge dv$$

où les opérateurs d sont soit l'opérateur identité, soit l'opérateur de dérivée extérieure (gradient, rotationel ou divergence) et où * désigne l'opérateur de Hodge (loi de comportement). Notons que $du \wedge *dv$ doit être une 3-forme.

Le champ inconnu étant écrit comme $u = \sum_{i=1}^{m} u_i \; ff_i$, où u_i est la valeur du coefficient lié au connecteur i et ff_i la fonction de forme associée. On a donc :

$$a(u,v) = \sum_{i} u_i \int_{\Omega} dff_i \wedge *dv$$

Les fonctions v sont appelées fonctions test ou fonctions d'essai. A chacune de ces fonctions test correspond une équation. Elles sont elles aussi choisies dans un espace fonctionnel discret ayant pour base $B = (ft_1, ft_2, \ldots, ft_n)$ pour obtenir un nombre fini n

d'équations. Les fonctions de forme désignent donc les inconnues et les fonctions test les équations. Le problème consiste donc à trouver $u = \sum u_i \; ff_i$ combinaison linéaire des $(ff_1,\ldots,ff_m)$ tel que :

$$\sum_i u_i \int_\Omega dff_i \wedge {}^*dft_j = \int_\Omega f \wedge dft_j \qquad pour\ j = 1,\ldots,n$$

Le problème continu se ramène donc à un problème discret de résolution de n équations à m inconnues. Il semble donc judicieux de choisir n = m c'est à dire que l'espace des fonctions de forme et celui des fonctions test soient de même taille.

6.7 Formulations locales

On n'a pas encore exploité ici le fait que notre domaine soit divisé en éléments. Les opérateurs agissant sur le domaine peuvent être scindés en opérateurs locaux agissant sur les éléments. Ceci permet en fait une systématisation des calculs : un domaine peut être de forme quelconque, un élément non. Un domaine quelconque est donné sous la forme d'une liste d'éléments est les opérateurs sont calculés sur des éléments. On définit donc ici le concept important de **matrice locale** ou **formulation locale**. Sur un élément quelconque, on définit deux applications locales :

$$A(i,j) = \int_E dff_i \wedge {}^*dft_j \; dE$$

$$B(j) = \int_E f \wedge {}^*dft_j \; dE$$

Les fonctions de forme sont définies dans l'espace de référence. On a donc une matrice élémentaire A et un vecteur élémentaire B. Ces deux opérateurs sont indépendants de la formulation employée, du champ physique utilisé. Aucune notion de connectivité n'est

nécessaire pour calculer ces opérateurs. On a besoin en fait d'un élément géométrique, d'une liste de fonctions de forme, d'une liste de fonctions test, d'une méthode d'intégration et d'une loi de comportement :

A(n × m) = FormeLinéaire(Élément géométrique,

m Fonctions de forme,

n Fonctions test,

Métrique,

Intégrateur,

Loi de comportement,

Dérivée sur les fonctions de forme ?,

Dérivée sur les fonctions test ?);

6.8 Lien entre local et global

Le lien entre une forme locale (matrice ou vecteur) et l'opérateur global (matrice du système et membre de droite) est fait à l'aide des connecteurs. La figure suivante résume la procédure d'intégration local puis d'assemblage dans le système global.

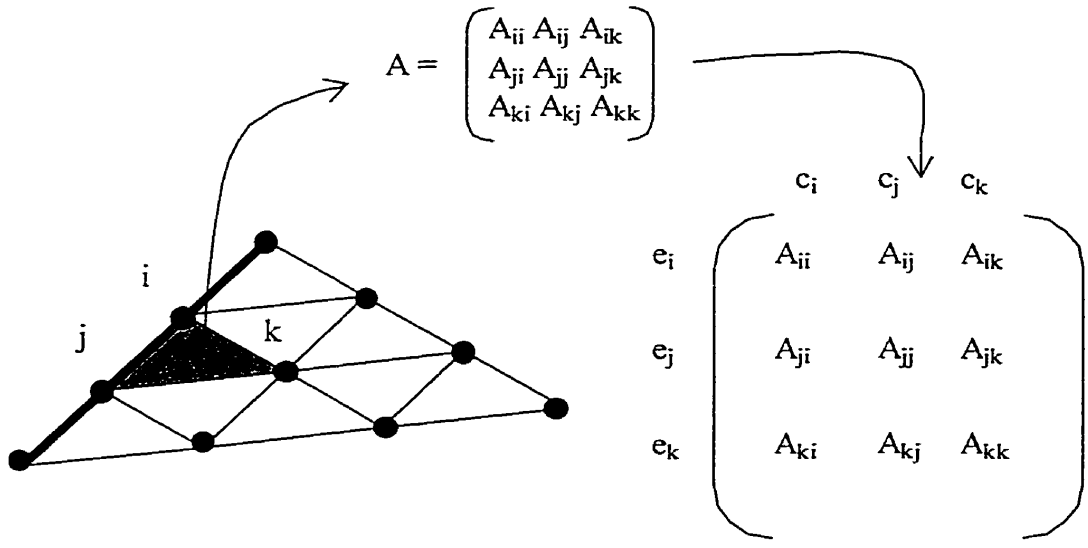


Figure 6.7 – Lien entre local et global

où les coefficients c_i sont les connecteurs relatifs aux fonctions de forme ff_i et où les indices e_j sont les numéros des équations relatives aux fonctions test ft_j.

Chapitre 7 Application de l'approche multi-paradigmes à la méthode des éléments finis

7.1 Domaine

Le domaine d'application est ici la résolution d'équations aux dérivées partielles (EDP) par la méthode des éléments finis (MEF). Ce domaine est connecté aux domaines du maillage et de la visualisation des résultats. Cependant, lier des applications de ces différents domaines ne demandant dans la plupart des cas rien de plus que la spécification de formats de fichiers, on peut considérer la résolution d'EDP par la MEF comme étant un domaine déconnecté. Certains traitements spéciaux, comme l'adaptation de maillage, exigent une connexion plus forte entre la MEF et un autre domaine. Cependant, toujours dans le cas de l'adaptation de maillage, l'impact sur la structure générale de l'application est minimal et peut être ignoré dans une première approximation.

7.2 Énumération des entités

L'objectif de cette partie est de repérer les entités susceptibles d'être retenues lors de la modélisation du moteur de calcul par élément fini généraliste. Pour ceci, on part d'un court résumé de la méthode des éléments finis. Il est essentiel de garder à l'esprit que les éléments de ce dictionnaire ne sont par forcément tous retenus et que d'autres éléments peuvent apparaître au cours du processus d'analyse et de développement.

« La MEF résout des EDP. Les EDP sont appelées formulations lorsqu'elles sont exprimées avec leurs conditions aux limites sous une forme variationnelle. Une EDP est composée de termes. Les termes peuvent être linéaires, bilinéaires, trilinéaires et ainsi de suite. Des lois physiques peuvent intervenir dans certains termes. Pour un problème donné, des espaces fonctionnels de fonctions de forme, de fonctions test et de fonctions de forme géométrique doivent être définis. Comme ce n'est pas directement l'EDP qui est résolue, mais sa forme faible, il est nécessaire d'utiliser un intégrateur pour évaluer chacun des termes de la formulation sur chacun des éléments du maillage discrétisant géométriquement le domaine du problème. Afin de conserver une grande souplesse, les fonctions de forme sont évaluées dynamiquement en un point de l'élément de référence. On a donc besoin de calculer le changement de coordonnées permettant d'obtenir les valeurs dans l'espace « physique » à partir des valeurs calculées dans l'espace de référence. Une autre étape importante est l'imposition de conditions aux limites. Finalement, assembler toutes les matrices élémentaires calculées sur chaque élément requiert l'utilisation d'une matrice globale, que l'on résoudra avec un résoluteur. Dans certains cas, il est nécessaire de résoudre successivement, voire itérativement. plusieurs EDPs. ».

On souhaite également disposer d'une méthode d'intégration adaptative afin d'être capable d'intégrer n'importe quel type de fonction et parce que cela est utile dans d'autres applications, comme le suivi d'une fissure dans un élément par exemple.

Tableau 7.1 – Dictionnaire

MEF	Méthode des éléments finis, c.-à-.d. le domaine d'application.
EDP – Formulation	Équations aux Dérivées Partielles (également appelée formulation faible).
Terme	Linéaire, bilinéaires, trilinéaires, etc. Les termes sont les composantess de la formulation.
Loi Physique	Données obtenues de l'expérimentation et organisées en courbes ou modèles mathématiques.
Intégrateur	« Chose » intégrant un terme sur un élément.
Intégrateur adaptatif	Intégrateur ayant la capacité d'intégrer n'importe quel type de fonction sur n'importe quel type d'élément,
Element	Élément géométrique du maillage
Maillage	Ensemble d'éléments

	géométriques couvrant le domaine.
Interpolation	Action d'approximer la solution (et donc les valeurs des degrés de liberté) en n'importe quel point.
Connecteur	Partie d'un élément utilisé comme support d'un degré de liberté. Lorsque l'on interpole linéairement une pression ou une température, on choisit typiquement les nœuds des éléments géométriques comme connecteurs.
Degré de liberté	Valeur associée à un connecteur.
Matrices élémentaires	Résultat de l'intégration numérique d'un terme sur un élément.
Fonction de forme (FF)	Vecteur d'une base d'un espace fonctionnel. On emploiera parfois abusivement le terme fonction de forme pour désigner la base des fonctions de forme, la base des fonctions test et la base des fonctions de forme géométrique.

Espace fonctionnel	Abusivement, l'espace fonctionnel sera la définition des espaces fonctionnels de fonctions de forme, de fonctions test et de fonctions de forme géométrique.
Élément de référence	Élément de l'espace de référence sur lequel les fonctions de forme sont définies.
Changement de coordonnées	Traduction d'une quantité définie dans un système de coordonnées en un autre système de coordonnées.
Condition aux limites	Spécification de la valeur d'une quantité physique ou d'une de ses dérivées en un lieu du maillage
Quantité physique	Quantités physiques modélisées ou utilisées par la formulation (pression, vitesse, température...).
Assemblage	Action de relier les valeurs contenues par les matrices élémentaires en des positions dans la matrice globale.
Matrice globale	Matrice représentant le problème

	entier.
Résoluteur	Brique logicielle capable de résoudre la matrice globale.

7.3 Familles

7.3.1 Éléments du dictionnaire retenus comme familles

L'énumération des familles est immédiate ici : presque tous les mots du dictionnaire constituent une famille. Les familles retenues sont regroupées en deux ensembles.

Le premier contient les familles essentiellement algorithmiques :

- formulations, intégrateurs (adaptatifs ou non), termes, espaces fonctionnels et fonctions de forme

Le second contient les familles représentant des données ou un mélange équitable de données et d'algorithmes :

- éléments géométriques, maillages, degrés de liberté, matrices élémentaires, changement de coordonnées, conditions aux limites et quantités physiques

L'apparition de ces deux ensembles sera importante lors du choix du paradigme d'implémentation.

Toutes ces familles sont essentielles à une couverture complète de la méthode des éléments finis. En enlever une obligerait à abâtardir une ou plusieurs autres familles.

Supprimons par exemple la famille des changements de coordonnées. Il faudrait alors déporter sa fonctionnalité dans la famille des éléments géométriques ou dans la famille des fonctions de forme. Or, l'élément géométrique ne représente et ne doit être qu'une information géométrique. Inclure le changement de coordonnées nous brimerait lors de l'implémentation de nouveaux changements de coordonnées dans cette famille : on serait alors forcé d'utiliser un mécanisme de switch/case. Il est toujours préférable d'éviter le recours à ce mécanisme, coûteux en temps et qui rend le code moins lisible et donc moins extensible et facile à entretenir. Intégrer le changement de coordonnées aux fonctions de forme poserait le même genre de problème.

Ces familles constituent l'énumération *à cette étape* des familles de FEMView. On veut dire par là que la suite de l'analyse peut faire apparaître de nouvelles familles qui n'auraient pas encore été identifiées.

7.3.2 Éléments du dictionnaire non retenus comme familles

On choisit dans ce développement de séparer le cœur d'assemblage du résoluteur. On définit l'interaction avec ce dernier par une interface. C'est pourquoi on ne retient pas comme familles les résoluteurs ou encore les matrices globales. Certains autres termes sont écartés, comme Interpolation. Il ne définit que l'action qu'effectue les fonctions de forme et les espaces fonctionnels. Élément de référence n'est pas une famille car il ne traduit qu'un concept déjà pris en compte par le mode de définition des fonctions de formes géométriques et l'existence d'une famille de changement de coordonnées.

Maintenant que nous avons brièvement justifié du choix des familles, analysons les communautés et les variations de chacune d'entre elles. L'assemblage, quant à lui, est effectué par les termes. Le mode d'assemblage étant unique, et comme on ne voit aucune raison d'en changer, on choisit de ne pas faire de l'assemblage une des familles de FEMView.

7.4 Analyse Points communs/Variations des fonctions de forme

Variations	Les ensembles (un pour chaque type d'élément géométrique) de connecteurs servant de supports aux fonctions de la base d'interpolation, le type de la valeur de retour (vectoriel ou scalaire). Certaines fonctions de forme peuvent être utilisées pour calculer le changement de coordonnées de l'espace de référence vers l'espace physique et sont dites géométriques.
Points communs	L'ordre d'interpolation

Le seul point commun entre les fonctions de forme semble être la spécification du degré de l'interpolation. Le reste des fonctions de forme est entièrement variable (on laisse de côté les fonctions de forme géométrique pour l'instant) :

- la définition des ensembles de connecteurs servant de support aux fonctions de la base d'interpolation;

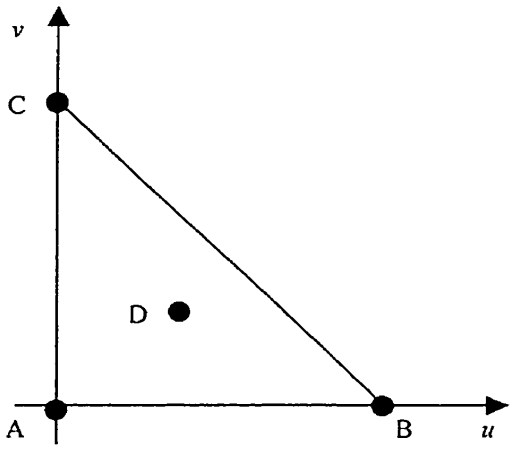
- le type de la valeur de retour, c'est-à-dire, le type de la grandeur interpolée (en termes d'analyse vectorielle, ce type peut être scalaire ou vectorielle).

Comme expliqué dans la partie « Description mathématique de la méthode des éléments finis », les connecteurs sont des parties de l'élément géométrique (comme les nœuds, les arêtes ou les faces) voire l'élément géométrique lui-même. Ils servent de support aux degrés de liberté. Comme le rôle des fonctions de forme est d'interpoler ou d'approximer un champ selon les valeurs aux connecteurs, il est naturel d'utiliser une

fonction d'interpolation pour chacun d'eux. Prenons l'exemple d'un élément triangulaire linéaire de Lagrange. Les connecteurs sont alors les nœuds de l'élément. Selon que l'on interpole un champ vectoriel ou un champ scalaire, on va placer un (dans le cas scalaire), deux (dans le cas vectoriel en deux dimensions) ou trois degrés de liberté sur chaque connecteur. Les fonctions de forme sont définies pour chaque connecteur et interpolent les degrés de liberté qui y sont placés.

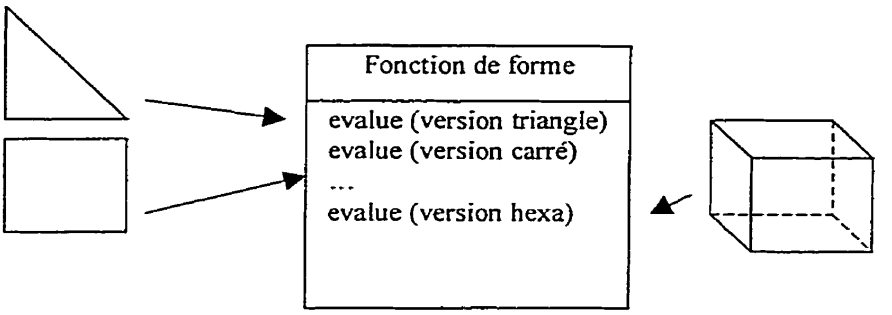
Tableau 7.2 - Fonctions de forme de lagrange sur un triangle

- $1 - u - v$ ayant pour support le nœud A
- u ayant pour support le nœud B
- v ayant pour support le nœud C



Ceci est plus amplement développé dans la partie « Gestionnaire de degrés de liberté ».

Gérer la première variation, à savoir la définition des ensembles de connecteurs, est trivial. Comme la définition de chaque fonction de forme est liée à la définition de ces ensembles, il est légitime de leur consacrer une méthode de chaque fonction de forme, surchargée pour chaque type d'élément. On obtient ainsi une bijection entre chaque surcharge de méthode et chaque définition d'ensemble de connecteurs supports.



La seconde variation peut être traitée directement avec les seuls outils informatiques. Cependant, sa gestion la plus astucieuse et la plus élégante passe par l'utilisation d'un outil mathématique : la géométrie différentielle et plus particulièrement, les formes différentielles. La section suivante les réintroduit et présente les différents aspects à considérer pour les implémenter.

7.5 Les formes différentielles : la clé mathématique de notre design

7.5.1 Présentation

On s'attache dans cette partie à présenter les formes différentielles dans une optique très appliquée, en vue de les utiliser dans un moteur de calcul par éléments finis. On ne cherche donc pas à introduire toute la complexité de la géométrie différentielle mais uniquement ce qui nous est utile dans le cadre du développement de FEMView. Le lecteur peut se référer à "Geometrical methods of mathematical physics" de Bernard Schutz pour une présentation plus complète de la géométrie différentielle [2].

Les formes différentielles sont définies sur des *variétés* (espace localement $\Re^n$) et une *p-forme* est évaluée sur une *p-chaîne*. Une 0-chaîne est un point, une 1-chaîne une courbe, une 2-chaîne une surface, etc. Ces objets géométriques sont indépendants de tout

système de coordonnées et donc définis dans un espace de dimensions *n*. On peut démontrer qu'une p-forme a C_n^p composantes indépendantes. Les 0 et 3-formes ont donc une représentation scalaire et les 1 et 2-formes une représentation vectorielle. La seule évaluation d'une p-forme ayant un sens est son évaluation sur une *p-chaîne*. On en déduit donc que :

- Les 0-formes sont des quantités évaluées en un point : des potentiels, comme la température.

- Les 1-formes sont des quantités évaluées sur un chemin. Le travail est un exemple de 1-forme.

- Les 2-formes sont des quantités évaluées sur une surface. La densité d'induction magnétique B est un exemple de 2-forme, ainsi que toutes les densités surfaciques.

- Les 3-formes sont évaluées sur un volume. Ce sont donc toutes les densités volumiques.

Les 0-formes sont des quantités scalaires indépendantes de la métrique. Les 1-formes dépendent de la métrique et peuvent être écrite de la manière suivante (en 3D) :

$$f_X\, dX + f_Y\, dY + f_Z\, dZ$$

Les 2-formes quant à elles peuvent être écrites de la manières suivante :

$$f_{YZ}\, dY \times dZ + f_{ZX}\, dZ \times dX + f_{XY}\, dX \times dY$$

Et finalement, les 3-formes ont une représentation scalaire et peuvent être notées :

$$f_{XYZ}\, dX \times dY \times dZ$$

Cette manière de noter les formes différentielles implique que l'on travaille dans l'espace cartésien tridimensionnel. On voit clairement apparaître pour chacune des formes sa nature uni, bi ou tridimensionnelle. Dans FEMView, on se situe toujours dans cet espace. Cela nous permet de précalculer les opérations s'appliquant aux différentielles. Il faut cependant souligner que les formes différentielles, dans leur forme « pure » sont indépendantes de tout système de coordonnées et que c'est uniquement pour des questions d'efficacité que l'on précalcule les opérateurs et transformations. On ne représentera par les suites les formes différentielles que par leurs coefficients f_i, la partie fonctionnelle étant sous-entendue.

7.5.2 Changement de coordonnées

(Dans cette section, J est la matrice jacobienne du changement de coordonnées).

Le changement de coordonnées d'une forme est défini de la manière suivante :

- ne rien faire dans le cas d'une *0-forme;*
- multiplier par J^{-1} dans le cas d'une *1-forme;*
- multiplier par *J/det J* dans le cas d'une *2-forme;*
- multiplier par *det J* dans le cas d'une *3-forme.*

7.5.3 Formes multiples

Certaines quantités physiques sont représentées par une *N*p-forme*. Le déplacement est une de ces quantités : *chacune de ses composantes est une 0-forme indépendante*. Le déplacement est par ailleurs plutôt un vecteur qu'une forme différentielle, mais il est pertinent de le représenter comme une *3*0-forme*.

7.5.4 Algèbre extérieure

Le produit d'une *j-forme* et d'une *k-forme* est une *(j+k)-forme*. Si *j+k>N*, (*N* étant la dimension de l'espace, soit 3 dans le cas de FEMView) cette forme est nulle. La multiplication d'une :

- *0-forme* par une *p-forme* consiste à multiplier chacune des composantes de la p-forme par la valeur scalaire de la *0-forme*, le résultat étant une *p-forme;*

- *1-forme* par une *1-forme* est le produit vectoriel des deux représentations et donne une *2-forme*;

- *1-forme* par une *2-forme* est le produit scalaire des deux représentations et donne une *3-forme*.

On peut donner une justification intuitive des ces opérations. La multiplication entre deux 1-formes, par exemple, est notée :

$$F_1 F_2 = \sum_{i,j} f_i f_j d\varphi_i \times d\varphi_j$$

Sachant que $d\varphi_i \times d\varphi_i = 0$, il est aisé de se rendre compte que le résultat est une *2-forme*.

La multiplication par une *3*0-forme* est un cas particulier: c'est un produit contracté entre la représentation vectorielle de la *3*0-forme* et celle de la *p-forme*. Le résultat est une *(p-1)-forme*.

La multiplication par une loi physique, quelle que soit sa représentation, est effectuée comme en algèbre vectorielle, selon la représentation de la forme.

7.5.5 Dérivée extérieure

On peut montrer que la dérivée extérieure d'une *p-forme* est une *(p+1)-forme.* De plus, en laissant quelque peu de côté la rigueur mathématique (mais elle n'est pas notre principal soucis; nous voulons plutôt donner une description des formes permettant d'implémenter leur comportement), on peut relier la dérivation d'une p-forme aux opérateurs usuels de l'analyse vectorielle :

- la dérivée extérieure d'une *0-forme* est effectuée par l'application de l'opérateur *gradient;*
- la dérivée extérieure d'une *1-forme* est effectuée par l'application de l'opérateur *rotationnel*;
- la dérivée extérieure d'une *2-forme* est effectuée par l'application de l'opérateur *divergence*

La dérivée extérieure d'une 3-forme donnant une 4-forme, nulle en 3D, on ne s'attarde pas à la décrire ici.

7.5.6 Utilisation dans FEMView

Dans FEMView, chaque quantité physique est représentée par une forme différentielle. Les bases d'interpolation n'interpolent donc pas simplement des valeurs scalaires, mais des *formes différentielles. Le type de la valeur de retour des fonctions de forme est donc une forme différentielle.* L'avantage de cette approche sur l'utilisation d'un simple type opaque représentant à la fois des quantités scalaires et vectorielles est de pouvoir utiliser l'algèbre présentée ci-dessus pour effectuer toutes les opérations s'appliquant aux quantités manipulées sans se soucier de la nature réelle de la forme : l'algèbre se charge de déterminer et d'exécuter la bonne opération. Cela induit d'immenses bénéfices en

termes de généralité et de compacité du code. Les formes différentielles, et plus largement la géométrie différentielle, sont la clé mathématique de FEMView.

Les formes différentielles constituent une famille de FEMView composée d'une unique classe *form*.

7.6 Analyse des points communs et des variations entre fonctions de forme – deuxième partie

7.6.1 Fonction de forme

L'ajout d'une famille de forme différentielle, constituée d'une seule classe, *form*, dans notre dictionnaire permet de décrire les variations de type de retour des fonctions de forme. Elle permet également de faire ressortir un autre point commun entre fonctions de forme : comme certaines formes peuvent être multiples et que nous choisissons d'interpoler chacune des composantes de la représentation d'une forme comme une quantité indépendante, la base d'interpolation doit être capable de retourner des formes multiples. La spécification de la nature simple ou multiple de la représentation de la quantité interpolée est donc un autre point commun entre toutes les fonctions de forme.

Reste à traiter les fonctions de forme géométriques. Ces fonctions de forme sont capables de faire tout ce que fait une fonction de forme et peuvent en plus être utilisées pour calculer le changement de coordonnées de l'espace de référence vers l'espace physique. C'est un cas typique de spécialisation : un sous-ensemble de l'ensemble des fonctions de forme est doté de capacités particulières. Afin de facilement reconnaître ces fonctions de forme géométrique, on crée un type, `geometrical_shape_function`, dont toutes les fonctions de forme géométrique sont dérivées. On pourra ainsi imposer, dans le calcul du changement de coordonnées par exemple, qu'une fonction de forme soit géométrique en la désignant non pas par un

pointeur sur une `shape_function` mais par un pointeur sur une `geometrical_shape_function` (On rappelle que le C++ interdit de désigner une instance par un pointeur autre que celui de son type ou d'un de ses types de base).

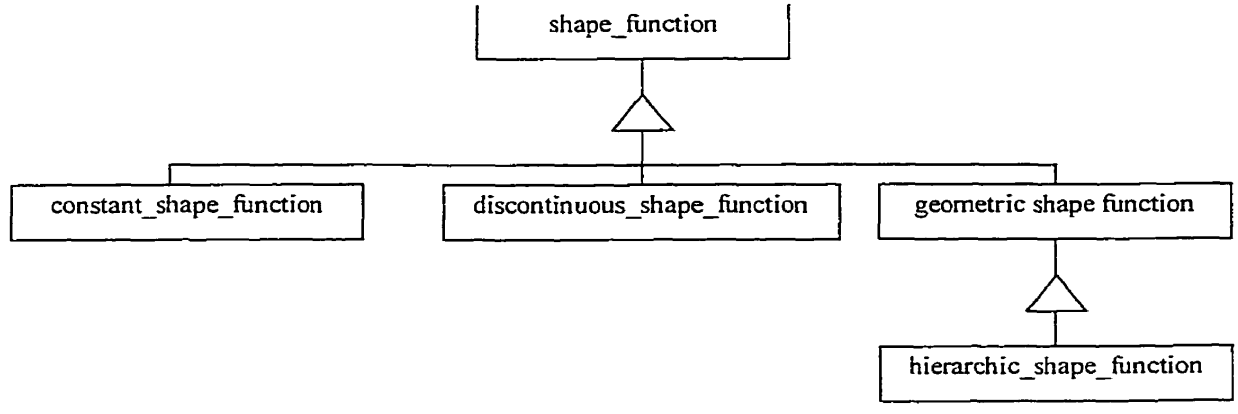


Figure 7.1 - Quelques fonctions de formes de FEMView

7.6.2 Adapteurs

On définit dans un soucis d'efficacité des *adapteurs*. Ce sont des objets capables d'effectuer des opérations simples sur une forme différentielle retournée par une fonction de forme comme prendre la partie symétrique. Il est possible de spécifier un adapteur pour une fonction de forme. Elle appelle alors automatiquement l'adapteur sur les formes différentielles évaluées. On présente plus en détail le fonctionnement des adapteurs dans le chapitre, exemple simple. On y verra l'évaluation d'une divergence dans des conditions quelques peu particulières.

7.7 Analyse points communs/variations des autres familles

7.7.1 Espace fonctionnel

Variations	Aucune
Points communs	Aucun

Dans FEMView, l'espace fonctionnel est, abusivement, le nom de l'objet contenant la définition des fonctions de forme, test, et de forme géométrique à utiliser. Il contient donc les bases d'interpolation à utiliser et sait s'il faut les évaluer directement ou prendre plutôt leur dérivée). Il y a un seul type d'espace fonctionnel et donc pas de place pour une analyse points communs/variations.

7.7.2 Éléments géométriques

Variations	- épaisseur (on rappelle que le signe '-' signale une variation négative)
Points communs	Définis par des nœuds, des arêtes et des faces

Les éléments géométriques partagent une caractéristique : leur définition en termes de nœuds, d'arêtes et de faces. Cependant, cette définition varie selon le type d'élément et justifie un regroupement classique en éléments 0D, 1D, 2D et 3D que l'on spécialise ensuite en points, lignes, triangles, quadrangles, etc. Les feuilles du graphe d'héritage seront chargées de garnir les ensembles de nœuds, arêtes et faces définis dans la classe

de base de la hiérarchie. On traite les points communs en faisant de ces ensembles des attributs de tous les éléments géométriques et la variation en laissant à chaque feuille le soin de les garnir de manière appropriée.

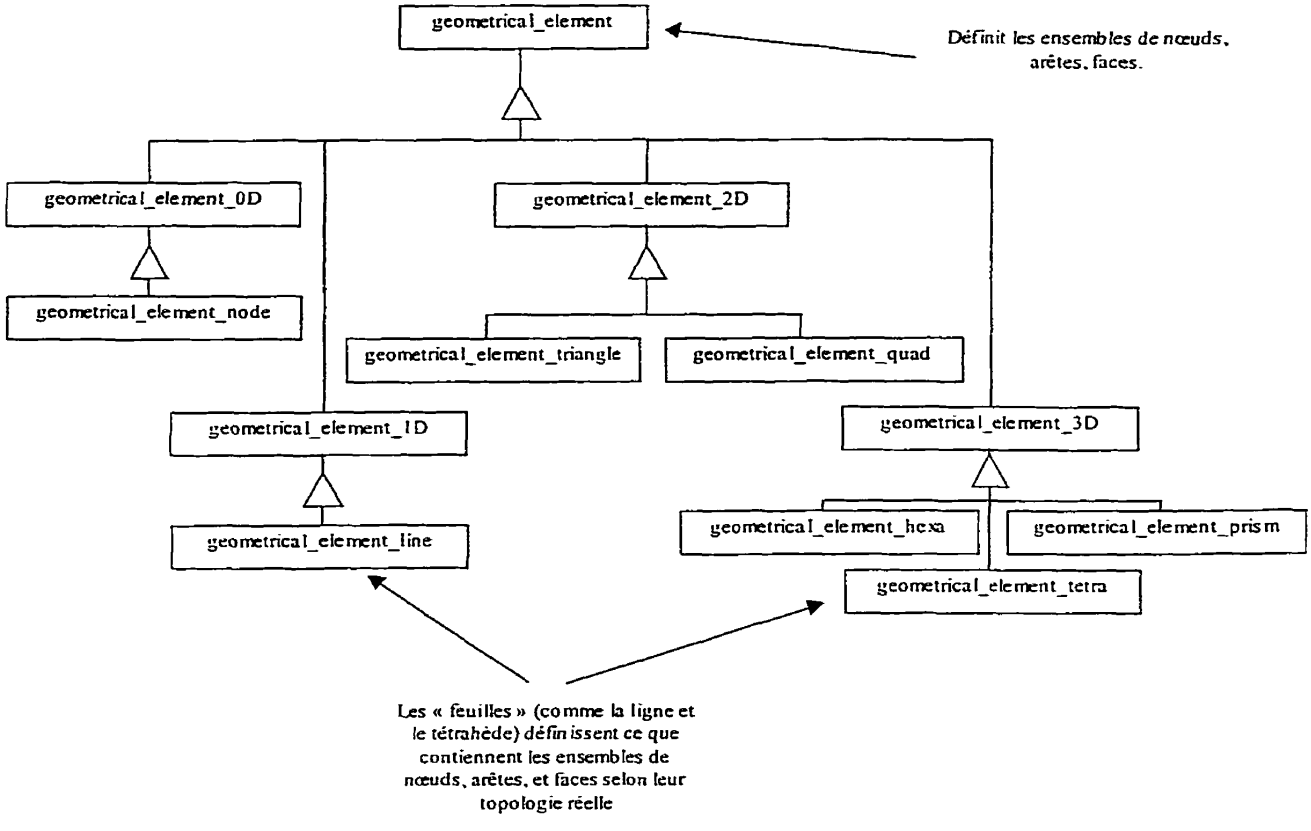


Figure 7.2 – Hiérarchie d'éléments géométriques

D'autre part, on considère dans FEMView que tous les éléments sont des éléments 3D afin de pouvoir utiliser la même formule de jacobien quelque soit l'élément. Tout élément géométrique non 3D est mis au niveau d'un élément 3D en complétant la base

de sa dimension par son complément orthogonal. On ajoute donc à tout élément géométrique non 3D une « épaisseur », permettant de le promouvoir comme élément 3D. Dans le cas d'une ligne, cette épaisseur sera considérée comme étant une section. Dans le cas d'un élément 3D, l'épaisseur est sans signification et est donc ignorée (c'est le seul cas de variabilité négative dans FEMView).

7.7.3 Maillages

Variations	Aucune
Points communs	Aucun

On peut définir un maillage comme étant un ensemble d'éléments géométriques formant une couverture d'un domaine physique. Un maillage peut être divisé en *domaines*, pour distinguer une zone constituée d'un matériau différent. Ceci a des implications sur l'efficacité du code et permet, par exemple, d'éviter de calculer un champ électrique dans une zone non conductrice. Une autre entité importante dans la définition du maillage est la *limite*. Cette limite est le support géométrique d'une condition aux limites. Dans FEMView on définit, par soucis d'homogénéité avec le code LCMFlot, la limite comme étant une paire {élément de volume, élément de face}. Si les éléments géométriques sont des hexaèdres, une telle paire pourrait être l'élément de volume ABCDEFGH et la face BCGF.

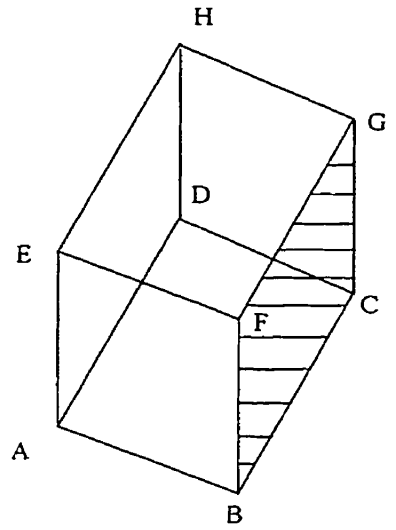


Figure 7.3 - Exemple de couple élément de volume – élément de face

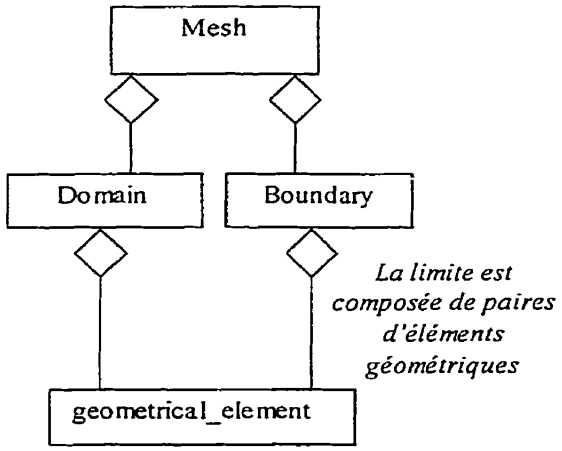


Figure 7.4 - Structuration du maillage

7.7.4 Changements de coordonnées

Variations	Type de changement de coordonnées
Points communs	Change les coordonnées des formes différentielles

Le changement de coordonnées est calculé et appliqué classiquement par l'utilisation d'une matrice Jacobienne, selon les transformations précalculées présentées dans la partie « Changement de coordonnées » de la section « Formes différentielles ». Cette transformation est actuellement unique et implémentée par la classe `JacobianMatrix`. Il est cependant possible qu'un autre type de changement de coordonnées soit introduit dans le futur. La variation cependant réside uniquement dans le mode de calcul du changement de coordonnées. Le mode d'application restera le même et devra être capable de traiter à la fois les formes différentielles et les lois physiques par le biais de l'appel d'une unique fonction membre `ChangeCoord` surchargée pour les types `form` et `PhyicalLaw`. On passe également à cette fonction membre l'élément géométrique courant et le point de son espace de référence où le changement de coordonnées doit être calculé.

7.7.5 Condition aux limites

Variations	La nature mathématique (constante, linéaire,...)
Points communs	Impose un scalaire. Donc, si la forme comporte plusieurs composantes, ne concerne qu'une seule composante à la fois

Les conditions aux limites sont composées d'un contenu géométrique, la limite où est imposée la condition (instance de la classe « Boundary » décrite dans la section « Maillages »), et d'un contenu fonctionnel : la fonction à imposer sur cette limite. La nature de cette fonction peut être très variée, allant de la simple constante jusqu'à la définition dans l'espace physique d'une fonction à appliquer sur un ensemble d'éléments. Afin de permettre à cette généralité de s'exprimer, il est nécessaire de laisser la source des données servant à calculer la condition aux limites totalement libre. On ne fera donc que préciser à cette source l'élément courant et le point de l'espace de référence courant, en échange de quoi elle devra nous retourner une valeur scalaire. La source de ces données (fichier,...), le fait que la condition effectue un calcul préliminaire ou quoi que ce soit d'autre est laissé au bon vouloir du développeur d'un type de condition aux limites particulier.

On confie l'évaluation de la fonction à imposer à une hiérarchie de classes dites de *caractéristiques de condition aux limite* et ayant pour racine la classe `BoundaryCharacteristic`. Chacune des classes de cette hiérarchie correspond donc à un type et/ou à source de données d'une condition aux limites.

La classe `BoundaryCharacteristic` déclare l'opérateur de fonction comme fonction membre virtuelle pure, forçant son implémentation dans ses classes dérivées. Elle définit ainsi la fonction membre d'une classe de caractéristique de condition aux limites appelée lors de l'évaluation d'une condition en un point.

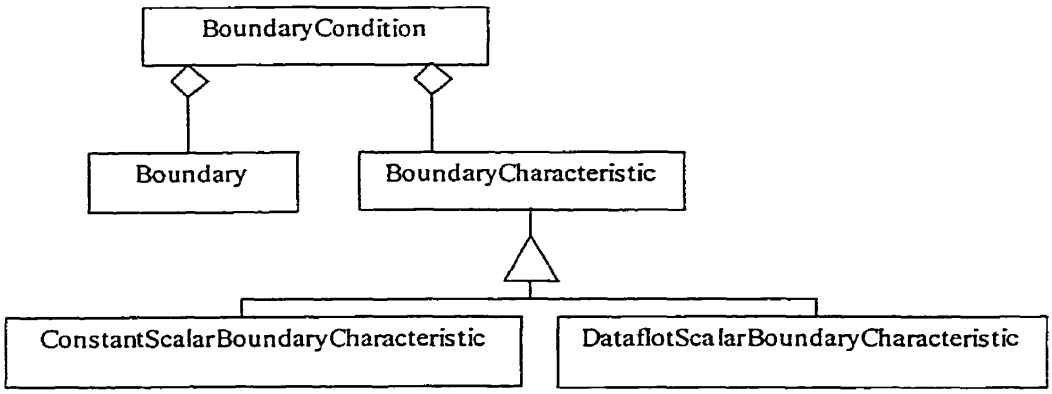


Figure 7.5 - Conditions aux limites

On choisit de n'imposer que des valeurs scalaires afin de simplifier le code. Ce n'est pas une limitation : l'imposition d'une vitesse, par exemple, se fera en imposant les composantes concernées.

On remarquera que l'approche séparant algorithmes et données permet une fois de plus d'obtenir une grande souplesse : on peut ici se servir de *Dataflot*, la composante de LCMFlot servant à la définition des lois physiques et conditions aux limites. On pourrait de manière similaire définir ainsi une interface avec n'importe quelle source de données.

7.7.6 Lois Physiques

Variations	La nature mathématique (constante, linéaire,...) (scalaire, tensorielle,...)
Points communs	Multiplié par des formes différentielles

Les lois physiques sont déterminées par l'expérimentation et sont, tout comme les conditions aux limites, de nature versatile. Afin de traiter leur généralité, on applique une approche similaire à celle utilisée pour les conditions aux limites. Mais au lieu de séparer le support géométrique du mode de représentation, c'est la représentation de la loi physique que l'on sépare de son évaluation. On définit ainsi une classe `PhysicalLaw` représentant n'importe quelle type de loi physique, qu'elle soit scalaire, matricielle ou encore tensorielle. La définition du type de la représentation et de sa valeur est laissée à la charge des classes de *caractéristiques*. Ces classes retournent donc, lors de l'appel de leur opérateur de fonction, une loi physique calculée en point de l'espace de référence (passé en paramètre) de l'élément géométrique passé en paramètre.

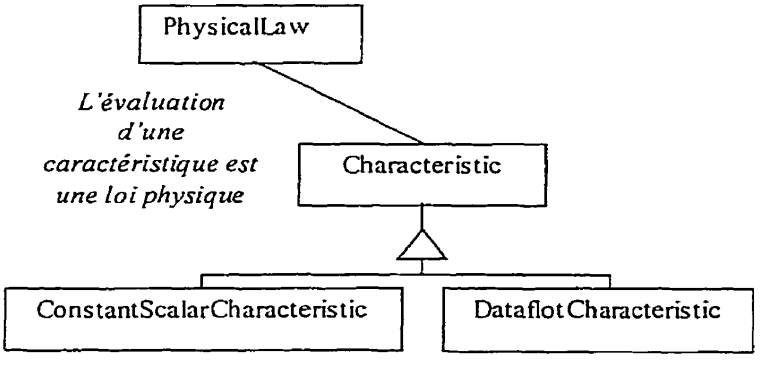


Figure 7.6 - Hiérarchie de caractéristiques

De manière analogue aux conditions aux limites, cette approche permet de définir des classes chargées d'extraire des données fournies dans un format particulier, comme celui de Dataflot, pris en charge par la classe `DataflotCharacteristic`.

Pourquoi faire de la loi physique un type unique au lieu de créer une hiérarchie de classes dont chaque membre se chargerait d'une représentation particulière? Souvenons-nous que lors du changement de coordonnées ou lors de la multiplication par une forme, une surcharge de fonction est effectuée et l'un des types surchargé est le type PhysicalLaw. Si l'on avait une hiérarchie de classes de loi physiques, on devrait avoir autant de surcharges de fonctions dans la classe de matrice jacobienne et dans la classe de forme différentielle, alors que cela n'apporte ni en élégance, ni en efficacité, encore moins en compacité. On choisit donc d'utiliser un type unique.

7.7.7 Intégrateurs

Variations	La méthode d'intégration
Points communs	Intègre des termes linéaires ou bilinéaires sur un élément ou sur le bord d'un élément

7.7.7.1 Définition générale

Dans FEMView, les intégrateurs sont responsables de l'intégrations des matrices élémentaires. Ils doivent être capables de traiter deux types de matrices élémentaires : un vecteur dont le nombre d'éléments est égal au cardinal de la base d'interpolation des fonctions test (NB FONCTIONS TEST) dans le cas d'un terme linéaire, une matrice rectangulaire de dimension NB FONCTIONS DE FORME * NB FONCTIONS TEST dans le cas d'un terme bi ou trilinéaire.

Un intégrateur est appelé par un terme (décrit plus loin) sur chacun des éléments du domaine sur lequel le terme doit être assemblé. On lui indique également, pour chacun de ces éléments, l'espace fonctionnel courant, l'élément géométrique utilisé comme

support de l'intégration, l'élément géométrique à utiliser pour calculer le changement de coordonnées. On verra par la suite que distinguer l'élément support de l'élément utilisé pour calculer le changement de coordonnées est indispensable afin implémenter efficacement l'intégration récursive. L'intégrateur, ayant alors suffisamment d'informations détermine automatiquement le nombre de points d'intégration. Il effectue pour cela la somme des degrés des espaces fonctionnels de fonctions de forme et de fonctions test, et la retranche de un pour chacun de ces espaces dont on prend la dérivée extérieure.

7.7.7.2 Intégration récursive

On s'était imposé lors de l'énumération des familles la contrainte de pouvoir intégrer n'importe quel type de fonction de forme sur n'importe quel type d'élément. Que devons nous faire pour satisfaire cette contrainte? Il nous faut découper l'intégration récursive en quatre parties :

- la *simplexation* des éléments géométriques;
- l'*évaluation* de l'intégrale sur la simplexation d'un élément;
- l'*estimation* d'un critère d'arrêt
- le *schéma* de récursion

Toutes ces parties sont des familles de classes, à l'exception de la simplexation.

7.7.7.2.1 Simplexation

Le *simplexe* d'un espace de dimension N est la *N-chaîne* la plus simple de cette espace. Il est en fait composé de N+1 points. On appelle *simplexation* l'opération consistant à découper un élément en simplexes de sa dimension.

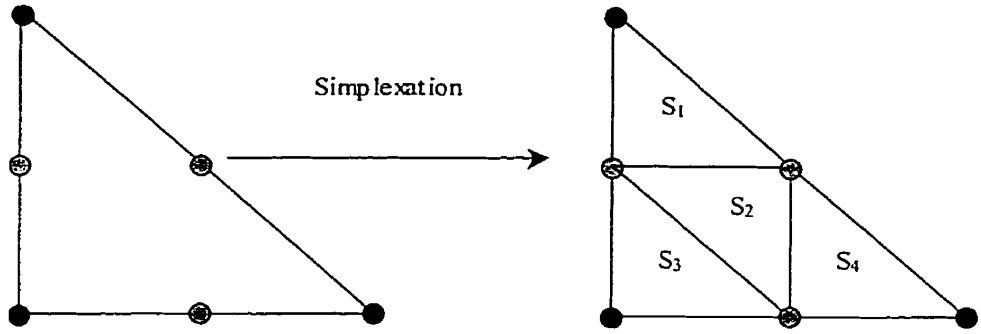


Figure 7.7 - Simplexation d'un triangle

Comme il n'existe que très peu de façons correctes de découper un élément géométriques, on choisit d'en retenir une pour chaque élément et de la coder directement dans cet élément. On viole ainsi notre dogme de séparation des algorithmes et des données. Cependant, il n'est utile que dans le cas où plusieurs algorithmes différents sont susceptibles d'être utilisés. On ne décrit pas la simplexation de tous les éléments géométriques ici. C'est plutôt le rôle de la documentation de référence.

7.7.7.2.2 Évaluation

L'évaluation a une unique tâche : évaluer l'intégrale sur un simplex donné. Afin d'accomplir cela, nous avons besoin d'ajouter une fonctionnalité supplémentaire à l'intégrateur : la capacité de déterminer les coordonnées dans l'espace de référence de l'élément « global » (le premier élément simplexé) les points d'intégration définis dans l'espace de référence d'un simplex. C'est pour cela que nous avons besoin de séparer l'élément à partir duquel on calcule le changement de coordonnées (l'élément global) de l'élément support (le simplex). Lors de l'évaluation de l'intégrale, ce seront donc les fonctions de forme de l'élément global qui seront utilisées et intégrées sur le simplex dans l'espace de référence de cet élément global.

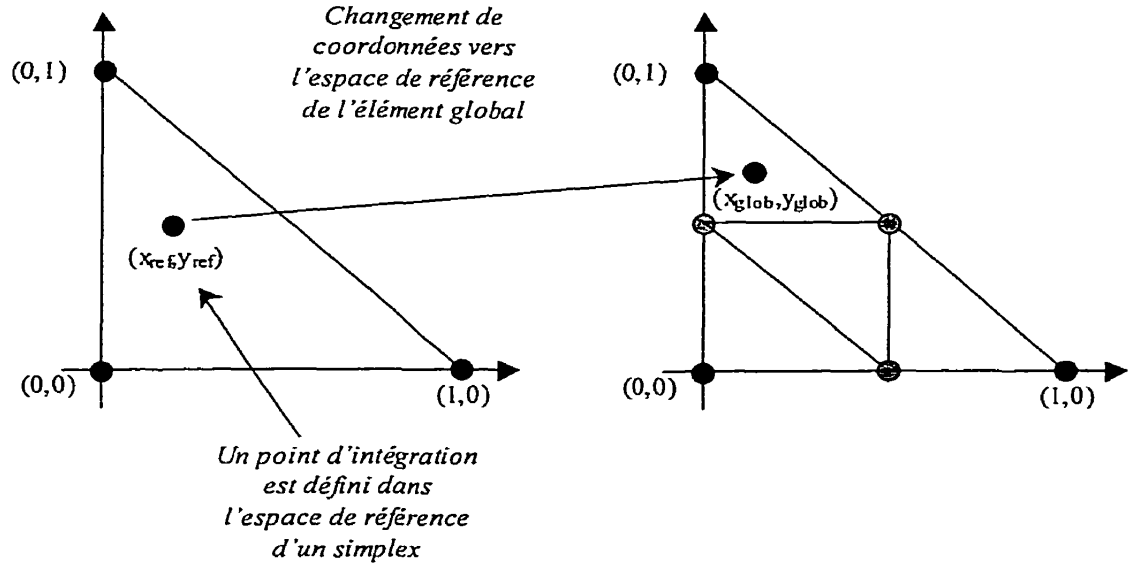


Figure 7.8 - Placement des points d'intégration

7.7.7.2.3 Estimation du critère d'arrêt

L'estimation doit être faite à l'aide de deux données : la matrice locale résultant de l'intégration sur un simplex d'un élément à un niveau de simplexation quelconque et la matrice résultant de l'évaluation de la simplexation de ce simplex. Un critère d'arrêt simple pourrait être, par exemple, d'imposer que la différence entre chaque composante des deux matrices soit inférieure à une certaine valeur.

7.7.7.2.4 Schéma de récursion

Les livres d'analyse numérique sont pleins de schémas d'intégration. Certains, comme l'intégration de Romberg, sont intrinsèquement adaptés à l'intégration récursive. Le schéma de récursion contrôle quel élément est simplexé, le moment où il est estimé, comment les résultats des intégrations sur les simplexes sont rassemblés et le mode

d'évaluation de la convergence. C'est en quelque sorte le chef d'orchestre de l'intégration récursive et l'utilisateur des trois autres familles. À ce titre, c'est cette classe qui appartient à la famille des intégrateurs. Cela nous permet d'utiliser indifféremment un schéma classique d'intégration ou un schéma récursif.

7.7.7.2.5 Prise en compte des variations de l'intégration récursive

Comme on a fourni trois degrés de liberté : le mode d'évaluation, le mode d'estimation et, le principal, le schéma de récursion, on affirme que n'importe quel schéma récursif peut être implémenté et que toute la variabilié est donc contenue dans ce design.

7.7.7.3 Intégration sur le bord

Vu le nombre important de cas où l'intégration d'un terme de bord est nécessaire, il est fondamental que tous les intégrateurs disposent de cette fonctionnalité. Dans FEMView, on calcule les intégrales de bord en intégrant les fonctions de forme de l'élément de volume sur la face de cet élément constituant le bord. Pour réaliser cela, il suffit simplement de déterminer les points d'intégration correspondant au degré du terme à traiter et de les translater de l'espace de référence de l'élément de bord vers l'espace de référence de l'élément de volume.

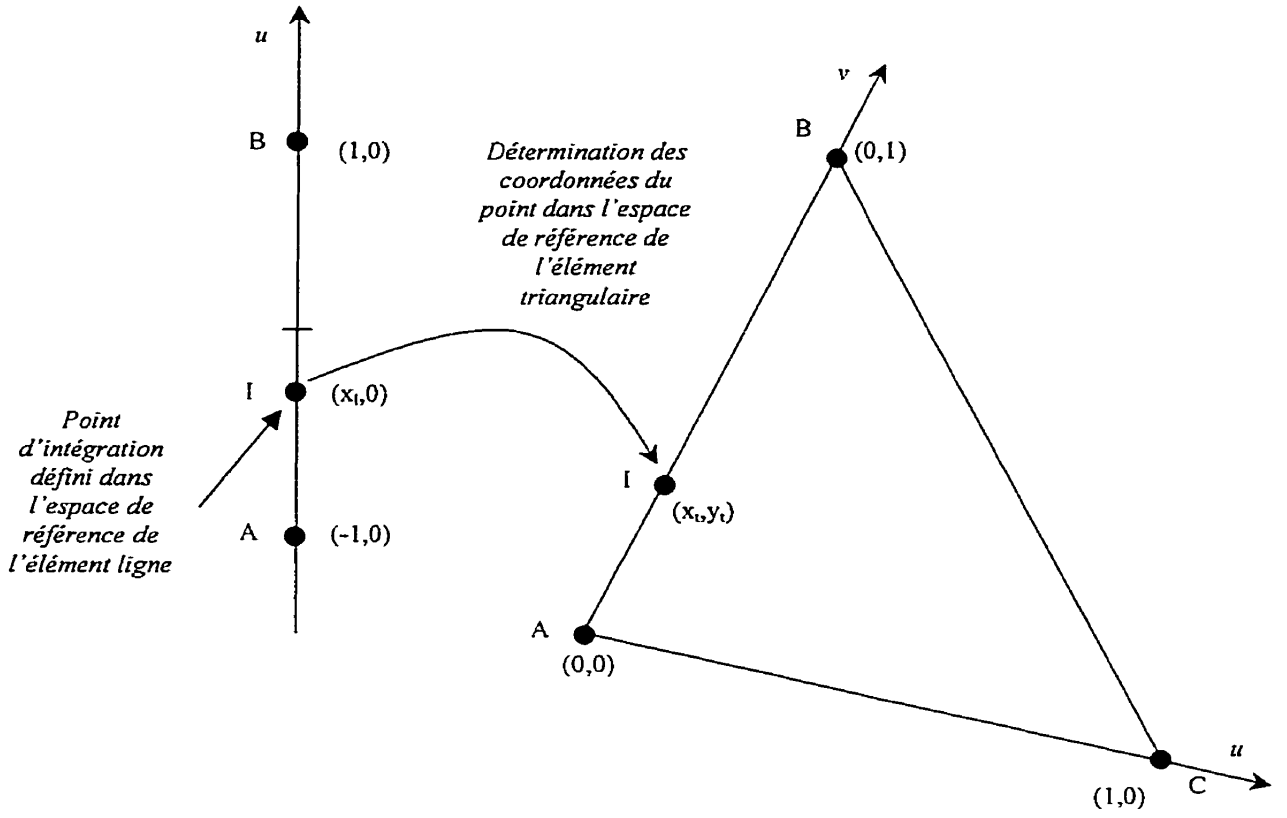


Figure 7.9 - Translation des points d'intégration de l'élément de bord de l'espace de référence de l'élément de bord vers celui de l'élément de volume

Il est vrai qu'en procédant de cette façon on effectue parfois des intégrations inutiles. Dans le cas de fonctions de forme de lagrange utilisées sur un élément triangulaire, par exemple, la fonction supportée par le nœud opposé à l'arête est nulle sur cette arête. Cependant, le coût (en temps ou en occupation mémoire) de la gestion de telles exceptions étant supérieur à celui du calcul de quelques intégrales superflues, on s'en tient à la solution proposée. L'implémentation de l'intégration de bord est intégrée de telle manière à ce qu'une intégration récursive sur un bord, de manière similaire à ce qui a été décrit précédemment, soit possible.

7.7.8 Termes

7.7.8.1 Présentation

Termes :

Variations	Linéaires, bilinéaires ou trilinéaires; de volume ou de bord; comporte ou non une loi physique.
Points communs	Calcule et assemble un terme (correspondant directement à un terme de l'équation de la formulation faible) sur un domaine en utilisant un espace fonctionnel et un intégrateur.

Assembleurs de matrices élémentaires :

Variations	Peut être linéaire, bilinéaire ou trilinéaire; de volume ou de bord;comporte ou non une loi.
Points communs	Effectue le calcul d'un terme sur un élément en un point d'intégration donné et ajoute sa contribution à une matrice élémentaire

Les termes sont la transposition littérale des termes mathématiques. Ils calculent et assemblent les matrices élémentaires correspondant à un terme sur les éléments d'un domaine (ou d'un bord) donné. Un terme utilise l'intégrateur qui lui est assigné pour calculer les matrices élémentaires. Afin d'accroître la réutilisabilité et la modularité du design, on introduit une nouvelle famille : les *assembleurs de matrices élémentaires*. Les

membres de cette famille ont la tâche suivante : ajouter la contribution à la matrice élémentaire d'un terme en un point d'intégration donné. L'assembleur de matrice élémentaire est donc appelé par l'intégrateur en chacun des points d'intégration que ce dernier a automatiquement choisi en fonction de la nature du terme.

Lorsque l'intégration a été effectuée, le terme doit déterminer les clés d'interaction avec le gestionnaire de degré de liberté (la génération de ces clés est traitée en détail dans la partie « Gestion des degrés de liberté »). Comme une partie importante de chacune de ces clés dépend de la nature des connecteurs et comme ce sont les fonctions de forme qui définissent l'ensemble des connecteurs les supportant pour chaque type d'élément géométrique, la génération des clés est confiée à l'élément géométrique courant et à la fonction de forme courante. La fonction de forme et l'élément géométrique interagissent en utilisant le mécanisme de « double dispatch » présenté dans la section du même nom de ce document.

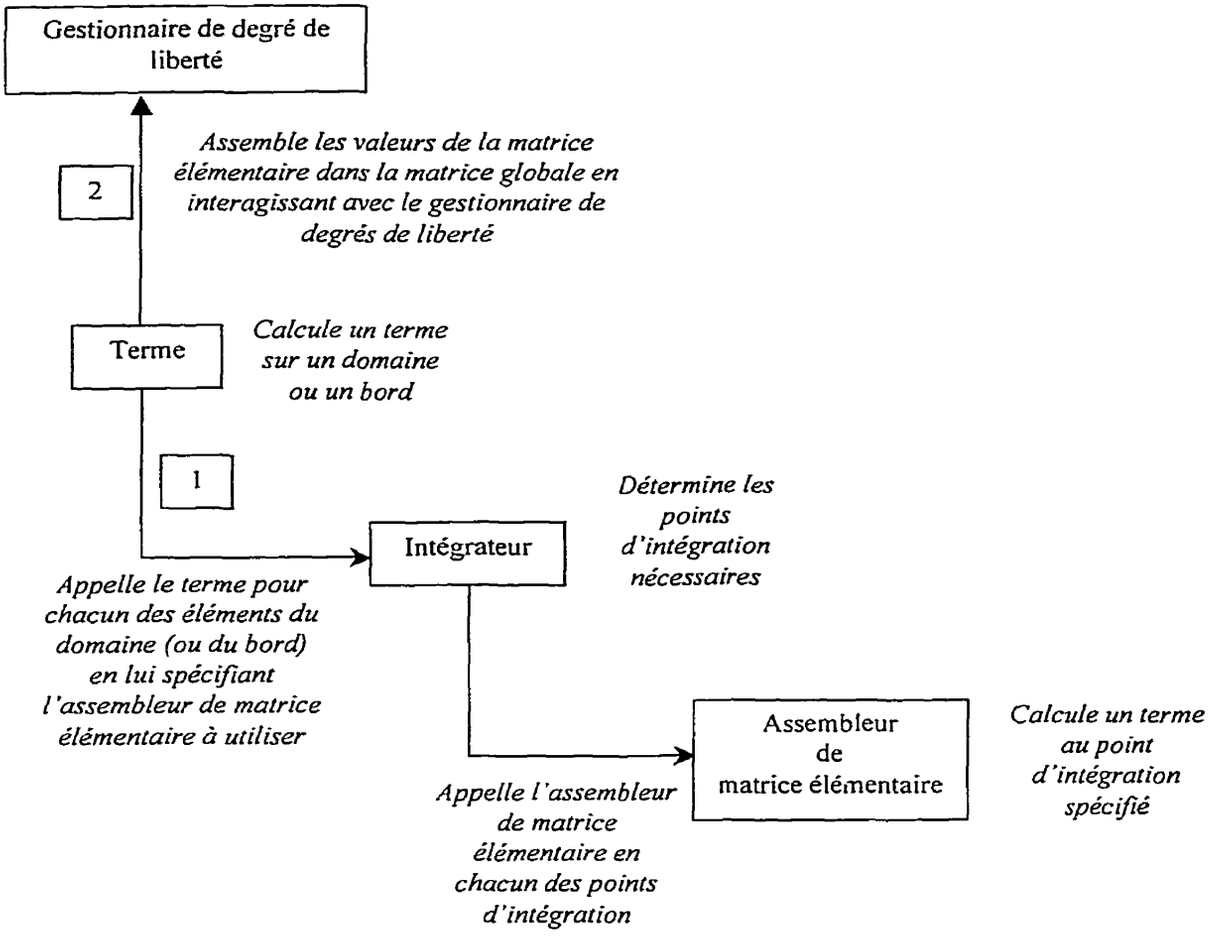


Figure 7.10 - Déroulement de l'assemblage d'un terme

Les termes sont organisés en une famille par la définition d'une interface commune. On ne la définit cependant pas par le biais d'une classe de terme abstraite définissant une interface commune à tous les termes, ce afin d'éviter le surcoût, minime certes mais bien réel, induit par la résolution d'une fonction virtuelle.

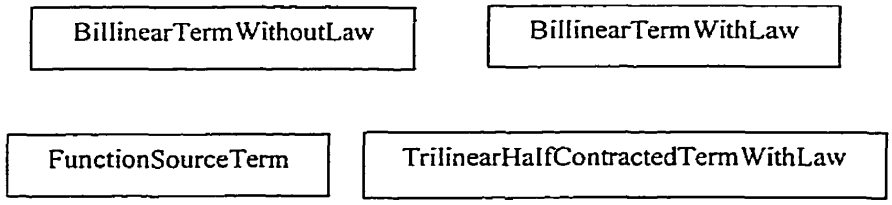


Figure 7.11 - Exemple de termes

Les termes sont peu nombreux. On ne distingue par exemple que deux termes bilinéaires : un comprenant une loi, l'autre n'en comprenant pas. Ceci est dû à la séparation rigoureuse entre les parties mathématiques, physique et géométrique. Un terme n'est qu'une entité mathématique et ne doit donc pas être pollué par d'autres considérations. De plus, un même terme est capable de traiter tous les cas où l'on ne prend pas la dérivée des fonctions de forme et test, la dérivée des unes ou des autres ou encore la dérivée des fonctions de forme et des fonctions test. Cette information est en effet contenu dans l'espace fonctionnel. Elle est prise en compte par l'intégrateur pour déterminer le bon nombre de points d'intégration. Finalement, l'assembleur de matrice élémentaire utilise également cette information afin de savoir si c'est la fonction de forme (ou test) ou sa dérivée qui doit être évaluée. On comprend donc qu'il est alors logique d'avoir aussi peu de termes.

Quatre variations du même terme bilinéaire sans loi	**Un seul terme bilinéaire dans FEMView**
$\int_{\Omega} FF\ FT\ d\Omega$	
$\int_{\Omega} dFF\ FT\ d\Omega$	BillinearTermWithoutLaw
$\int_{\Omega} FF\ dFT\ d\Omega$	*Les variations sont déterminées par l'espace fonctionnel courant*
$\int_{\Omega} dFF\ dFT\ d\Omega$	

Les assembleurs de matrice élémentaire sont organisés en deux hiérarchies de classe, dont les interfaces sont définies par l'utilisation de fonctions membres virtuelles pures dans la classe de base. On est forcé ici de recourir à ce mécanisme car les assembleurs de matrices élémentaires doivent pourvoir être maniés indistinctement par les intégrateurs.

Deux hiérarchie de classes sont ainsi créées :

- une pour les termes linéaires
- une pour les termes bilinéaires et trilinéaires

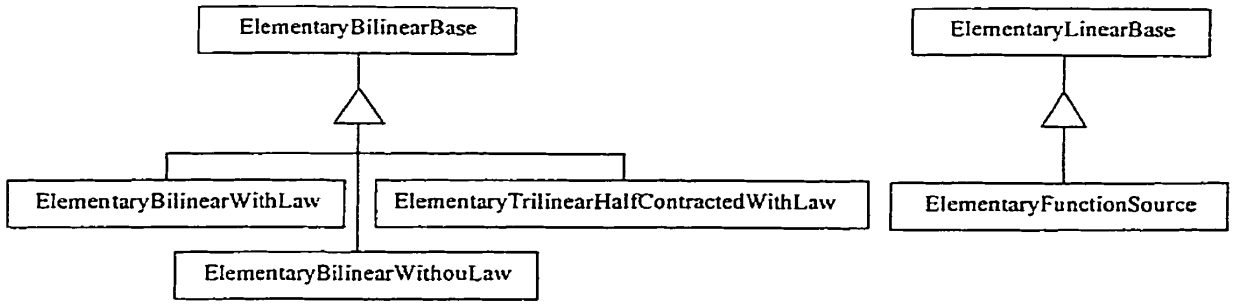


Figure 7.12 - Hiérarchies d'assembleurs de matrices élémentaires

Les termes trilinéaires et bilinéaires sont réunis car comme un terme plus que bilinéaire est forcément contracté (et un résoluteur non-linéaire, comme un point fixe, est alors utilisé), il peut être considéré sans perte de généralité comme un terme bilinéaire par l'intégrateur.

7.7.8.2 Termes linéaires et fonctions

Les termes linéaires sont principalement des termes sources. On y multiplie généralement par la base de fonctions test l'évaluation d'une fonction en un point. Aussi est-il nécessaire d'introduire une famille de *fonctions* utilisées par un unique terme source. Ces fonctions doivent être capables de calculer leur valeur en un point d'un élément donné à un pas de temps donné. Le reste (données sources pour le calcul) est laissé entièrement libre, ceci afin de disposer d'une grande souplesse. D'autres termes linéaires existent, comme le terme trilinéaire doublement contracté de la formulation Lesaint-Raviart. Cependant, ces termes étant rarement rencontrés, on préfère les décrire dans le contexte de leur utilisation.

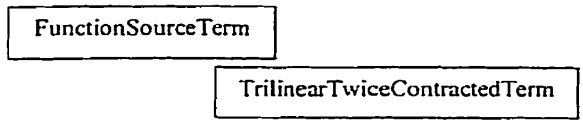


Figure 7.13 - Exemples de termes sources

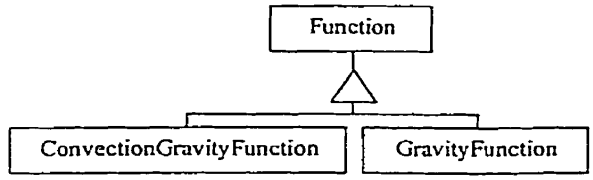


Figure 7.14 - Exemples de fonctions sources

7.7.8.3 Termes trilinéaires et Contracteurs

Les termes trilinéaires sont ramenés dans FEMView à des termes bilinéaires par une contraction des formes évaluées par les fonctions de forme ou les fonctions test par une troisième quantité. Dans FEMView, cette quantité est calculée et la contraction est effectuée par un *contracteur*. Les contracteurs constituent une famille de FEMView et sont appelés logiquement par les assembleurs de matrice élémentaires de termes trilinéaires.

Un contracteur dispose des informations suivantes :

- l'élément géométrique courant;
- le pas de temps courant.

Il les utilise, généralement, pour aller récupérer une quantité dans une source de données. On définit ainsi des contracteurs capable d'aller récupérer une forme différentielle dans un gestionnaire de degrés de liberté. On peut ainsi, comme on le verra dans l'exemple convection thermique, récupérer la vitesse calculée par une formulation (en ce cas, Navier-Stokes) pour la contracter avec la grandeur évaluée par les fonctions

de forme ou test (en ce cas la vitesse évaluée par les fonctions de forme lors du calcul de la formulation de convection thermique).

D'autres adapteurs peuvent être définis, utilisant d'autres données. La seule contrainte qu'ils aient à respecter est d'être capable de traiter les informations concernant l'élément courant et le pas de temps courant ainsi que d'effectuer la contraction par le biais de l'invocation de la surcharge de leur opérateur de fonction :

```
contToUse(formToContract,&formResult);
```

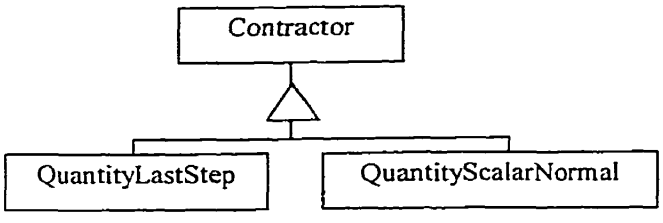


Figure 7.15 – Contracteurs

Les deux contracteurs présentés ci-dessus récupèrent une quantité dans un gestionnaire de degrés de liberté. QuantityScalarNormal calcule le produit de cette quantité avec la normale extérieure d'une des faces de l'élément courant. Cet adapteur est utilisé par un terme trilinéaire de bord. En effet, seul un terme de bord est capable de spécifier un élément et une face de cet élément.

7.7.8.4 Assemblage élément par élément

Il est parfois utile de calculer ou de n'assembler un terme que sur un élément donné. Afin de tenir compte de cela, on fournit deux versions de la fonction principale (son opérateur de fonction). La première version prend, entre autres, un domaine (ou un bord)

en paramètre. Cette première version appelle alors la seconde version pour chacun des éléments qu'elle doit assembler. La seconde version prend, en effet, en lieu et place du domaine (ou du bord) l'élément de volume (et l'élément de bord s'il y a lieu).

Cette séparation en deux fonctions permet d'effectuer l'assemblage sur un seul élément. Si l'on souhaite uniquement calculer la matrice élémentaire, il est tout à fait indiqué d'utiliser directement l'assembleur de matrices élémentaires adéquat.

7.7.9 Formulations

Variations	Très nombreuses
Points communs	Assemble des termes, applique des conditions aux limites, invoque le résoluteur.

Les formulations sont en grande partie une agrégation de termes. On illustrera cela dans la section « Exemples Complets », montrant comment un problème physique correspond à une formulation et à des termes. Avant l'assemblage des termes, il est généralement nécessaire d'imposer des conditions aux limites. Cette tâche est effectuée par des fonctions membres de la classe `Formulation` dédiée à un type de condition aux limites particulier. Une fonction est également fournie, capable de récupérer l'ensemble des conditions aux limites dans le registre de conditions aux limites, de déterminer leurs natures et d'appeler la fonction adéquate à chacune d'entre elles.

Le reste d'une formulation est constitué de tous les traitements particuliers requis par un problème physique donné. C'est afin de permettre la prise en compte de ces particularités que l'on n'impose que très peu de choses aux formulations : fournir une

fonction `TreatmentOfFormulation` provoquant l'assemblage et la résolution de la formulation.

Les formulations sont organisées en une hiérarchie de classes. La classe de base Formulation fournit les services décrit ci-dessus et définit dans son interface la fonction virtuelle `TreatmentOfFormulation`.

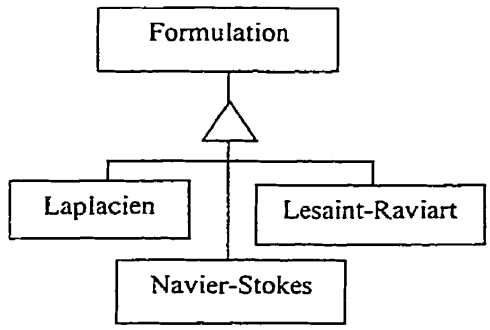


Figure 7.16 - Hiérarchie de formulations

7.7.10 Matrices élémentaires

Variations	N/A
Points communs	N/A

Une matrice élémentaire est uniquement une structure de données capable de contenir une matrice N*M.

7.7.11 Gestion des degrés de liberté

7.7.11.1 Définition des degrés de liberté

La gestion des degrés de liberté, avec la décomposition en termes de familles d'algorithmes et de données ainsi que l'utilisation des formes différentielles, est l'une des principales innovations de FEMView. On définit les degrés de liberté comme étant les coefficients scalaires multipliant les vecteurs de la base d'interpolation (base de l'espace fonctionnel de fonctions de forme), ce quel que soit le type de l'espace fonctionnel.

Considérons un exemple simple : l'utilisation de fonctions de forme linéaires de Lagrange sur un triangle :

- $1 - u - v$ ayant pour support le nœud A
- u ayant pour support le nœud B
- v ayant pour support le nœud C

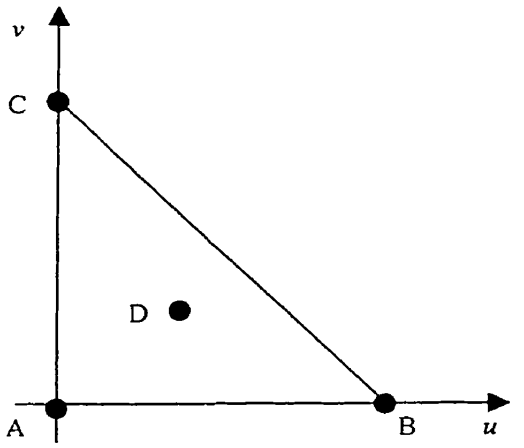


Figure 7.17 – Fonctions de forme linéaires de Lagrange sur un élément triangulaire

Rappelons que les fonctions de forme interpolent les valeurs aux connecteurs en un point de l'élément et que la valeur au point D est :

$$V_D = (1 - u_D - v_D)\, V_A + u_D\, V_B + v_C\, V_C$$

Les degrés de liberté sont les valeurs scalaires V_a, V_b, V_c.

Prenons maintenant un cas légèrement plus compliqué : l'utilisation de fonctions de forme hiérarchiques quadratiques pour interpolation d'un champ de représentation vectorielle.

- $1 - u - v$ ayant pour support le nœud A
- u ayant pour support le nœud B
- v ayant pour support le nœud C
- u-u^2-vu ayant pour support l'arête AB
- $u\ v$ ayant pour support l'arête BC
- $v - u\ v$ -v^2 ayant pour support l'arête AC

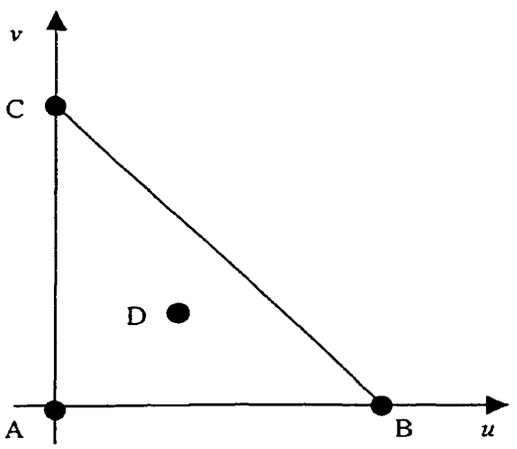


Figure 7.18 – Fonctions de forme hiérarchiques quadratiques sur un élément triangulaire

On utilise cette base d'interpolation pour interpoler chacune des composantes de la représentation vectorielle (une 1-forme ou une 2-forme). Si l'on nomme $B_1 ... B_6$ les fonctions de forme de la base d'interpolation scalaire, il est alors évident que la base d'interpolation vectorielle est :

$$(B_1,0,0),\ (0,B_1,0),\ (0,0,B_1),\ ...,(0,0,B_6)$$

On a par conséquent trois degrés de liberté sur chaque connecteur : un pour chaque composante de la représentation vectorielle.

On aurait pu présenter les degrés de liberté en termes plus rigoureux mathéamtiquement (ce sont en fait des fonctionnelles linéaires), mais comme le thème de ce travail est principalement la conception d'un logiciel, on invite le lecteur intéressé à se référer à Nedelec [3].

7.7.11.2 Gestionnaire de degrés de liberté (DOF manager)

Variations	N/A
Points communs	N/A

Ayant défini ce que sont les degrés de liberté, on peut maintenant présenter leur gestionnaire. Le gestionnaire de degrés de liberté est un outil uniquement algébrique s'occupant de stocker les degrés de liberté et de constituer la matrice globale du système. Chaque degré de liberté est associé à deux clés qui en fin de course indiquent une ligne et une colonne de la matrice globale. Ces clés sont calculées par la fonction `SetNat` de la fonction de forme. Cette fonction est surchargée pour tous les types d'éléments et fait

l'objet d'une interface de double dispatch (voir la description dans la section « Notions avancées de C++ »).

Les clés, implémentées par la classe `Dof_c` sont de même nature et composées de quatre parties :

- l'identificateur du connecteur supportant le degré de liberté, c'est à dire le numéro global (unique) de ce connecteur;
- l'identificateur de la quantité physique, c'est à dire le numéro unique de la variable physique interpolée;
- l'identificateur de la nature mathématique;
- une valeur pouvant servir, entre autre, à déconnecter les connecteurs (un degré de liberté en un nœud aura ainsi une valeur différente pour chacun des éléments comprenant ce nœud).

Dans le cas d'une fonction de forme linéaire interpolant la pression sur un triangle, tous les connecteurs sont des nœuds. Les identificateurs des nœuds seront donc utilisés comme identificateurs de connecteurs. La nature mahtématique est unique ici : il n'y a que des fonctions de forme linéaires. On utilise la valeur 1, par exemple, pour souligner la linéarité des fonctions de forme. Les trois parties de la clé présentées jusqu'à présent sont dépendantes de l'élément et sont suffisantes pour interpoler un champ continu. On mettrait alors la quatrième clé à une valeur constante pour tous les degrés de liberté (la fonction SetNat le fait automatiquement selon que l'on lui demande de générer les clés de degrés de liberté pour une quantité continue ou non). Si la grandeur physique n'est pas constante par élément, la quatrième partie clé peut être utilisée pour déconnecter les connecteurs d'un élément à l'autre : en utilisant, par exemple, l'identificateur de l'élément géométrique comme valeur de cette quatrième partie, la clé générée pour un

degré de liberté en un nœud pour une interpolation de même nature mathématique de la même quantité physique sera différente pour chacun des éléments auxquels le nœud appartient.

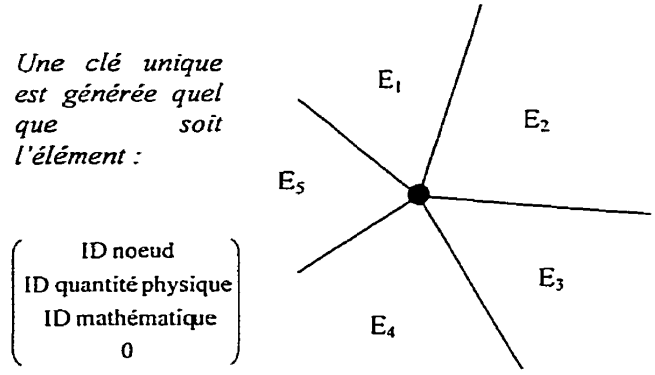


Figure 7.19 - Génération d'une clé pour un connecteur nodal dans le cas d'une quantité continue

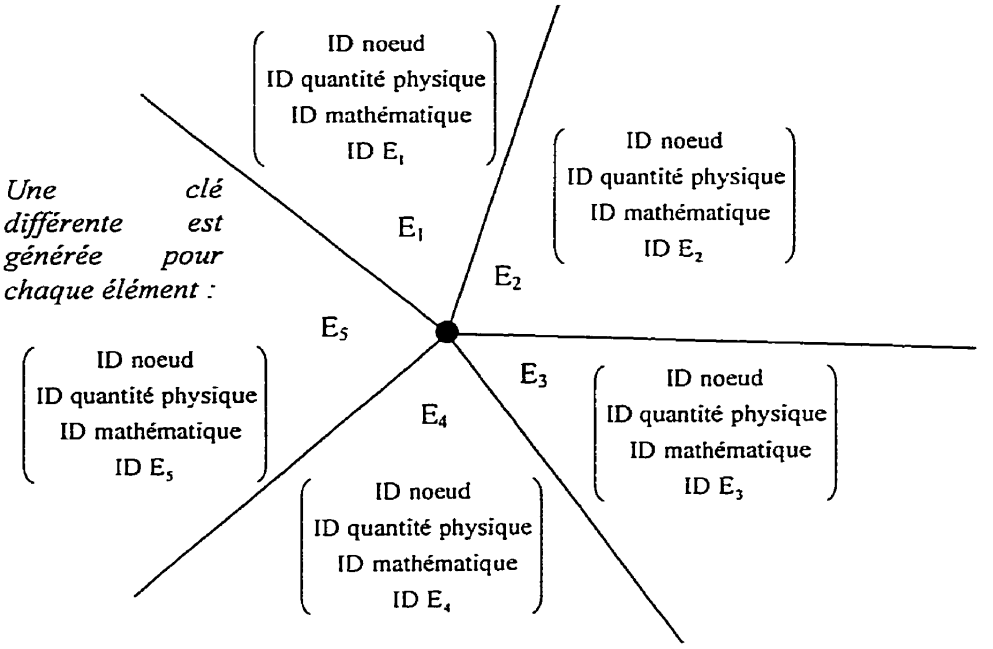


Figure 7.20 - Génération d'une clé pour un connecteur nodal dans le cas d'une quantité discontinue

Comme indiqué précédemment, deux clés sont nécessaires pour pleinement caractériser un degré de liberté. La première est générée avec la fonction de forme et indiquera la colonne de la matrice globale, la seconde est générée avec la fonction test et indiquera la ligne dans la matrice globale.

Le gestionnaire de degré de liberté enregistre au cours de l'assemblage tous les degrés de liberté dans un arbre équilibré, triant selon les valeurs des deux clés. Ceci permet de garantir un temps d'accès logarithmique aux degrés de liberté. C'est uniquement lors de l'étape de résolution que le gestionnaire crée la matrice globale et la garnit.

7.7.11.3 Gestion du temps

Comme l'on souhaite pouvoir résoudre des problèmes incluant des dérivées temporelles, il faut pouvoir conserver certains pas de temps dans le gestionnaire de degrés de liberté. Le gestionnaire stocke les valeurs pour plusieurs pas de temps d'un degré de liberté dans un vecteur. En fournissant un couple de clés pour la fonction de forme et la fonction test, on peut accèder à ce vecteur. Certaines fonctions d'accès retournent la valeur au pas de temps courant et sont habituellement utilisées. Certaines autres permettent de récupérer la valeur à un pas de temps donné. Ce système astucieux permet d'utiliser très facilement des schémas temporels variés : il suffit de spécifier quel schéma de différences finies on souhaite utiliser (schéma de Gear par exemple) et le gestionnaire récupère automatiquement les valeurs aux pas de temps précédents afin d'effectuer le calcul de la différence finie.

7.7.12 Quantités Physiques

Les quantités physiques constituent la dernières des familles de FEMView présentées plus en détail. Elles regroupent la définition :

- d'un gestionnaire de degrés de liberté;
- du code physique nécessaire lors de l'interaction avec le gestionnaire de degrés de liberté;
- de la fonction de forme à utiliser pour interpoler les valeurs aux degrés de liberté en n'importe quel point;
- les primitives d'exportation de résultat dans un format de fichier donné.

7.8 Paradigme d'implémentation

7.8.1 Choix du paradigme

Une fois terminée l'analyse points communs/variations des familles, il est temps de choisir un paradigme d'implémentation. Dans notre analyse, à l'étape de reconnaissance des familles, on a distingué des familles algorithmiques et des familles de données. Comme l'on développe un moteur de calcul, il est préférable que le paradigme utilisé ait été pensé pour être efficace, ce que n'est pas à priori le paradigme objet. L'approche générique a été pensée pour spécifier et implémenter efficacement les algorithmes. De plus, M. Alexander Stepanov a prouvé sa redoutable efficacité en obtenant avec sa STL du code aussi efficace que le code assembleur effectuant la même fonction. On ne peut certes pas affirmer avec certitude que l'utilisation de ce paradigme donnera des résultats aussi radicaux que ceux obtenus par M. Stepanov car :

- on ne dispose pas du temps nécessaire à la très fine optimisation qu'il a effectuée,
- comme on va le constater dans la section suivante, le paradigme générique est insuffisant pour décrire complètement la méthode des éléments finis.

C'est cependant ce paradigme, avec quelques aménagements, que l'on retient comme paradigme principal pour l'implémentation.

7.8.2 Programmation générique

La programmation générique consiste à séparer systématiquement les données et les algorithmes. C'est donc une approche radicalement distincte de l'approche objet qui les rassemble en une même classe. Les relations entre données et algorithmes sont définies en termes *d'itérateurs*. Ce sont des objets permettant de parcourir des *containeurs*, c.à.d

des collections d'objets. Les itérateurs sont l'objet d'une classification bi-dimensionnelle :

- sur la nature de l'accès : lecture, écriture, lecture-écriture;
- sur le mode de parcours : avant, arrière, avant/arrière, aléatoire.

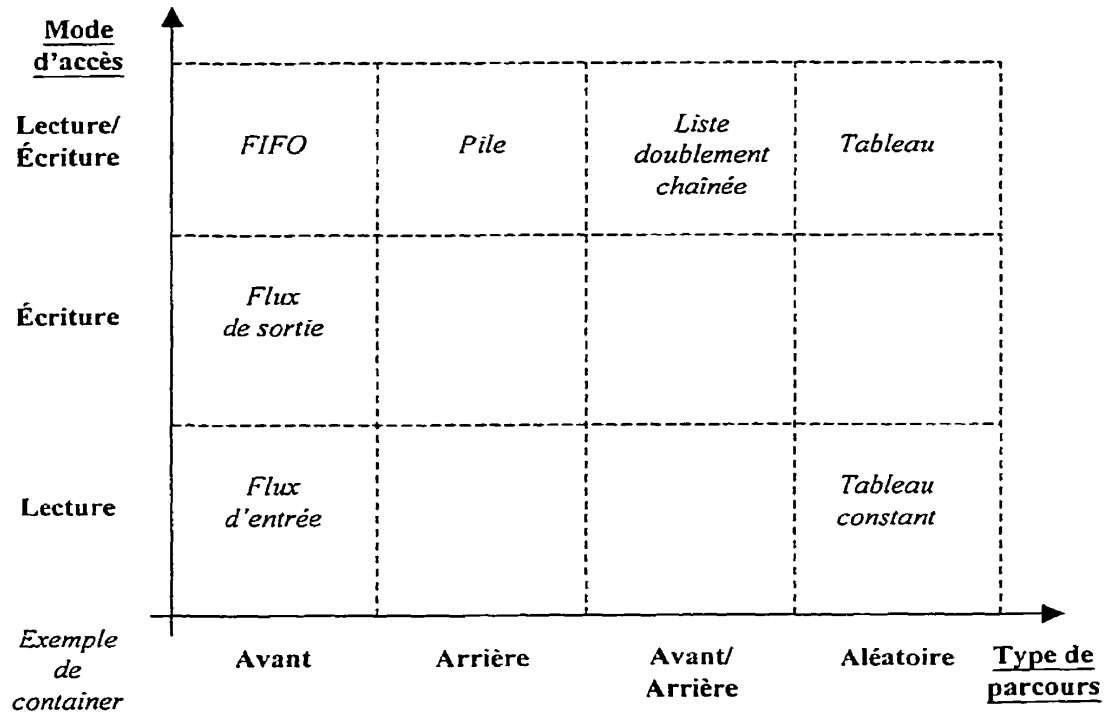


Figure 7.21 - Quelques structures de données correspondant à un mode d'accès et à un type de parcours

Cette classification permet de tenir compte élégamment, par des vérifications de types ou d'opérations supportées, l'adéquation entre un algorithme et des données.

Ce modèle est parfaitement suffisant pour des algorithmes ne nécessitant que peu de connaissances des données manipulées, comme un algorithme de tri par exemple.

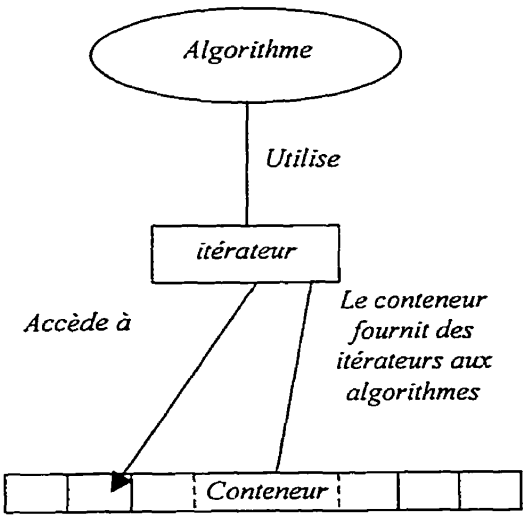


Figure 7.22 - Relations entre algorithmes, containers et itérateurs

La méthode des éléments finis par contre requiert que beaucoup plus d'informations soient mises à la disposition de l'algorithme. Une première façon de fournir cette information est d'utiliser le « patron » du visiteur. Dans ce patron, un objet supplémentaire est chargé de « visiter » l'objet à traiter. Il en extrait alors les information de type et les met à disposition de l'algorithme, lequel peux ainsi déterminer le comportement à adopter. Le problème de cette approche est qu'elle oblige à faire un code switch/case pour tenir compte de tous les cas possibles. La seconde manière de procéder consiste à utiliser des interfaces de « double-dispatch ». Comme cette approche produit un code plus rapide, ne contenant pas de switch/case et devenant ainsi moins facilement opaque, c'est celle qui est retenue pour l'implémentation. On la décrit dans la section suivante, après avoir introduit une autre notion utile : celle de classes homomorphes.

7.8.3 Adapter la programmation générique

7.8.3.1 Classes homomorphes

Les classes homomorphes sont un outil permettant de structurer un ensemble de classes par l'héritage de manière à ce que chacun de ses membres soit interchangeable. On obtient des classes homomorphes en spécifiant une interface de fonctions virtuelles pures dans la classe de base et surtout, en s'interdisant d'étendre l'interface des fonctions dérivées avec d'autres fonctions. Que signifierait en effet l'ajout de fonctions? D'une part, plus de fonctionnalités dans la classe en question. Mais, d'autre part, que l'on risque de ne plus avoir une parfaite interchangeabilité entre une instance de cette classe et une instance d'une autre classe de la même hiérarchie de classes homomorphes. Hors, la définition d'une telle hiérarchie ayant justement pour but de rendre les classes interchangeables, on s'interdit toute extension.

Toutes les familles comportant plusieurs classes sont implémentées sous la forme de hiérarchies de classes homomorphes.

7.8.3.2 Double Dispatch

7.8.3.2.1 Présentation

L'évaluation d'une fonction de forme est un bon exemple de l'inadéquation de la programmation générique dans certains cas. Cette évaluation nécessite en effet de connaître le type réel de la fonction de forme ainsi que le type réel de l'élément géométrique. Comme les fonctions de formes et les éléments géométriques sont implémentés sous la forme de hiérarchies de classes homomorphes, une instance de chacun d'entre eux est manipulé comme un pointeur sur le type de base, sans fournir plus d'information. Le mécanisme standard de fonction virtuelle permet de lever l'indétermination de type pour uniquement un des types. Il est donc nécessaire de le compléter.

Le mécanisme de « double dispatch » consiste à utiliser non pas une seule, mais deux interfaces de fonctions virtuelles : une dans chaque hiérarchie de classes homomorphes. Un appel à la première interface résout une des deux indéterminations de type (étape 2). Dans la fonction appelée, on invoque alors l'interface de la seconde hiérarchie de classes homomorphes (étape 2). Cette interface est surchargée pour chacun des types de la première famille. Ainsi, le mécanisme de résolution de surcharge permet de conserver l'information obtenue par la résolution du premier type. Le mécanisme de fonction virtuelle permet alors, par la détermination du type réel de l'instance d'une classe de la seconde hiérarchie, de lever la seconde indétermination (étape 3).

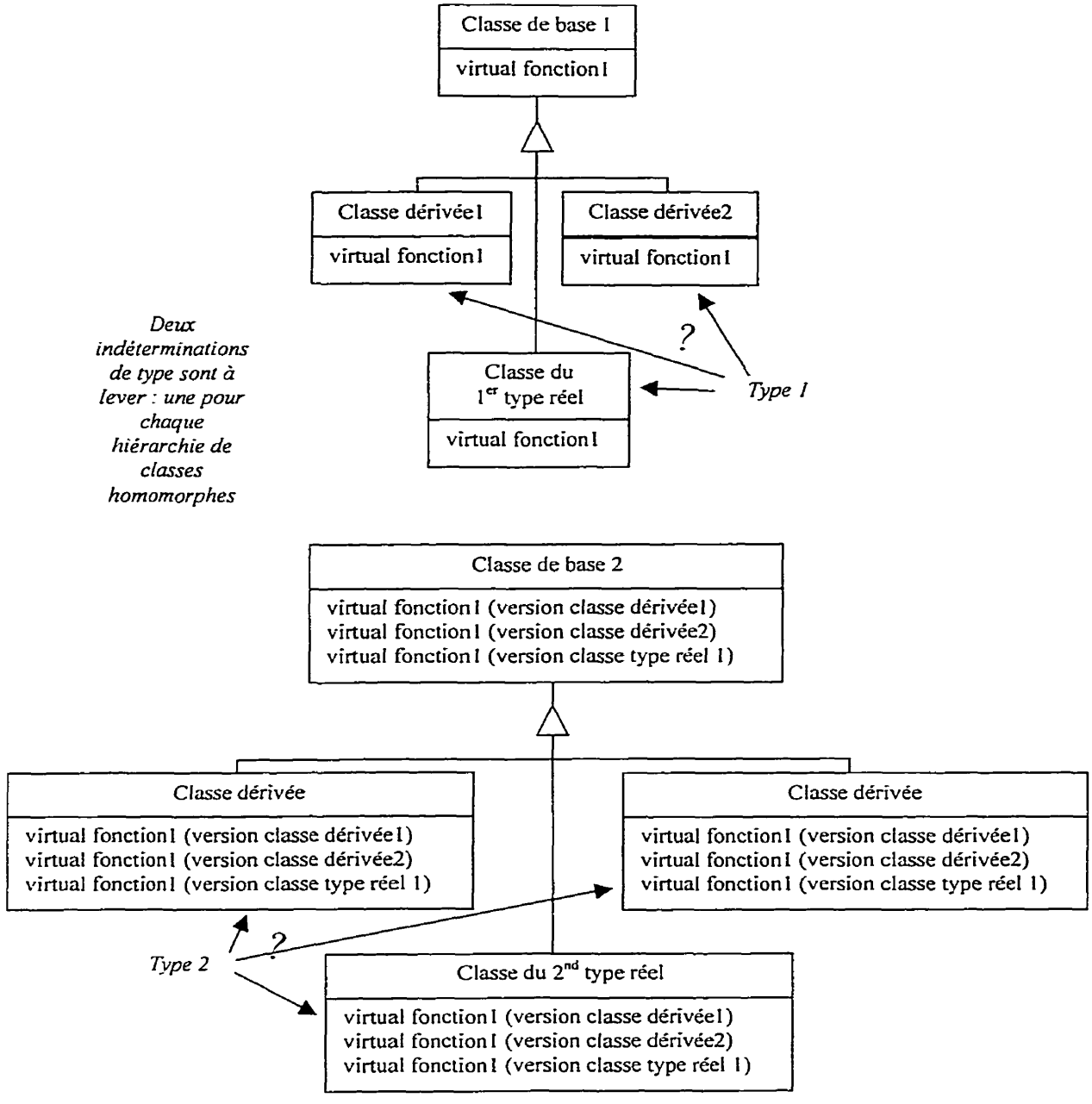


Figure 7.23 - Mécanisme de double dispatch – étape 1

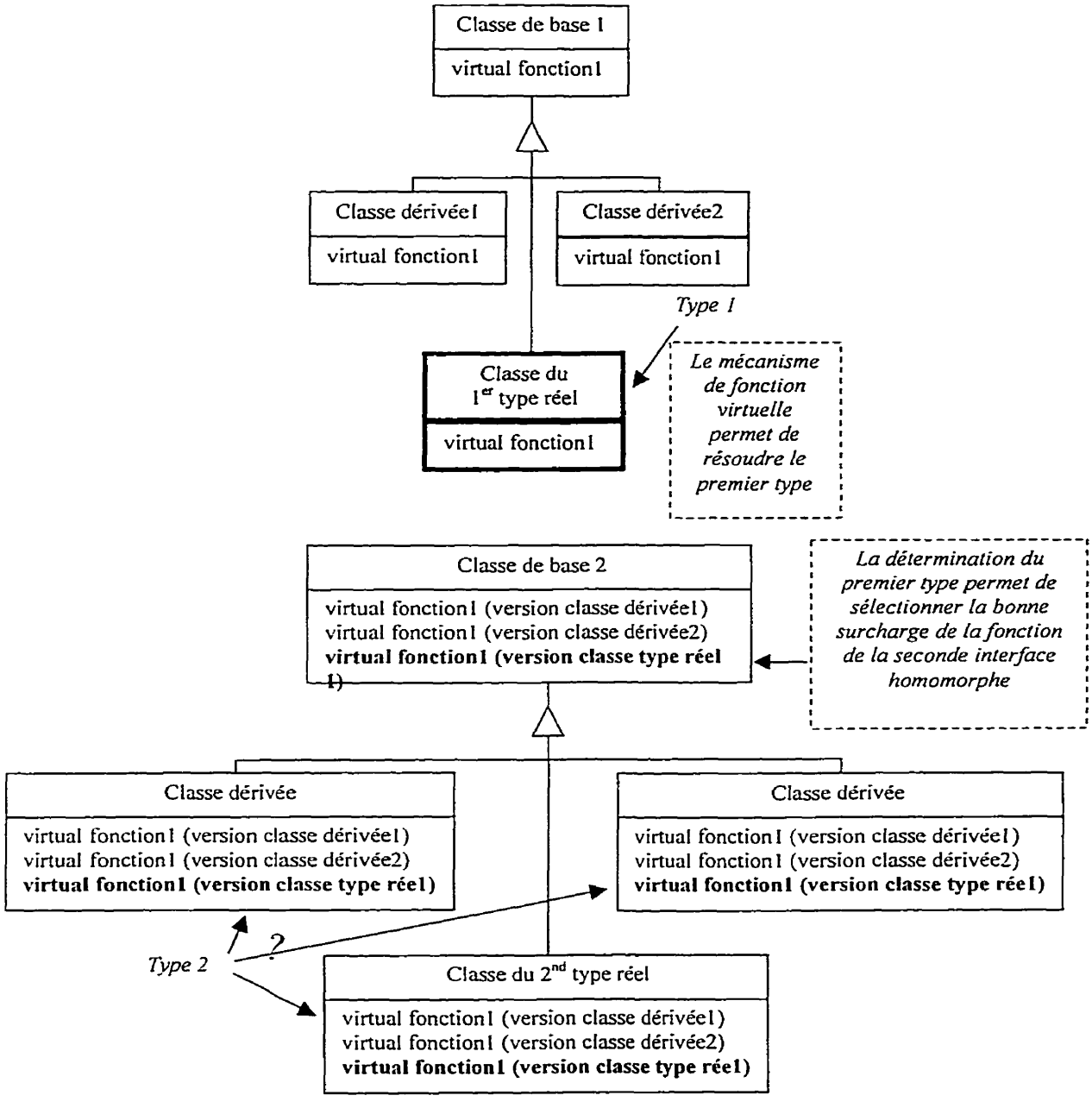


Figure 7.24 - Mécanisme de double dispatch – étape 2

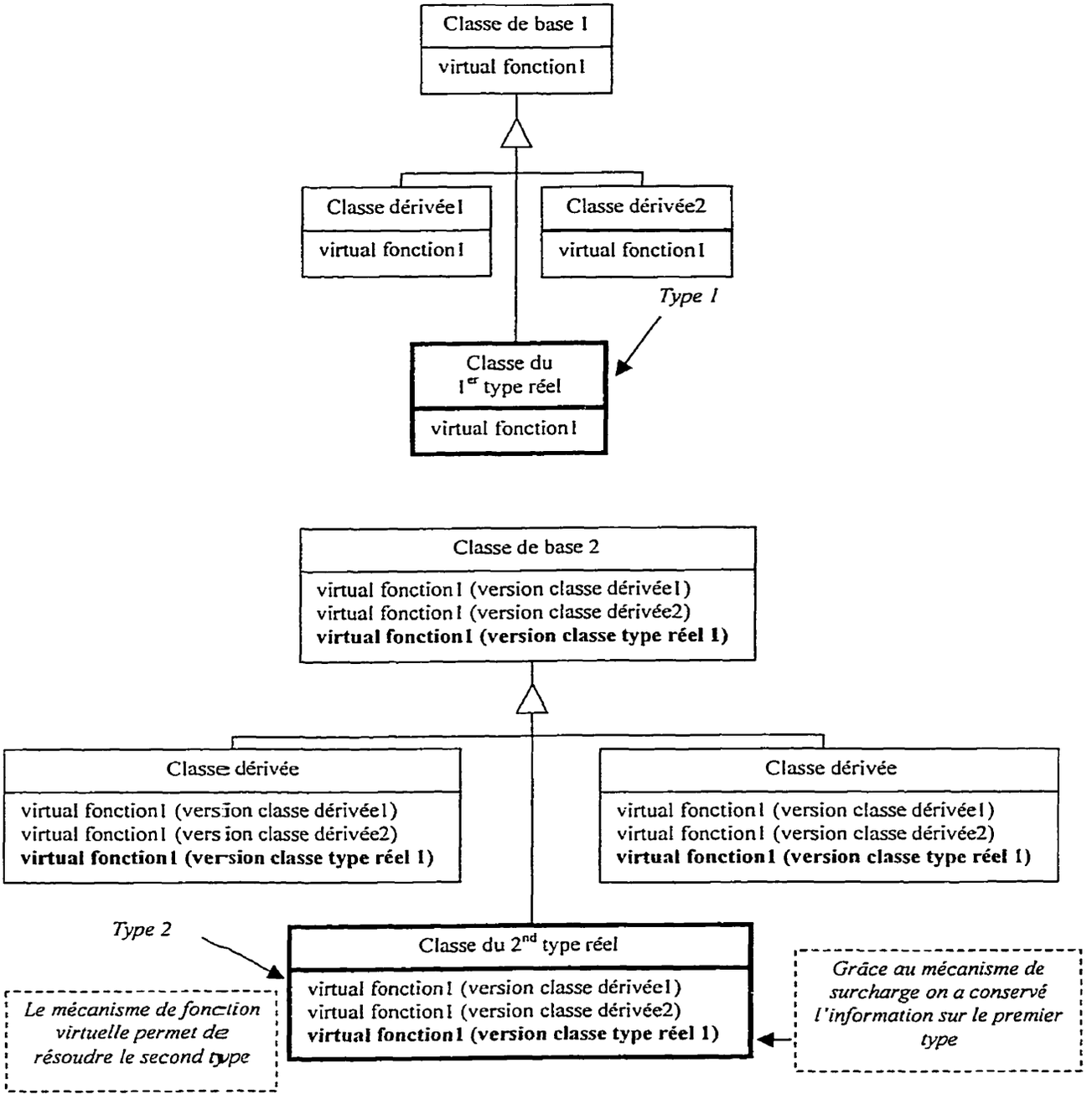


Figure 7.25 - Mécanisme de double dispatch – étape 3

Dans le cas de l'évaluation d'une fonction de forme sur un élément, les deux hiérarchies seraient celle des éléments géométriques et celle des fonctions de forme. La première serait celle des éléments géométriques car ainsi, c'est la fonction membre de la classe de fonction de forme qui détient toute l'information de type lors de son appel et est donc à même d'évaluer la fonction en un point. Inverser les rôles serait revenu à placer toutes les fonctions de forme dans l'élément géométrique, ce qui serait stupide.

7.8.3.2.2 Inconvénient des interface de double dispatch

L'inconvénient majeur des interfaces de double dispatch et plus généralement des interfaces de multiple dispatch est que, lorsque leur utilisation n'est pas maîtrisée elles entraînent une prolifération des fonctions membres dans les deux interfaces. Cependant, dans le cas présent (comme dans tous les autres cas d'utilisation dans FEMView), le nombre de fonctions généré n'est absolument pas abusif : il est parfaitement justifié de disposer dans une classe de fonction de forme d'une fonction membre pour chaque type d'élément.

7.8.3.2.3 Avantages

Les interfaces de double dispatch sont plus rapides que les switch/case, lorsque l'héritage est uniquement simple dans les deux familles prenant partie à la définition de l'interface. La résolution des deux types par ce moyen n'implique en effet lors de l'exécution que d'effectuer deux décalages dans un tableau.

Le second avantage des interfaces de double dispatch est qu'elles permettent de faire une liaison différée. Cela signifie que la « liaison » entre l'élément géométrique et la fonction de forme n'est effectuée que lors de l'exécution. Il est parfaitement possible d'imaginer un choix interactif de fonction de forme dans une portion du programme : le reste du code utilisant la liaison différée, le type de fonction de forme sera automatiquement et adéquatement traité. Comme une évolution possible de FEMView

après cette thèse est l'adjonction d'un langage de script, on mesure toute l'importance de la liaison différée.

7.9 Schéma global

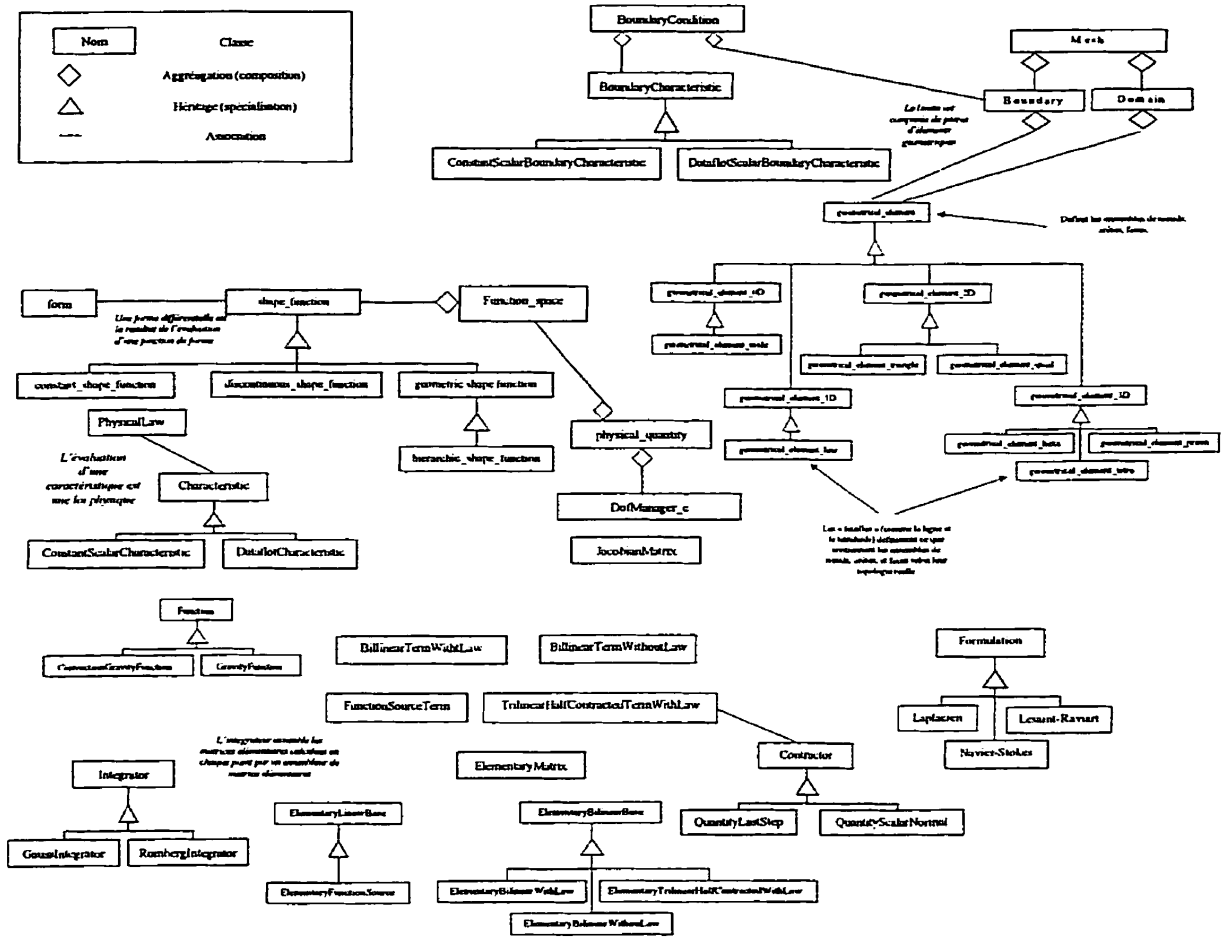


Figure 7.26 – Schéma global

Chapitre 8 EXEMPLES SIMPLES

8.1 Introduction

Les exemples simples détaillent quelques unes des tâches les plus courantes intervenant dans la méthode des éléments finis. Comme on ne veut présenter que l'essentiel du déroulement de ces tâches simples, on omet parfois délibérément certaines étapes, comme des initialisations ou déclarations, non indispensables à la compréhension.

8.2 Évaluation d'une fonction de forme

Comme expliqué dans les parties précédentes, on utilise une interface de double dispatch pour évaluer les fonctions de forme. Dans l'exemple suivant, on évalue une fonction de forme hiérarchique sur l'élément géométrique courant:

```
1) hierarchic_shape_function hsfCurrent(2);
2) lpgeoelemCurrent->GetF(hsfCurrent,u,v,w,iNthSF,formRetvalue);
```

On commence par instancier la fonction de forme hiérarchique. On précise en paramètre le degré d'interpolation désiré lors de l'instanciation de la fonction de forme.

On effectue ensuite l'évaluation de la fonction de forme proprement dite. On appelle pour cela la fonction GetF de la partie de l'interface de double dispatch contenue dans la classe d'élément géométrique. Ce premier appel permet de déterminer le type d'élément géométrique. Dans le corps de cette méthode, on invoque la fonction GetF membre de la classe de fonction de forme. Grâce à l'utilisation de la surcharge de fonctions et du mécanisme de fonctions virtuelles, on peut conserver l'information du type d'élément

géométrique (déterminé lors du premier appel) et déterminer le type réel de la fonction de forme. Le code de la fonction GetF de la fonction de forme ressemble typiquement à :

```
void GeomShapeFunction::GetF(const geometrical_element_triangle&,
double& u , double& v , double& w,int iNth,form& retvalue) const{

      retvalue.SetNat(0,false);

      switch(iNth) {

            case 1  : retvalue[0] = 1.-u-v ; break ;

            case 2  : retvalue[0] =    u   ; break ;

            case 3  : retvalue[0] =      v ; break ;

            case 4  :  retvalue[0] = u-u*u-v*u;  break ;

            case 5  :  retvalue[0] = v*u;  break ;

            case 6  :  retvalue[0] = v-v*u-v*v;  break ;

            case 7  :  retvalue[0] = 3.0*u*u-u+2.0*v*u-2.0*u*u*u-
            3.0*v*u*u-v*v*u;break;

            case 8  :  retvalue[0] = v*v*u-v*u*u;  break ;

            case 9  :  retvalue[0] = v-2.0*v*u-
            3.0*v*v+v*u*u+3.0*v*v*u+2.0*v*v*v;break;

            case 10 :  retvalue[0] = v*u-v*u*u-v*v*u;break;

            default : retvalue[0] = 0.; break ;

      }

}
```

(La fonction est surchargée pour chaque type d'élément géométrique.)

On commence par contrôler que la forme différentielle utilisée pour stocker la valeur de retour est correctement initialisée par l'appel à la fonction `SetNat` de la classe de forme différentielle. On laisse l'initialisation du type correct de la forme à la discrétion de la fonction de forme. C'est en effet elle et elle seule qui connaît le type de sa valeur de retour. Dans le cas où la forme était déjà initialisée avec un type donné, une erreur est

générée lors de la tentative d'initialisation à un type différent. On effectue ensuite un switch/case sur l'indice de la fonction de forme à évaluer afin de déterminer la bonne formule à utiliser pour calculer la forme différentielle au point de l'espace de référence spécifié.

Une autre fonction de l'interface de double dispatch, `GetNbFF`, permet d'obtenir le nombre de fonctions de forme de la base d'interpolation. La correspondance entre l'index de la fonction de forme dans la base et le connecteur géométrique est ensuite établie lors de la génération des clés d'interaction par la fonction `SetNat`, également membre de l'interface de double dispatch. On évalue en effet généralement toutes les fonctions de forme et toutes les fonctions tests pour calculer la matrice élémentaire d'un élément en un point. La génération des clés par `SetNat` permet de relier les entrées de la matrice élémentaire à une position dans la matrice globale.

On remarque que la fonction `GetF` ne retourne que l'évaluation directe de la fonction de forme. L'évaluation de la dérivée se fait de manière similaire, mais par le biais de la fonction `GetdF`.

On avait évoqué lors de la description des formes différentielles l'existence de formes différentielles multiples. Elles sont évaluées par les fonctions `GetMF` et `GetMdF`, de manière similaire aux deux précédentes. On spécifie cependant un paramètre additionnel : la composante de la forme multiple à évaluer.

Comme on évalue généralement toutes les fonctions de forme de la base, et afin de cacher l'existence de quatre fonctions différentes à utiliser selon que la forme soit multiple ou non et que l'on évalue directement la base ou sa dérivée, on fournit la fonction `GetFConnectors`. Elle retourne un vecteur STL contenant les formes différentielles résultant de l'évaluation de toutes les fonctions de la base, en prenant soin d'évaluer composante par composante et de prendre la dérivée extérieure s'il y a lieu. Comme toutes les fonctions précédentes elle est invoquée de manière similaire à `GetF`.

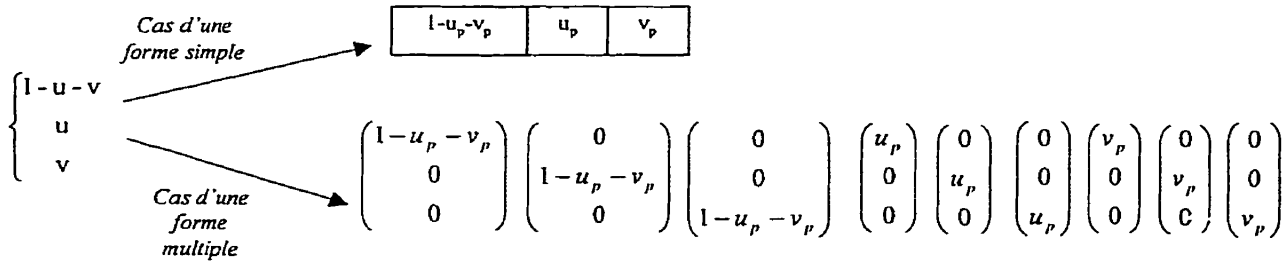


Figure 8.1 - Vecteur de formes généré par GetFConnectors

On remarque que les fonctions de formes sont évaluées pour chacune des composantes dans le cas des formes multiples. Cette décomposition est nécessaire pour pouvoir utiliser des *adapteurs*.

8.3 Exemple d'utilisation d'un adapteur : évaluation de la divergence d'une vitesse

Les adapteurs sont une famille d'objets effectuant des traitements sur les formes différentielles évaluées pas les fonctions de la base d'interpolation. Ils sont automatiquement appelés par la fonction `GetFConnectors`.

Supposons que l'on doive, par exemple, évaluer $div(\vec{V})$. Le problème est que $\vec{V}$ est une 3*0-forme et que son opérateur de dérivation extérieur est donc le gradient. En fait, une loi de comportement a forcément été simplifiée pour en arriver à une telle expression, comme dans le cas de Navier-Stokes en incompressible. La première solution serait donc d'utiliser une loi de comportement unitaire. On regagnerait alors la cohérence mathématique. Cependant, la multiplication par la loi physique superflue peut être évitée: il suffit de prendre la trace du gradient des 3*1-formes évaluées par les fonctions

de la base d'interpolation. Rappelons qu'une 3*1-forme est décomposée en 3 3*1-formes lors de son évaluation:

$$\begin{pmatrix} V_{11} & V_{12} & V_{13} \\ V_{21} & V_{22} & V_{23} \\ V_{31} & V_{32} & V_{33} \end{pmatrix} = \begin{pmatrix} V_{11} & 0 & 0 \\ V_{21} & 0 & 0 \\ V_{31} & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & V_{12} & 0 \\ 0 & V_{22} & 0 \\ 0 & V_{32} & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & V_{13} \\ 0 & 0 & V_{23} \\ 0 & 0 & V_{33} \end{pmatrix}$$

En prenant la trace de chacune de ces formes on assemble bien la divergence de la vitesse comme illustre le schéma suivant.

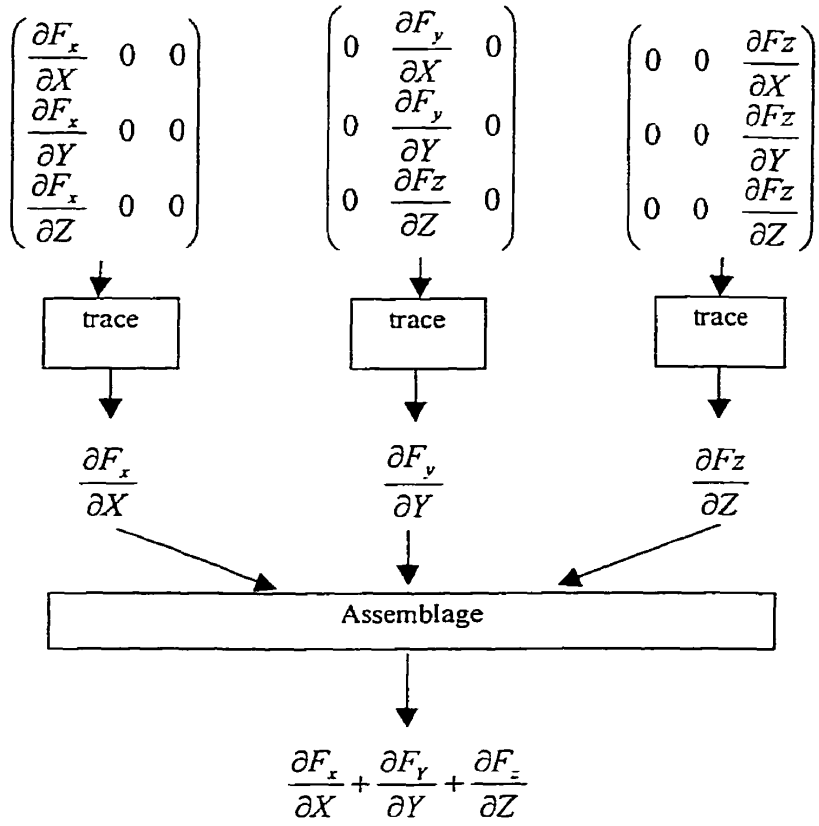


Figure 8.2 - Décomposition de l'évaluation de la divergence d'une vitesse à l'aide de l'adapteur de trace

On obtient ainsi une solution alliant élégance et efficacité. D'autres adapteurs sont définis tels que l'adapteur de partie symétrique et celui de partie antisymétrique. Ils permettent notamment d'obtenir la partie rotation pure et la partie déformation pure du tenseur de déformation.

L'utilisation d'un adapteur est très simple et peut être effectuée en 2 étapes. La première est l'instanciation de l'adapteur. La seconde consiste à passer l'adapteur en troisième paramètre au constructeur d'une fonction de forme. On rappelle que le premier

paramètre est l'ordre d'interpolation désiré et que le second indique si la forme différentielle interpolée est multiple ou non.

```
1) symetric_part spCurrent;
2) hierarchic_shape_function hsfCurrent(2,true,spCurrent);
```

La fonction de forme est utilisée par la suite comme n'importe quelle autre fonction de forme. L'utilisation d'un adapteur est complètement encapsulée. Ce n'est que lors de l'appel de la fonction `GetFConnectors` que l'adapteur, s'il y en a un, sera appelé. Cela signifie que, si l'on appelle directement les fonctions GetF, GetdF, GetMF ou encore GetMdF, l'adapteur ne sera pas automatiquement utilisé. Cela n'est pas gênant car ces fonctions ne sont jamais directement invoquées, sauf dans des cas très spécifiques, où l'utilisateur sait exactement ce qu'il fait. Il lui est toujours possible de demander quel est l'adapteur courant d'une fonction de forme et de l'appliquer « manuellement » si besoin est.

8.4 Calcul d'une matrice élémentaire en un point d'intégration

Le calcul d'une matrice élémentaire en un point est effectué par un assembleur de matrice élémentaire. Il reçoit en paramètres les données suivantes :

- les coordonnées du point de l'espace de référence où calculer
- l'élément géométrique sur lequel calculer
- la matrice jacobienne de changement de coordonnées
- la loi physique à utiliser (s'il n'y a pas de loi physique, on ne se sert tout simplement pas de cette donnée; on procède ainsi afin d'avoir une interface unifiée, que le terme prenne une loi physique ou non)

- la matrice élémentaire où ajouter la contribution du point

L'assembleur a en attribut l'espace fonctionnel courant.

Le calcul est ensuite effectué de la manière suivante :

- évaluation de la base de fonctions de forme au point courant;
- évaluation de la base de fonctions test au point courant;
- calcul et ajout à chaque entrée de la matrice élémentaire du terme bilinéaire évalué pour la fonction de forme correspondant à la colonne et de la fonction test correspondant à la ligne.

Ceci est valable dans le cas d'un terme bilinéaire.

Dans le cas d'un terme linéaire, on évalue uniquement la base de fonctions test. Dans le cas d'un terme trilinéaire, on effectue la contraction soit des formes résultant de l'évaluation des fonctions de forme, soit des formes résultant de l'évaluation des fonctions test, comme décrit lors de la présentation des fonctions de forme.

8.5 Intégration sur un élément

L'intégration d'un terme sur un élément est demandée par le terme à l'intégrateur qu'on lui a assigné pour chacun des éléments du domaine (ou du bord) sur lequel il doit s'assembler. L'élément géométrique courant est ici nommé `geoelemCurrent`. Il est également nécessaire de spécifier l'espace fonctionnel courant (`fsdCurrent`) et la loi physique (`plCurrent`) s'il y a lieu. S'il n'y a pas de loi physique, on passe `NULL` en lieu et place de `&plCurrent`. Comme la loi physique n'est de toute façon utilisée que par

l'assembleur de matrice élémentaire, et comme ce dernier est choisi par le terme, on n'introduit pas ici de risque.

1. `integratorCurrent.SetCurrentElement(&geoelemCurrent);`
   Spécification de l'élément géométrique courant à l'intégrateur.

2. `integratorCurrent.SetCoordChangeElement(&geoelemCurrent);`
   Spécification de l'élément géométrique à utiliser pour calculer le changement de coordonnées.

3. `integratorCurrent.SetFunctionSpace(&fsdCurrent);`
   Spécification de l'espace fonctionnel courant à l'intégrateur.

4. `integratorCurrent.SetIntegrationPoints();`
   Demande à l'intégrateur de déterminer automatiquement les points d'intégration selon le type de l'élément et la nature de l'espace fonctionnel. L'intégrateur calcule pour cela le degré des fonctions à intégrer en additionnant les degrés des fonctions de forme et test, puis en retranchant un à la somme pour chaque dérivée extérieure prise sur les fontions de forme ou test.

5. `integratorCurrent.AssembleBilinearMatrix(&plCurrent,&ele_bilinear_with_law,&integresRetValue);`
   Enfin, on demande l'intégration du terme sur l'élément. On spécifie ici un assembleur de matrice élémentaire. (`ele_bilinear_with_law`). On rappelle que les assembleurs de matrices élémentaires sont responsables du calcul en un point. L'assembleur de matrice élémentaire est donc appelé en chaque point d'intégration et ajoute la contribution de chacun de ces points à la matrice élémentaire `integresRetValue`.

L'intégrateur ne s'occupe que du calcul de la matrice élémentaire. Son assemblage dans la matrice global est effectué par le terme.

L'assemblage d'un terme de bord est tout à fait similaire. Seule la quatrième étape change : au lieu d'appeler la fonction `SetIntegrationPoints`, on appelle la fonction `SetBoundaryIntegrationPoints`.

8.6 Assemblage d'une matrice élémentaire

L'assemblage d'une matrice élémentaire dans le gestionnaire de degrés de liberté est effectué en trois étapes :

1.
```
geoelemCurrent.SetNat(fsdCurrent.GetShapeBase(),
fsdCurrent.IsShapedF(),
lppqShape->GetCode(),
lppqShape->Disconnected(),
&vecdnConnToAddToDofManager,
&veciConnToAddToDofManager,
&veciDisconnectorsConnToAddToDofManager);
```

   Génération des clés d'interaction de la fonction de forme, obtenue par l'appel `fsdCurrent.GetShapeBase()`. D'autres informations sont également nécessaires à la génération de ces clés : si l'on évalue la dérivée de la base d'interpolation (`fsdCurrent.IsShapedF()`), le code de la quantité physique (`lppqShape->GetCode()`), si la quantité physique est déconnectée élément par élément (`lppqShape->Disconnected()`) (on rappelle que cela est important pour la génération de la $4^{ième}$ partie de chaque clé d'interaction). Les trois vecteurs suivants stockent des parties de la clé. Le premier vecteur stocke les natures mathématiques et physiques, le second vecteur les natures géométriques (les connecteurs) et le troisième les clés de déconnexion. On sépare les clés de cette façon car le gestionnaire de degrés de liberté les demande ainsi.

2.
```
geoelemCurrent.SetNat(fsdCurrent.GetTestBase(),
fsdCurrent.IsTestdF(),
lppqTest->GetCode(),
lppqTest->Disconnected(),
&vecdnEqToAddToDofManager,
&veciEqToAddToDofManager,
&veciDisconnectorsEqToAddToDofManager);
```

Génération des clés d'interaction de la fonction test. Similaire à la génération des clés de la fonction de forme.

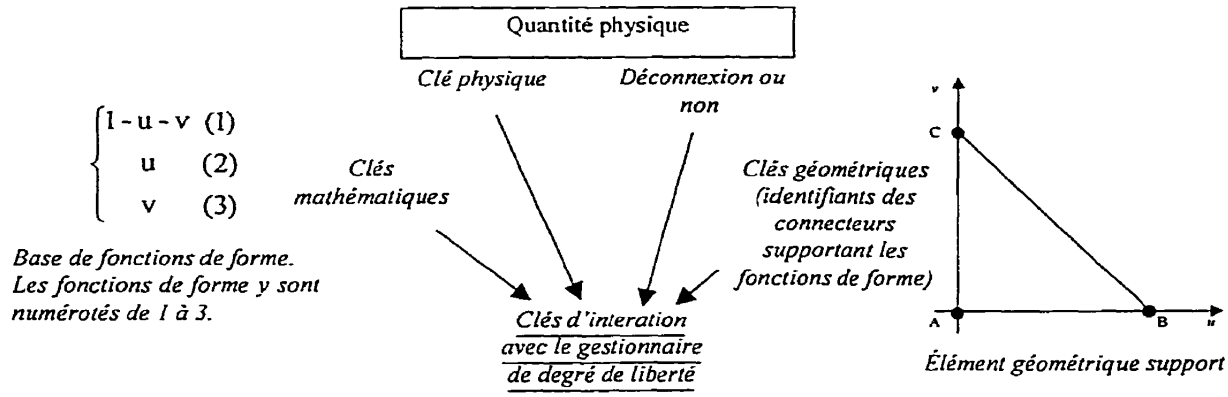


Figure 8.3 - Composantes d'une clé d'interaction avec le gestionnaire de degrés de liberté

(on génère une clé par fonction de forme)

3.
```
iCellToAssemble = 0;

for(j=0;j<vecdnConnToAddToDofManager.size();j++){
     for(k=0;k<vecdnEqToAddToDofManager.size();k++){
          lpDofData->AssembleTerm(vecdnEqToAddToDofManager[k],
                    veciEqToAddToDofManager[k],
                    veciDisconnectorsEqToAddToDofManager[k],
                    vecdnConnToAddToDofManager[j],
                    veciConnToAddToDofManager[j],
                    veciDisconnectorsConnToAddToDofManager[j],
                    integresRetValue[iCellToAssemble++]);
     }
}
```

Assemblage des valeurs de la matrice élémentaire (`integresRetvalue`) dans le gestionnaire de degrés de liberté. Pour chaque valeur on passe la clé de la fonction test correspondant à la ligne et la clé de la fonction de forme correspondant à la colonne.

8.7 Itération sur une collection

8.7.1 Description

On présente ici la syntaxe permettant d'itérer sur tous les objets contenus dans un conteneur.Afin de pouvoir parcourir tous les éléments de la collection, il est nécessaire de connaître :

- le premier élément
- la fin de la collection
- le moyen de passer à un autre élément (le plus souvent l'élément suivant) depuis l'élément courant

FEMView retient pour cela la syntaxe de la STL et l'utilisation d'itérateurs ou de pointeurs dans les cas les plus simples. Dans la STL, un itérateur est une sorte de pointeur généralisé. Il indique la position d'un objet et dispose de fonctionnalités permettant de passer, selon le type d'itérateur, à l'élément suivant, l'élément précédent ou encore un élément quelconque. On obtient le premier et la fin de la collection par le biais d'itérateurs :

```
type_collection::iterator begin,end;
begin = instance_collection.begin();
end = instance_collection.end();
```

Pour itérer sur toute la collection, on utilise habituellement la syntaxe suivante :

```
for(;begin != end;begin++)
      /*action où l'on désigne l'élément par *begin*/
```

La surcharge de l'opérateur « ++ » fait pointer l'itérateur sur l'élément suivant l'élément courant. Ainsi, on parcourt tous les éléments jusqu'à atteindre la fin de la collection.

8.7.2 Exemple : assemblage d'un terme sur un domaine

On a décrit dans les sous-sections précédentes comment intégrer un terme sur un élément géométrique et comment assembler la matrice élémentaire dans la matrice globale. Un terme ne fait rien de plus, excepté itérer sur tous les éléments d'un domaine (ou d'un bord) :

```
Domain::iterator itBegin,itEnd;
itBegin = domSupport.begin();
itEnd = domSupport.end();
for(;itBegin != itEnd;itBegin++)
      /*calcul et assemblage de la matrice de l'élément désigné par
itBegin*/
```

Dans le fragment de code ci-dessus, `domSupport` est le domaine sur lequel le terme est assemblé. Un domaine est défini dans FEMView comme étant un conteneur d'éléments géométriques (et un peu plus que cela...). On permet l'itération sur les éléments géométriques contenus par le mécanisme standard de la STL :

- définition d'une fonction membre begin retournant un itérateur sur le premier élément géométrique de la collection

- définition d'une fonction membre end retournant un itérateur sur la fin de la collection

De plus, on se sert d'un conteneur standard de la STL pour stocker les éléments géométriques dans la classe domaine. Ainsi, on a juste à retourner les itérateurs de ce

conteneur, le reste (parcours de la collection par les itérateurs) étant implémenté par les itérateurs de la STL.

On utilise ce mécanisme en de nombreuses autres occasions, comme pour parcourir les points d'intégration par exemple.

8.8 Imposition d'une condition aux limites

FEMView offre, mais n'oblige pas à utiliser, une imposition automatisée des conditions aux limites. Elle est effectuée par l'appel de la fonction dans le corps de la méthode `TreatmentOfFormulation` d'une formulation :

```
TreatmentOfGroups(meshCurrent,bcnrCurrent,veclppqCurrent[0],((DofManage
r_c*&)*(veclppqCurrent[0])),0);
```

On lui passe en paramètre :

- le maillage courant (`meshCurrent`);
- le registre des conditions aux limites (l'objet contenant toutes les conditions aux limites) (`bcnrCurrent`);
- la quantité physique pour laquelle on impose la condition (`veclppqCurrent[0]`);
- le gestionnaire de degrés de liberté où imposer les conditions concernant cette quantité physique `((DofManager_c*&)*(veclppqCurrent[0]))`;
- le type de condition à imposer (`0` pour une condition de Dirichlet, `1` pour une condition de Robin).

On remarque que l'on ne se sert pas obligatoirement du gestionnaire de degrés de liberté spécifié dans la définition de la quantité physique. On agit ainsi pour obtenir plus de souplesse.

Le corps de la fonction `TreatmentOfGroups` effectue une détermination de la fonction à appeler selon le type de la condition. On appelle `DirichletBoundaryCondition` pour imposer une condition de `Dirichlet` et `RobinBoundaryCondition` pour imposer une condition de Robin.

Une condition de Dirichlet ne requiert que la fixation de degrés de liberté. Elle peut donc être imposée avant que l'ensemble des degrés de liberté du problème ne soit créé : le gestionnaire de degré de liberté crée automatiquement la ligne de la matrice globale nécessaire.

L'assemblage de la condition de Robin par contre exige l'assemblage d'un terme dans la matrice globale. Du fait du fonctionnement du gestionnaire de degrés de liberté, il est donc nécessaire d'avoir préalablement créé les degrés de liberté du problème. C'est pourquoi l'imposition des conditions aux limites est généralement effectuée dans l'ordre suivant :

1)
```
TreatmentOfGroups(meshCurrent,bcnrCurrent,veclppqCurrent[0],((DofMan
ager_c*&)*(veclppqCurrent[0])),0);
```

   Imposition des conditions de Dirichlet.

2)
```
ActionOnElements(DEFDOF_ACTION,   meshCurrent,   veclppqCurrent[0],
meshCurrent.GetAllElements(),  (DofManager_c*&)*(veclppqCurrent[0]),
matregCurrent ,0) ;
```

   Création des degrés de liberté du problème, c'est à dire de la matrice et du second membre.

3)
```
((DofManager_c*&)*(veclppqCurrent[0]))->AllocMatrix();
```

   Allocation de la matrice globale.

4) `TreatmentOfGroups(meshCurrent,bcnrCurrent,veclppqCurrent[0],((DofManager_c*&)*(veclppqCurrent[0])),1);`

Imposition des conditions de Robin.

On peut imposer successivement les conditions de Dirichlet ou de Robin de plusieurs quantités.

Cette imposition systématique impose toutes les conditions aux limites d'un type mathématique donné pour une quantité physique donnée. Si l'on désire les imposer plus finement, il faut surcharger la fonction `TreatmentOfGroups`. Dans la version surchargée, il faut alors parcourir le registre de conditions aux limites et appeler directement les fonctions `DirichletBoundaryCondition` et `RobinBoundaryCondition`. Les conditions aux limites y sont stockées en fonction :

- du code de la quantité physique concernée
- du code du type mathématique de la condition
- du code de la région concernée par cette condition

On peut donc aisément retrouver une condition aux limites donnée. La condition aux limites contient également :

- la `BoundaryCharacteristic` à utiliser
- l'espace fonctionnel à utiliser

On utilise toutes ces données pour appeler `DirichletBoundaryCondition` (l'appel de `RobinBoundaryCondition` est similaire) :

```
DirichletBoundaryCondition (lpbcToUse,lpfsdToUse, iPhysCode,
bDisconnecter, lpdofAssemblyPlace, meshCurrent, iGeomId);
```

On lui passe (dans l'ordre) : la « boundary characteristic », l'espace fonctionnel, le code de la quantité physique , le gestionnaire de degrés de liberté où assembler, le maillage courant, le code de la zone où imposer la condition aux limites.

Chapitre 9 EXEMPLES COMPLETS

9.1 Introduction

On développe dans cette section des exemples de difficultés mathématiques et informatiques croissantes. On commence par un Laplacien, implémentable en quelques lignes, qui nous permet de montrer le fonctionnement d'une formulation simple. On enchaîne en bâtissant étape après étape deux formulations nous permettant de simuler la convection naturelle dans une cavité. Cet exemple permet de montrer l'utilisation d'un schéma non linéaire. Enfin, on termine avec la formulation de Lesaint-Raviart. Elle est complexe autant mathématiquement que dans son implémentation et requiert, entre autres, l'écriture d'un terme de bord particulier.

9.2 Laplacien

9.2.1 Développement mathématique et présentation

Cette section présente un exemple complet très simple mathématiquement : un Laplacien. Il nous permet de présenter une formulation ne comprenant pas de difficultés mathématiques. On peut ainsi se concentrer sur l'implémentation.

Soit le problème suivant :

$$\begin{cases} \nabla^2 f = 0 \\ f_{|\Gamma_a} = 1 \\ f_{|\Gamma_b} = 0 \end{cases}$$

Équation 1 - Laplacien

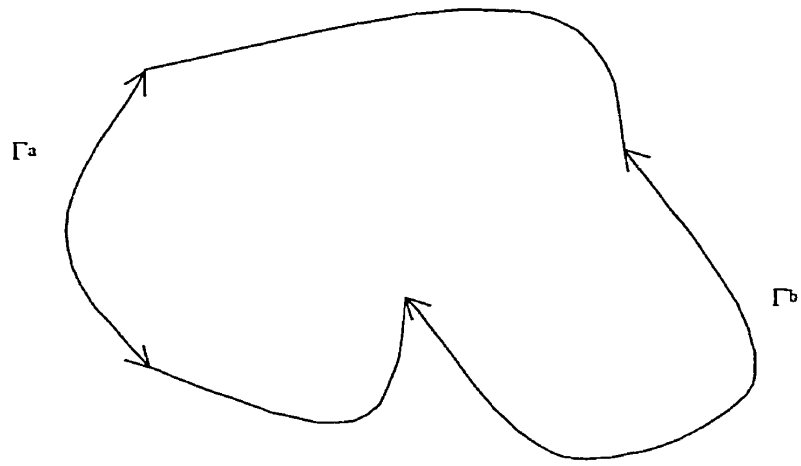


Figure 9.1 – Domaine sur lequel on résout le Laplacien

Afin de simplifier l'explication, on suppose que f est un potentiel scalaire (une 0-forme) comme la pression ou la température.

Multiplions par une fonction test f' et intégrons par parties.

$$\int_\Omega \nabla^2 f f' \, d\Omega = 0 = \int_\Omega \nabla f \nabla f' \, d\Omega - \int_\Gamma \frac{\partial f}{\partial \vec{n}} f' \, d\Gamma$$

On obtient ainsi une forme faible de l'équation. En appliquant les conditions aux limites, elle devient :

$$\int_{\Omega} \nabla f \nabla f' \, d\Omega = 0$$

Traduisons cette équation mathématique en entités de FEMView.

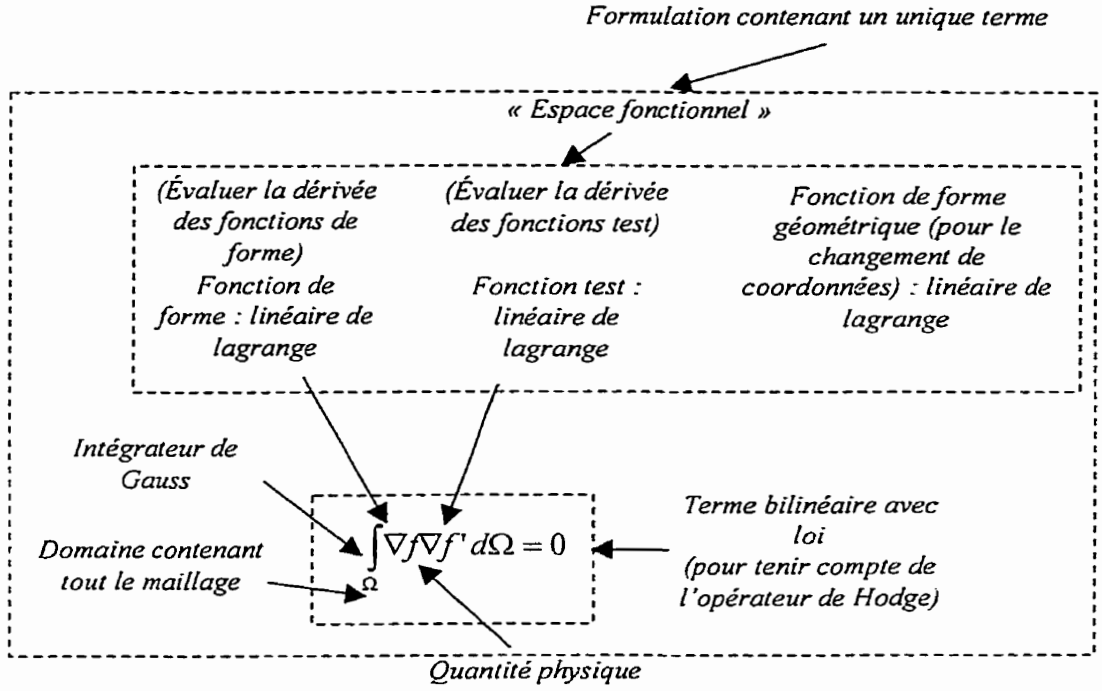


Figure 9.2 - Décomposition du Laplacien en entités de FEMView

Il y a une seule formulation : le Laplacien. On ne retrouvera donc qu'une seule classe de formulation. Son unique équation ne comporte qu'un seul terme bilinéaire, fourni par le canevas. Il faut cependant remarquer que dans ce terme l'opérateur de Hodge est sous-entendu. Comme il est automatiquement appliqué dans le cas d'un terme bilinéaire

comportant une loi de comportement, on utilisera un tel terme avec une loi unitaire et constante. Il n'y a pas besoin ici d'un espace fonctionnel particulier. On utilise de simples fonctions de forme linéaires de Lagrange comme fonctions de forme et fonctions test. Comme explicité dans la présentation des familles, un « espace fonctionnel » est utilisé et contient la définition des fonctions de forme, test et de forme géométrique ainsi que l'information indiquant si l'on doit évaluer directement les fonctions de forme (test) ou leurs dérivées extérieures. On utilise des éléments géométriques linéaires et on se sert donc de la même fonction de forme de Lagrange comme fonction de forme géométrique. On définit une quantité physique pour représenter f. Il ne nous manque plus qu'un intégrateur. On se sert d'un intégrateur de Gauss tout à fait usuel.

9.2.2 Formulation

Le diagramme suivant résume ce qui se passe lors de l'assemblage et la résolution de la formulation « Laplacian ».

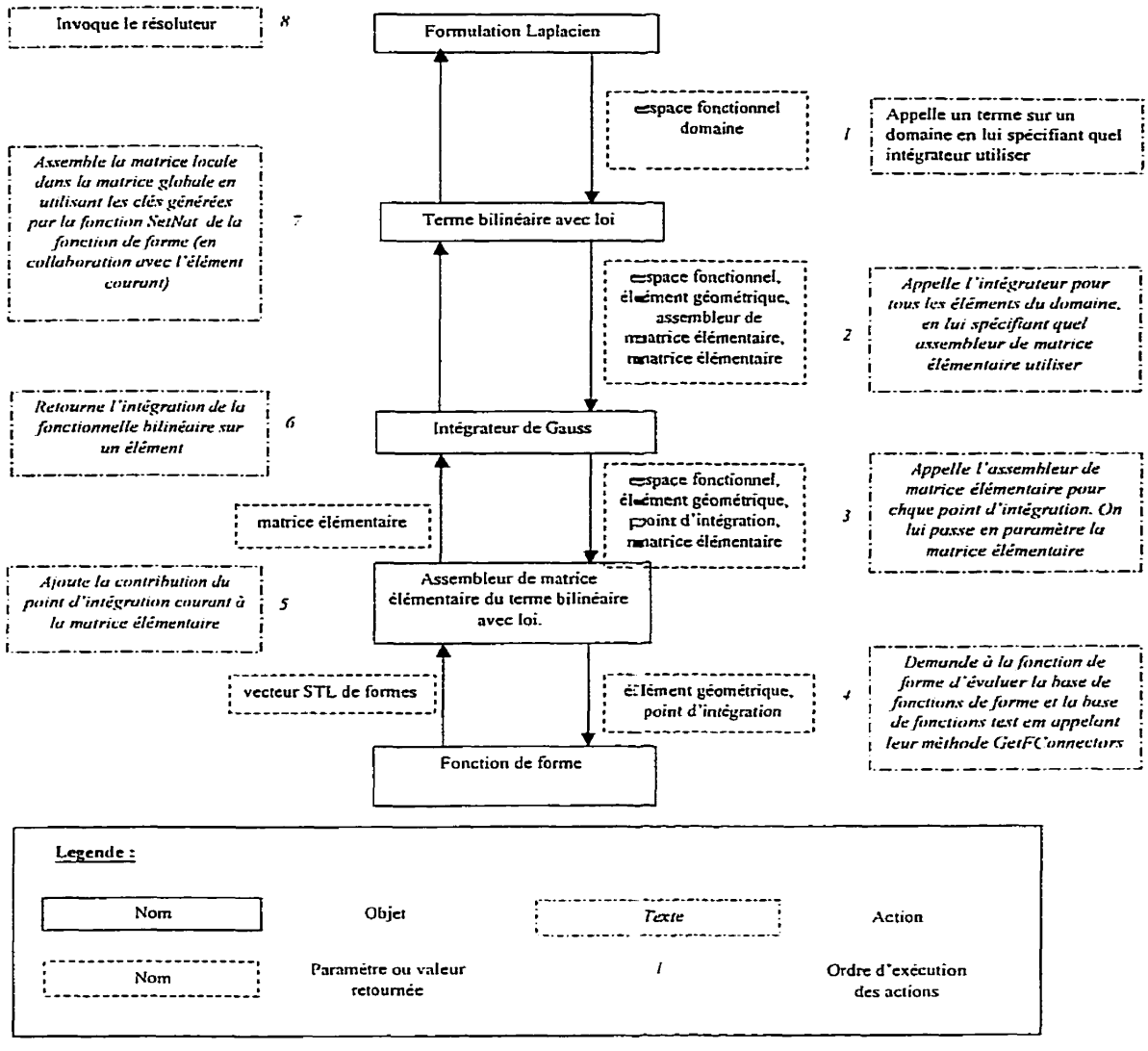


Figure 9.3 - Déroulement de l'assemblage et de la résolution du Laplacien

Le fragment de code suivant est un extrait de la fonction principale de cette formulation, `TreatmentOfFormulation`. Dans cet extrait :

- `meshCurrent` est une référence sur le maillage;

- `bcnrCurrent` est une référence sur le registre de conditions aux limites;

- `veclppqCurrent` est un vecteur de la STL contenant des pointeurs sur les quantités physiques manipulées (une seule quantité physique ici :*f*).

- `veclpcharacCurrent` est un vecteur STL contenant des pointeurs sur les charactéristiques (générant les lois physiques) à utiliser.

On effectue les actions suivantes :

1) `TreatmentOfGroups(meshCurrent,bcnrCurrent,veclppqCurrent[0],((DofManager_c*&)*(veclppqCurrent[0])),0);`

   Imposition des conditions aux limites de Dirichlet. On passe pour cela le maillage courant (`meshCurrent`) , le registre de conditions aux limites (`bcnrCurrent`), la quantité physique pour laquelle on veut imposer les conditions aux limites (`veclppqCurrent[0]`), le gestionnaire de degrés de liberté où imposer les conditions `((DofManager_c*&)*(veclppqCurrent[0]))` et le type de condition (0) à la fonction `TreatmentOfGroups`. On remarque le transtypage (conversion de type) d'une quantité physique en gestionnaire de liberté. La surcharge de cette opération retourne le gestionnaire de degrés de liberté associé à la quantité physique.

2) `ActionOnElements(DEFDOF_ACTION,meshCurrent,veclppqCurrent[0],meshCurrent.GetAllElements(),(DofManager_c*&)*(veclppqCurrent[0]),matregCurrent,0);`

   Création des degrés de liberté. En fait, seuls les degrés de liberté n'ayant pas été

créés lors de l'imposition des conditions de Dirichlet sont créés. Le gestionnaire de degrés de liberté gère automatiquement cet aspect.

3) `((DofManager_c*&)*(veclppqCurrent[0]))->AllocMatrix();`

Allocation de la matrice globale.

4) `TreatmentOfGroups(meshCurrent,bcnrCurrent,veclppqCurrent[0],((DofManager_c*&)*(veclppqCurrent[0])),1);`

Imposition des conditions aux limites de Robin. Comme cela requiert l'assemblage de termes dans la matrice globale, il est nécessaire d'avoir préalablement créé les degrés de liberté. La seule différence avec l'imposition des conditions de Dirichlet est dans la valeur du dernier paramètre (1), spécifiant le type de condition à imposer.

5) `static GaussIntegrator integGauss;`

Instanciation d'un intégrateur de Gauss.

6) `static BilinearTermWithLaw A(*(veclpfsdCurrent[0]),integGauss,veclppqCurrent[0],veclppqCurrent[0]);`

Instanciation du terme bilinéaire avec loi. On lui passe en paramètre l'espace fonctionnel courant (`*veclpfsdCurrent[0]`), l'intégrateur à utiliser (`integGauss`), la quantité physique interpolée par les fonctions de forme et test (`veclppqCurrent[0]`).

7) `A(((DofManager_c*&)*(veclppqCurrent[0])),veclpcharacCurrent[0],*lpdomSupport);`

Assemblage du terme par invocation de son opérateur de fonction. On indique dans quel gestionnaire de degrés de liberté assembler les valeurs (`((DofManager_c*&)*(veclppqCurrent[0]))`), quelle caractéristique physique doit être utilisée pour générer la loi en chaque élément (`veclpcharacCurrent[0]`) et sur quel domaine le terme doit être assemblé (`*lpdomSupport`).

8) `ResolveFormulation((DofManager_c*&)*(veclppqCurrent[0]));`

Enfin, on appelle le résoluteur.

Toutes les étapes de cette formulation ont été décrites dans la section « Exemples simples ». On peut apprécier la compacité et la simplicité de l'écriture nécessaire à la résolution d'un Laplacien. Mais le bénéfice n'est pas seulement là : du fait de l'utilisation du mécanisme de double dispatch, on effectue la liaison entre l'élément géométrique et la fonction de forme lors de l'exécution et non pas lors de l'écriture du programme. Cela signifie qu'on peut utiliser n'importe quel type d'élément 2D, mais aussi 3D. **Ainsi, la formulation écrite ci-dessus fonctionnera non seulement pour la résolution d'un laplacien sur un maillage composé de triangles mais également pour n'importe quel maillage (au sens des éléments finis bien sûr) composé de n'importe quels types d'éléments de n'importe quelle dimension.**

9.3 Convection naturelle dans une cavité

9.3.1 Introduction

On traite dans cette section de la simulation de la convection naturelle dans une cavité et de la démarche suivie pour développer une formulation capable de résoudre ce problème. Après avoir présenté le développement mathématique et les approximations effectuées pour mettre le problème en équations, on présente le processus en cinq étapes pour développer une formulation résolvant numériquement ce problème :

1) écriture d'une formulation résolvant l'équation de Stokes;

2) modification de cette formulation pour résoudre l'équation de Navier-Sokes (intégration d'un schéma non-linéaire);

3) implémentation de la formulation thermique;

4) ajout du terme de convection à la formulation Navier-Stokes

5) couplage informatique des deux formulations.

9.3.2 Mise en équations

9.3.2.1 Équation de Navier-Stokes en régime permanent

On considère dans ce problème qu'aucun des termes de l'équation de Navier-Stokes n'est négligeable. On conserve donc le terme d'inertie, le terme visqueux et le terme de pression. Afin de tenir compte des effets de la convection il est nécessaire de considérer la gravité. On a par conséquent l'équation de conservation de la quantité de mouvement suivante:

$$\rho U \nabla U = -\nabla P + F_G + \mu \nabla^2 U$$

(inertie=pression+gravité+viscosité)

Équation 2 – Conservation de la quantité de mouvement

et l'équation de conservation de la masse suivante :

$$\nabla(\rho U) = 0$$

Équation 3 – Conservation de la masse

On ne peut à priori pas se placer, comme on le fait très souvent, dans le cas d'un fluide incompressible : s'il n'y a pas variation de la densité, les forces massiques seront constantes et rien ne bougera. On utilisera ici l'approximation de Boussinesq, présentée dans la sous-section du même nom, et comment elle permet d'obtenir des équations similaires à celle du cas incompressible.

9.3.2.2 Équation de l'énergie en régime permanent

Le calcul de la température se fait par la résolution de l'équation de l'énergie qui est dans le cas de la convection naturelle en régime permanent :

$$U \nabla T = \alpha \nabla^2 T$$

Équation 4 – Équation de l'énergie dans le cas de la convection naturelle en régime permanent

Où α est le coefficient de diffusivité thermique, égal au rapport de la conductivité thermique sur la capacité calorifique volumétrique.

$$\alpha = \frac{k}{\rho c_p}$$

9.3.2.3 Approximation de Boussinesq

L'approximation de Boussinesq consiste à considérer la variation de densité du fluide comme trop petite pour créer une variation de volume et le fluide comme incompressible la plupart du temps. Il est cependant nécessaire que l'incompressibilité du fluide ne soit pas totalement ignorée (sinon, on supprime tout effet de convection), mais que cette variation de compressibilité intervienne uniquement dans les forces massiques.

9.3.2.4 Loi constitutive

On considère ici une loi constitutive linéaire reliant la variation de densité à la variation de température :

$$\frac{\rho - \rho_0}{\rho_0} = -\beta (T - T_0)$$

Équation 5 – Loi constitutive

9.3.2.5 Équations résultantes et nombres adimensionnels

Considérer ρ constant partout sauf dans le terme de gravité nous permet de simplifier l'équation de conservation de la masse en :

$$\mathrm{div}(U) = 0$$

Il sera cependant important de se rappeler que l'équation « réelle » est celle faisant intervenir ρ lorsque nous écrirons notre formulation éléments finis.

L'introduction de l'approximation de Boussinesq donne l'équation de quantité de mouvement suivante :

$$\rho_0 U \nabla U = -\nabla P + \rho_0 (1 - \beta (T - T_0)) \nabla z + \mu \nabla^2 U$$

L'équation de l'énergie reste inchangée.

9.3.3 Formulation faible

On utilisera ici des éléments finis de Galerkin, c'est à dire que l'on prendra la même base de fonctions d'interpolation pour les fonctions de forme et les fonctions test d'une même grandeur.

Afin de garantir la cohérence de notre problème et la convergence de notre calcul (condition LBB), il est nécessaire de prendre des fonctions d'interpolation (fonctions de forme) quadratiques pour la vitesse, quadratiques pour la température et linéaires pour la pression.

On se place dans H_0^1 (espace des fonctions de forme dont la dérivée au sens des distributions existe) et on multiplie l'équation de conservation de la masse par les fonctions de forme de pression (P'), l'équation de CQM par les fonctions de forme de vitesse(U') et l'équation de l'énergie par les fonctions de forme de température (T'). On intègre chaque équation sur tout le domaine (ici une surface).

$$\iint \rho_0 U \nabla U \, U' ds = -\iint \nabla P \, U' ds + \iint \rho_0 (1 - \beta (T - T)) \nabla z \, U' ds + \iint \mu \nabla^2 U \, U' ds$$

$$\iint \mathrm{div}(U) \, P' ds = 0$$

$$\iint U\,\nabla T\,T'ds = \alpha \iint \nabla^2 T\,T'ds$$

L'espace que l'on a choisi nous force à abaisser l'ordre des Laplaciens de la vitesse et de la température par une intégration par partie :

$$\iint \nabla^2 U\,U'dS = \int \frac{\partial U}{\partial n} dl - \iint \nabla U\,\nabla U'dS$$

où $\vec{n}$ est le vecteur normal au bord de notre domaine.

Du fait de l'ellipticité de notre problème le terme de bord disparaît, soit par imposition directe de U, soit par imposition de sa dérivée.

Un autre traitement à effectuer consiste à remplacer la dérivée sur la pression par une dérivée sur la fonction test, également au moyen d'une intégration par partie.

On obtient alors :

$$\iint \rho_0\,U\nabla U\,U' - \iint P\,\nabla U' + \mu \iint \nabla U\,\nabla U' = \rho_0 g \iint (1-\beta(T-T_0))\nabla z\,U'$$

$$1+2+3=4$$

$$\iint \mathrm{div}(U)\,P' = 0$$

$$5=0$$

$$\iint U\,\nabla T\,T' - \alpha \iint \nabla T\,\nabla T' = 0$$

$$6+7=0$$

On distingue alors sept termes, que nous séparons en deux formulations : une pour les équations de Navier-Stokes (formulation « Navier-Stokes »), une pour l'équation d'énergie (formulation « Thermique »). Chacune de ces formulations comporte au moins un terme trilinéaire. Il s'agit du terme 1 pour la formulation Navier-Stokes et du terme 6 pour la formulation thermique.

9.3.4 Stokes

La première étape de la résolution du problème de la convection naturelle est l'écriture d'une formulation résolvant les équations de Stokes pour un fluide incompressible, sans tenir compte du terme de convection ni, bien évidemment, du terme d'inertie.

$$\begin{cases} -\iint P\,\mathrm{div}(U')\,d\Omega + \mu \iint \nabla U\,\nabla U'\,d\Omega = 0 \\ \iint \mathrm{div}(U)P'\,d\Omega = 0 \end{cases}$$

Équation 6 – Forme faible des equations de Stokes pour un fluide incompressible

Traduisons ces termes en des entités de FEMView. Il y a deux termes bilinéaires sans loi physique et un terme en comportant une précédemment simplifiée, tout comme dans l'exemple « Laplacien » : le terme de viscosité $\mu \iint \nabla U\,\nabla U'\,d\Omega = 0$. La formulation serait tout à fait similaire à celle du Laplacien si un détail important ne demandait un traitement particulier. On évalue en effet la *divergence* de la vitesse. Or, la vitesse est une 3*0-forme et, comme expliqué dans la partie présentant la structure mathématique de la méthode des éléments finis, l'opérateur de dérivation extérieure d'une 0-forme est le gradient. Cette situation est due, tout comme pour le terme de viscosité, à la

simplification d'une loi de comportement. C'est en fait $\mathrm{div}(\rho U')$ que l'on évalue, le ρ ayant été simplifié lors de la prise en compte de l'incompressibilité du fluide dans le développement des équations de Stokes. on rappelle que le produit d'une p-forme par une loi physique est une (N-p)-forme, où N est la dimension de l'espace. Ainsi, ρU est une (3-1)-forme et l'opérateur de dérivation extérieure d'une 2-forme est bien la divergence. On a vu lors de l'analyse points communs/variations des fonctions de forme que cette situation est traitée par l'utilisation d'un adapteur. On évite ainsi l'utilisation d'une pseudo loi de comportement de valeur unitaire. On évaluera ainsi grad(U') :

$$\mathrm{grad}(U') = \begin{pmatrix} \frac{\partial U_X}{\partial X} & \frac{\partial U_X}{\partial Y} & \frac{\partial U_X}{\partial Z} \\ \frac{\partial U_Y}{\partial X} & \frac{\partial U_Y}{\partial Y} & \frac{\partial U_Y}{\partial Z} \\ \frac{\partial U_Z}{\partial X} & \frac{\partial U_Z}{\partial Y} & \frac{\partial UZ}{\partial Z} \end{pmatrix}$$

puis on lui appliquera l'adapteur de trace. On obtient ainsi :

$$trace(grad(U')) = \frac{\partial U_X}{\partial X} + \frac{\partial U_Y}{\partial Y} + \frac{\partial U_Z}{\partial Z} = div(U')$$

Comme expliqué lors de la présentation des adapteurs, ces derniers sont spécifiés à une fonction de forme lors de son instanciation. Ainsi, on ne voit pas directement apparaître l'adapteur de trace dans le code de la formulation permettant de résoudre les équations de Stokes. Il est cependant bien présent et utilisé par l'espace fonctionnel défini pour assembler le terme $-\iint P\,\mathrm{div}(U')d\Omega$.

Le code de la formulation est décomposé en 8 étapes. On passe comme toujours en paramètres à la formulation les espaces fonctionnels, quantités physiques et autres registres dont elle doit se servir pour accomplir sa tâche. Ainsi :

- `veclpfsdCurrent[0]` désigne l'espace fonctionnel déclaré pour assembler le terme de pression

- `veclpfsdCurrent[1]` désigne l'espace fonctionnel déclaré pour assembler le terme de viscosité

- `veclpfsdCurrent[2]` désigne l'espace fonctionnel déclaré pour assembler le terme de continuité

- `veclppqCurrent[0]` désigne la quantité physique pression

- `veclppqCurrent[1]` désigne la quantité physique vitesse

- `((DofManager_c*&)*(veclppqCurrent[1]))` est l'utilisation de la conversion de type d'une quantité physique en un gestionnaire de degrés de liberté. Cette conversion retourne le gestionnaire associé à cette quantité physique.

Il faut remarquer que l'on peut, lors de la définition des quantités physiques, déclarer la pression et la vitesse comme utilisant le même gestionnaire de degrés de liberté. C'est uniquement pour souligner que l'on utilise un unique système global que l'on assemble tous les termes dans le gestionnaire de degrés de liberté défini pour la quantité vitesse.

1) `static GaussIntegrator integGauss;`

   Déclaration d'un intégrateur de Gauss.

2)
```
static BilinearTermWithoutLaw
     P_divU(*(veclpfsdCurrent[0]), integGauss, veclppqCurrent[0],
          veclppqCurrent[1]);

static BilinearTermWithLaw
     viscosite(*(veclpfsdCurrent[1]), integGauss, veclppqCurrent[1],
          veclppqCurrent[1]);

static BilinearTermWithoutLaw
     continuite(*(veclpfsdCurrent[2]), integGauss,
```

```
veclppqCurrent[1],
          veclppqCurrent[0]);
```

Déclaration des termes.

```
3) for(iCompteur = 0;iCompteur < NB_PHYS_STOKES;iCompteur++)
     TreatmentOfGroups(meshCurrent, bcnrCurrent,
          veclppqCurrent[iCompteur],
          ((DofManager_c*&)*(veclppqCurrent[1])),0);
```

Imposition des conditions aux limites de Dirichlet pour la vitesse et la pression. On remarque que ces conditions aux limites sont assemblées dans le même gestionnaire de degrés de liberté. On doit en effet tout assembler dans la même matrice globale pour résoudre le problème de Stokes.

```
4) ActionOnElements(DEFDOF_ACTION, meshCurrent,
          veclppqCurrent[iCompteur], meshCurrent.GetAllElements(),
          (DofManager_c*&)(*veclppqCurrent[1]), matregCurrent,0);
```

Création des degrés de liberté pour les deux quantités

```
5) ((DofManager_c*&)(*veclppqCurrent[1]))->AllocMatrix();
```

Allocation de la matrice globale

```
6) for(iCompteur = 0;iCompteur < NB_PHYS_STOKES;iCompteur++)
     TreatmentOfGroups(meshCurrent, bcnrCurrent,
          veclppqCurrent[iCompteur],
          ((DofManager_c*&)*(veclppqCurrent[1])),1);
```

Imposition des conditions aux limites de robin

```
7) P_divU(((DofManager_c*&)(*veclppqCurrent[1])),*lpdomSupport);

   viscosite(((DofManager_c*&)(*veclppqCurrent[1])),lpcharacMu,*lpdomSu
   pport);

   continuite(((DofManager_c*&)(*veclppqCurrent[1])),*lpdomSupport);
```

Assemblage des termes

```
8) ResolveFormulation(((DofManager_c*&)(*veclppqCurrent[1])));
```

Appel du résoluteur

9.3.5 Navier-Stokes

Ajoutons maintenant le terme d'inertie $\iint \rho_0 \, U \nabla U \, U' \, d\Omega$ pour obtenir les équations de Navier-Stokes pour un fluide incompressible :

$$\begin{cases} \iint \rho_0 \, U \nabla U \, U' \, d\Omega - \iint P \, \mathrm{div} U' \, d\Omega + \mu \iint \nabla U \, \nabla U' \, d\Omega = 0 \\ \iint \mathrm{div}(U) P' \, d\Omega = 0 \end{cases}$$

Équation 7 – Forme faible des équations de Navier-Stokes pour un fluide incompressible

Le terme d'inertie a une particularité : il est trilinéaire et exige donc un traitement spécial. Comme toutes les composantes de ce terme trilinéaire constituent l'une des quantités pour lesquelles on cherche à obtenir une solution en résolvant les équations de Navier-Stokes, en l'occurrence la vitesse *U*, il est nécessaire d'utiliser un schéma non-linéaire. On choisit ici d'utiliser un simple point fixe car il est démontré que, utilisé pour résoudre l'équation de Navier-Stokes, ce mécanisme converge vers la solution. On considérera à chaque pas de temps non-linéaire la vitesse *U* comme étant fixe et égale à la valeur calculée au pas non-linéaire précédent (U_{p-1}). On assemblera donc le terme suivant :

$$\iint \rho_0 \, U_{p-1} \nabla U \, U' \, d\Omega$$

Le besoin des méthodes non linéaires de récupérer des valeurs à l'itration précédente, et plus généralement, la nécessité d'être capable de récupérer la valeur d'une quantité calculée par une formulation lors de l'assemblage d'un terme induit l'existence des

contracteurs. On rappelle (voir la section traitant de l'analyse points communs/variations des termes) que les contracteurs sont des objets capables de récupérer une valeur dans une source de données. On se sert ici du contracteur `QuantityLastStep` pour extraire la vitesse au barycentre de l'élément géométrique courant au pas de temps précédent. Ainsi lors du calcul en chaque point d'intégration, les fonctions de forme et test seront évaluées et multipliées comme dans le cas d'un terme bilinéaire sauf que l'on effectuera ici en plus une multiplication par la valeur extraite par ce contracteur.

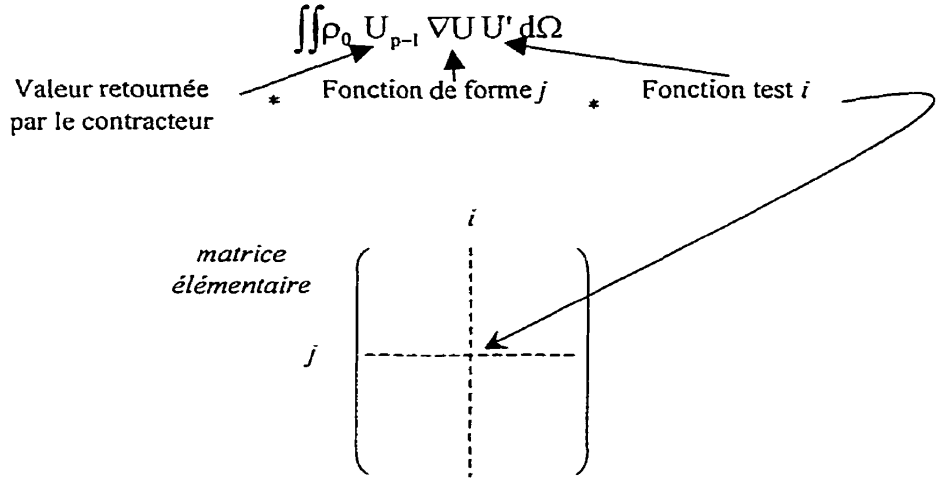


Figure 9.4 - Calcul d'un terme trilinéaire en un point

L'utilisation d'un algorithme de résolution non-linéaire implique également le stockage puis l'effacement d'itérations non linéaires. Il est donc nécessaire d'avertir le gestionnaire de degrés de liberté du début et de la fin d'une résolution non linéaire afin qu'il puisse mémoriser quelles sont les itérations non linéaires pour les effacer toutes ensuite, hormis la dernière itération, lorsque l'algorithme aura convergé.

Le code de la formulation Navier-Stokes est similaire à celui de la formulation Stokes à l'ajout du terme trilinéaire et du code mettant en œuvre l'algorithme de point fixe près.

```
1) static GaussIntegrator integGauss;
```

Instanciation d'un intégrateur de Gauss.

```
2) for(iCompteur = 0;iCompteur < NB_PHYS_STOKES;iCompteur++)
        TreatmentOfGroups(meshCurrent, bcnrCurrent,
             veclppqCurrent[iCompteur],
             ((DofManager_c*&)*(veclppqCurrent[1])), 0);
```

Imposition des conditions de Dirichlet pour la vitesse et la pression.

```
3) for(iCompteur = 0;iCompteur < NB_PHYS_NAVIER-STOKES;iCompteur++)
        ActionOnElements(DEFDOF_ACTION, meshCurrent,
             veclppqCurrent[iCompteur], meshCurrent.GetAllElements(),
             (DofManager_c*&)*(veclppqCurrent[1]), matregCurrent,0);
```

Création des degrés de liberté pour la vitesse et la pression sur tout le maillage.

```
4) ((DofManager_c*&)*veclppqCurrent[1])->AllocMatrix();
```

Allocation de la matrice globale.

```
5) static BilinearTermWithoutLaw
   P_divU(*(veclpfsdCurrent[0]),integGauss,veclppqCurrent[0],veclppqCur
   rent[1]);

   static BilinearTermWithLaw
   viscosite(*(veclpfsdCurrent[1]),integGauss,veclppqCurrent[1],veclppq
   Current[1]);

   static BilinearTermWithoutLaw
   continuite(*(veclpfsdCurrent[2]),integGauss,veclppqCurrent[1],veclpp
   qCurrent[0]);
```

Instanciation des termes bilinéaires.

```
6) static QuantityLastStep
   qlsSpeed((DofManager_c*&)*veclppqCurrent[1],veclppqCurrent[1]);
```

Instanciation du contracteur. On lui précise le gestionnaire de degré à utiliser pour récupérer la valeur de la quantité passée en second paramètre.

```
7) static TrilinearHalfContractedTermWithoutLawCbC
   UlsdUUp(*(veclpfsdCurrent[3]), qlsSpeed, integGauss,
   veclppqCurrent[1],
        veclppqCurrent[1]);
```

Instanciation du terme trilinéaire. Tout à fait similaire à celle d'un terme bilinéaire si ce n'est le passage du contracteur en paramètre.

```
8) do{
        resid = 1.0;
        it = 0;
        if(it2++ == 0)Convergence_Factor = 1.0;
        do{
```

Début de la boucle de résolution non linéaire.

```
9)          ((DofManager_c*&)*veclppqCurrent[1])->NonLinearStep();
```

Faire du pas de temps courant le pas p-1. La matrice globale est remise à 0.

```
10)          for(iCompteur = 0;iCompteur < NB_PHYS_NAVIER_STOKES;
                 iCompteur++)
                   TreatmentOfGroups(meshCurrent, bcnrCurrent,
                        veclppqCurrent[0],
                        ((DofManager_c*&)*(veclppqCurrent[1])), 1);
```

Imposition des conditions de Robin pour la vitesse et la pression

```
11)            UlsdUUp(((DofManager_c*&)*veclppqCurrent[1]),
                     *lpdomSupport);
               P_divU((DofManager_c*&)*veclppqCurrent[1],
   *lpdomSupport);
               viscosite((DofManager_c*&)*veclppqCurrent[1],
                     veclpcharacCurrent[0], *lpdomSupport);
               continuite((DofManager_c*&)*veclppqCurrent[1],
                     *lpdomSupport);
```

Assemblage des termes.

```
12)            ((DofManager_c*&)*veclppqCurrent[1]) \
                     ->NonLinearSolveMatrix();
```

Résolution de la matrice globale.

```
13)            residex = resid;
               resid = ((DofManager_c*&)*veclppqCurrent[1]) \
                           ->NonLinearResidual();
               it ++;
        }while( (resid > 5.e-4) && (it < 50));
        Convergence_Factor = sqrt(Convergence_Factor);
        printf("cf = %12.5E\n",Convergence_Factor);
   }while(Convergence_Factor > 1.001);
```

Fin de la boucle non-linéaire. Calcul du taux de convergence et répétition du processus tant qu'il n'est pas satisfaisant.

```
14)    ((DofManager_c*&)*veclppqCurrent[1])->NonLinearEndResolution();
```

Signale au gestionnaire de degré de liberté que la résolution non-linéaire est terminée. Toutes les itérations non-linéaires, à l'exception de la dernière bien sûr, sont effacées.

9.3.6 Thermique

On écrit dans cette partie une formulation permettant de résoudre numériquement par éléments finis la formulation faible de l'équation de l'énergie :

$$\iint U \nabla T \, T' - \alpha \iint \nabla T \nabla T' = 0$$

Le premier terme ($\iint U \nabla T \, T'$) de cette équation est trilinéaire. Cependant, contrairement au terme trilinéaire de l'équation de Navier-Stokes, il ne nécessite pas de traitement particulier. La vitesse U n'est en effet pas la variable pour laquelle on recherche une solution en résolvant cette équation. Il suffit de se servir d'un contracteur pour aller récupérer la valeur de la vitesse en un élément au pas de temps précédent dans le gestionnaire de degrés de liberté de la vitesse et de procéder comme pour l'assemblage d'un simple terme bilinéaire, sans devoir utiliser de schéma de résolution d'équation non linéaire. Le code de la formulation ne devrait plus poser le moindre problème au lecteur à cette étape de ce mémoire, tant il est similaire aux codes précédents.

```
1) static GaussIntegrator integGauss;
```

Instanciation d'un intégrateur de Gauss.

```
2) TreatmentOfGroups(meshCurrent,bcnrCurrent, veclppqCurrent[0],
       (DofManager_c*&)(*veclppqCurrent[0]),0);
```

Imposition des conditions aux limites de Dirichlet.

```
3) ActionOnElements(DEFDOF_ACTION,meshCurrent,veclppqCurrent[0],
   meshCurrent.GetAllElements(),(DofManager_c*&)*veclppqCurrent[0],
```

```
matregCurrent,0);
```

Création des degrés de liberté sur tous les éléments.

```
4) ((DofManager_c*&)*veclppqCurrent[0])->AllocMatrix();
```

Allocation de la matrice globale.

```
5) TreatmentOfGroups(meshCurrent,bcnrCurrent,
   veclppqCurrent[0],(DofManager_c*&)(*veclppqCurrent[0]),1);
```

Imposition des conditions de Robin.

```
6) static BilinearTermWithLaw
   TTp(*(veclpfsdCurrent[1]),integGauss,veclppqCurrent[0],veclppqCurren
   t[0]);
```

Instanciation du terme bilinéaire.

```
7) static QuantityLastStep
   qlsSpeed(((DofManager_c*&)(*veclppqCurrent[1])),veclppqCurrent[1]);
```

Instanciation du contracteur chargé d'aller récupérer la vitesse dans le gestionnaire de degrés de liberté.

```
8) static TrilinearHalfContractedTermWithoutLaw
   speed(*(veclpfsdCurrent[0]),
         qlsSpeed,
         integGauss,
         veclppqCurrent[0],
         veclppqCurrent[0]);0
```

Instanciation du terme trilinéaire semi-contracté avec loi.

```
9) TTp(((DofManager_c*&)*veclppqCurrent[0]),&lpcharacCurrent[0],*lpdomS
   upport);
```

Assemblage du terme trilinéaire.

```
10)    speed((DofManager_c*&)(*veclppqCurrent[0]),*lpdomSupport);
```

Assemblage du terme bilinéaire.

```
11)    ResolveFormulation((DofManager_c*&)(*veclppqCurrent[0]));
```

Résolution de la matrice globale.

9.3.7 Prise en compte du phénomène de convection par la formulation Navier-Stokes

La prise en compte de la convection ne requiert, lorsque l'on fait l'approximation de Boussinesq, que l'ajout d'un terme source à l'équation de conservation de la quantité de mouvement.

$$\iint \rho_0 \, U \nabla U \, U' dS - \iint P \nabla U' dS + \mu \iint \nabla U \nabla U' dS = \rho_0 g \iint (1 - \beta \, (T - T_0)) \nabla z \, U' dS$$

On utilise ici un terme source multipliant le résultat de l'évaluation de la fonction $(1-\beta\,(T-T_0))\nabla z$ par la base de fonctions test. On peut donc utiliser le terme source générique `FunctionSourceTerm` et n'écrire qu'une fonction récupérant la valeur de la température au point courant de l'espace de référence de l'élément courant au pas de temps courant et calculant la formule.

9.3.8 Coupler les formulations Navier-Stokes et Thermique

Le travail de couplage des formulations Thermique et Navier-Stokes ne requiert aucune modification des formulations déjà écrites. En effet, celles-ci échangent déjà des données (valeurs de *T* et de *U*) par l'utilisation de contracteurs lors de la récupération de résultats de calcul de l'autre formulation. Il suffit donc maintenant d'écrire une routine principale appelant tour à tour la formulation Thermique et la formulation Navier-Stokes. Comme la convection naturelle n'est qu'un exemple d'utilisation du code et pas un sujet de recherche à part entière, on ne s'est pas donné la peine d'imaginer des mécanismes complexes : on exécute simplement les deux formulations jusqu'à ce que le résultat produit soit jugé stabilisé (par examen des itérations successives) par un simple examen visuel.

```
1) for (iIteration = 0;iIteration < NB_ITERATIONS;iIteration++){
```

```
thermique.TreatmentOfFormulation(maillage,GlobalBCNamesRegistry
            ,mrMateriaux,genparamsCurrent);

      NavierStokes.TreatmentOfFormulation(maillage,
            GlobalBCNamesRegistry,mrMateriaux,genparamsCurrent);
```

Résolution itérative des formulations thermique et Navier-Stokes.

2)
```
static char lpacNomFichier[1024];
static char lpacNomCommentaire[1024];

cout << "Exporting temperature in GMSH file format..." \
     << endl;
sprintf(lpacNomFichier,"Temperature%d.pos",iIteration);
sprintf(lpacNomCommentaire,"Temperature%d",iIteration);
GlobalPhysicalQuantitiesNamesRegistry["temperature"].\\
            ExportGMSH(maillage,lpacNomFichier
                  ,lpacNomCommentaire);

cout << "Exporting speed in GMSH file format..." << endl;
sprintf(lpacNomFichier,"Vitesse%d.pos",iIteration);
sprintf(lpacNomCommentaire,"Vitesse%d",iIteration);
GlobalPhysicalQuantitiesNamesRegistry["speed"].\\
ExportGMSH(maillage,lpacNomFichier,lpacNomCommentaire);
```

Sauvegarde des résultats

3)
```
((DofManager_c*&)GlobalPhysicalQuantitiesNamesRegistry["temperatu
re"])->SaveOnFile("Convection.temperature.dof");

      ((DofManager_c*&)GlobalPhysicalQuantitiesNamesRegistry["spe
ed"])->SaveOnFile("Convection.navier-stokes.dof");
```

Sauvegarde des degrés de liberté afin de pouvoir reprendre la simulation.

4)
```
((DofManager_c*&)GlobalPhysicalQuantitiesNamesRegistry["temperatu
re"])->TimeStep(0.1);
((DofManager_c*&)GlobalPhysicalQuantitiesNamesRegistry["speed"])-
>TimeStep(0.1);
```

Passage au pas de temps suivant des deux gestionnaires de degrés de liberté

}

9.4 Lesaint-Raviart

9.4.1 Introduction

On présente dans cette partie une formulation dite de Lesaint-Raviart. Elle permet de calculer le transport d'une quantité f par un champ de vitesse $\vec{u}$. L'avantage de cette formulation est qu'elle permet de calculer un transport exact et peut être calculée élément par élément. On obtient ainsi une vitesse de résolution plus importante qu'avec un système global.

Après avoir présenté le développement mathématique de la formulation Lesaint-Raviart (on tâche d'être le plus rigoureux possible, mais comme cette formulation n'est pas le thème central de cette thèse, on omet volontairement certains détails afin de fournir au lecteur une démonstration plus heuristique, mais plus digeste). On étudiera les solutions apportées aux problèmes posés par cette formulation très particulière.

9.4.2 Développement mathématique de la formulation Lesaint-Raviart

Notation :

Ω Le domaine sur lequel on résoud la formulation

Ω_i Un des éléments faisant partie du maillage couvrant le domaine

$\partial\Omega$ Le bord du domaine Ω

$\partial\Omega_i$ Le bord de l'élément Ωi

On transporte sur le domaine Ω une quantité f connue sur une partie de la frontière $\partial\Omega$ de Ω par le biais d'un champ de vitesse $\vec{u}$ préalablement calculé (par les formulations Stokes ou Navier-Stokes par exemple ou toute autre solution laminaire ou sans zones de recirculation). Pour cela ,on veut résoudre le problème suivant :

$$P : \begin{cases} \vec{u}.\nabla f = 0 & dans\ \Omega \\ f = f^{-} & sur\ \partial\Omega^{-} \end{cases}$$

où :

$$\partial\Omega^{-} \equiv \{x \in \partial\Omega \mid \vec{u} \cdot \vec{n}(x) < 0\}$$

$$\partial\Omega^{0} \equiv \{x \in \partial\Omega \mid \vec{u} \cdot \vec{n}(x) = 0\}$$

$$\partial\Omega^{+} \equiv \{x \in \partial\Omega \mid \vec{u} \cdot \vec{n}(x) > 0\}.$$

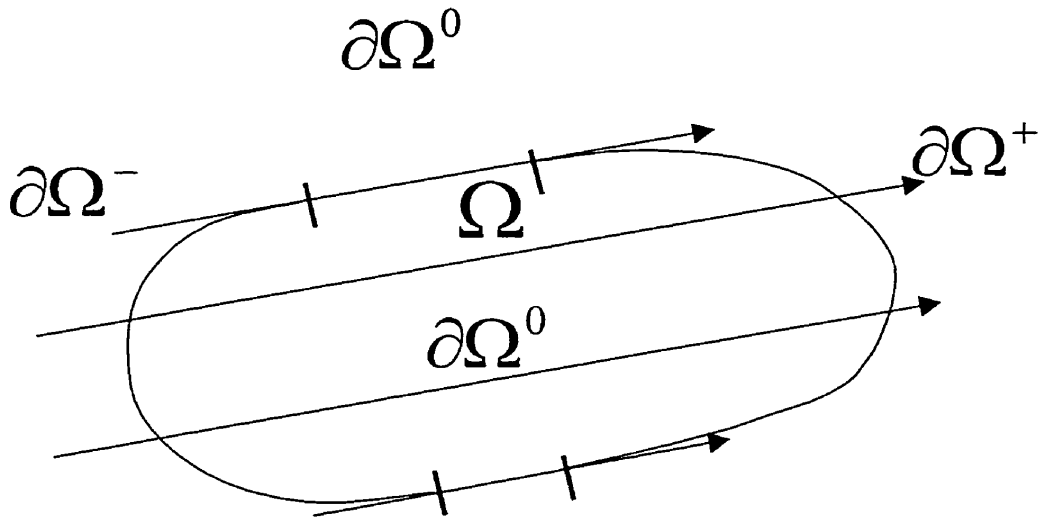


Figure 9.5 – Géométrie considérée

On choisit de le résoudre par la méthode des éléments finis. On définit pour cela un maillage du domaine Ω par:

$$T(\Omega) \equiv \left\{ \Omega_t \mid \Omega = \bigcup_{i=1}^{n} \Omega_i \ et\ \Omega_i \cap \Omega_j = \text{simplexe de dimension} \leq n-1 \text{ ou } \phi, \forall i \neq j \right\}$$

On veut déterminer une forme faible de ce problème. On se place donc dans l'espace

$$V = \left\{ h \in L^2(\Omega) / h|_{\Omega_i} \in H_1(\Omega_i) \right\}$$

et on multiplie P (l'équation du problème) par $g \in V$. On obtient alors:

$$\text{P':} \begin{cases} \int_{\Omega} (\vec{u}.\nabla f)\, g = 0 \quad dans\ \Omega \\ f = f^- \quad sur\ \partial\Omega^- \end{cases} \qquad \forall g \in V$$

On désire maintenant justifier par une série de manipulations une version de P' permettant d'imposer faiblement f- sur $\partial\Omega^-$.

On peut montrer ([6]) par la méthode de Lions-Magenes ([7]) que les vecteurs de

$$H(div;\Omega) \equiv \left\{ \vec{q} \mid \vec{q} \in (L^2(\Omega))^2; \nabla \cdot \vec{q} \in L^2(\Omega) \right\}$$

admettent une trace normale sur $\partial\Omega$ bien définie $\vec{q}.\vec{n} \in H^{1/2}(\partial\Omega)$ et que l'on a la formule d'intégration par parties suivante:

$$\int_{\Omega} \vec{q}.\nabla v\, dX + \int_{\Omega} div(\vec{q})\, v\, dX = <\vec{q}.\vec{n}, v>_{H^{-1/2}(\partial\Omega)\times H^{1/2}(\partial\Omega)}, \ \forall \vec{q} \in H(div,\Omega) \text{ et } v \in H^1(\Omega) \qquad (1)$$

où $H^1(\Omega) \equiv \left\{ v \mid v \in L^2(\Omega); \partial v / \partial x_i \in L^2(\Omega) \right\}$.

Si l'on choisit $\vec{q} = \vec{u}\, g$ et $v = f$ avec $\vec{u} \in H(div;\Omega); f, g \in H^1(\Omega)$ alors $\nabla \cdot (\vec{u}g) \in L^2$ existe au sens des distributions et est contenu dans $L^2(\Omega)$. En effet,

$$< \nabla.(\vec{u}\, g), \varphi >_{D'\times D} = < [(\nabla.\vec{u})\, g + (\vec{u}.\nabla g)], \varphi >_{D'\times D}$$

au sens des distributions, mais comme $\vec{u} \in H(div;\Omega) \Rightarrow \nabla.\vec{u} \in L^2(\Omega) \Rightarrow \nabla.\vec{u}\, g \in L^2(\Omega)$ et $g \in H^1(\Omega) \Rightarrow \vec{u}.\nabla g \in L^2(\Omega)$. Donc $\vec{u}\, g \in H(div;\Omega)$. La relation (1) est donc valable pour $\vec{u}g$, f et g tel que définis. Ceci nous permet donc d'écrire la formule d'intégration par parties suivante:

$$\int_{\Omega}(\vec{u}.\nabla f)\, g\, dX + \int_{\Omega}(\nabla.\vec{u}\, g)\, f\, dX = \int_{\partial\Omega} f\, g\, (\vec{u}.\vec{n})\, dS \qquad (2)$$

où les fonctions f et f^- peuvent être discontinues. La fonction f peut même être, du fait d'un certain champ de vitesse, discontinue dans des cas où f^- est continue Ceci sera exclu en choisissant une solution laminaire de Stokes => $\vec{u} \in H(div;\Omega)$. On doit donc rechercher une solution pouvant contenir un certain niveau "d'irrégularité". Pour une triangulation $T(\Omega_i)$ on construit l'espace « irrégulier » suivant:

$$V \equiv \prod (H^1(\Omega_i)) = \left\{ f \in L^2(\Omega_i) \mid f_{|\Omega_i} \in H^1(\Omega_i), \forall \Omega_i \in T(\Omega_i) \right\},$$

dans lequel on cherchera une solution.

Comme $H^1(\Omega) \subset \prod_{i=1}^{n} H^1(\Omega_i)$, l'équation (2) s'étend à V et comme $\int_{\Omega} \vec{u}.\nabla f\, g = 0$, on peut écrire:

$$\int_{\Omega} f\, \nabla.(\vec{u}g)\, dX = \int_{\partial\Omega^-} f\, g\, \vec{u}.\vec{n}\, dS$$

Comme on connaît la valeur de f sur $\partial\Omega^-$ (on la note f), cette équation devient:

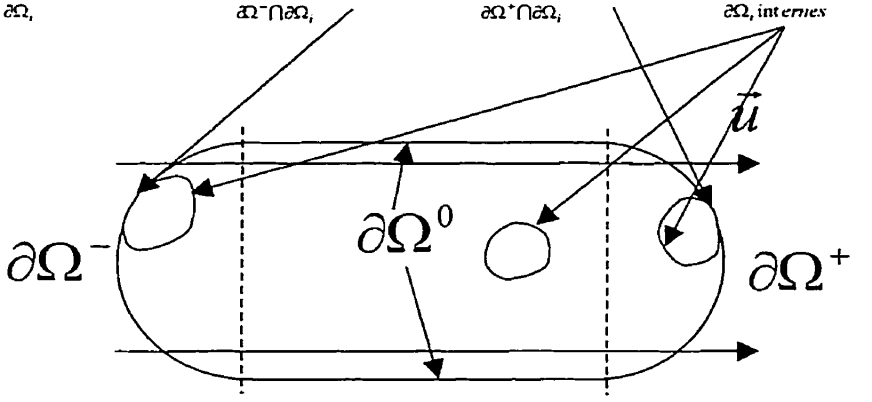
$$\int_{\Omega} f\,\nabla.(\vec{u}g)\,dX = \int_{\partial\Omega^-} f^-\,g\,\vec{u}.\vec{n}\,dS + \int_{\partial\Omega^+} f\,g\,\vec{u}.\vec{n}\,dS$$

On intégre cette équation par parties sur chacun des éléments Ω_i du maillage, en utilisant l'égalité (3) (valable sur chaque Ω_i) et en décomposant la frontière en $\partial\Omega^-$ et $\partial\Omega^+$, on obtient :

$$\sum_i \left[-\int_{\Omega_i} (\vec{u}.\nabla f)\,g + \int_{\partial\Omega_i} f\,g\,(\vec{u}.\vec{n})\,dS \right] = \sum_i \left(\int_{\partial\Omega^- \cap \partial\Omega_i \neq \phi} f^-\,g\,\vec{u}.\vec{n}\,dS + \int_{\partial\Omega^+ \cap \partial\Omega_i \neq \phi} f\,g\,\vec{u}.\vec{n}\,dS \right)$$

Comme $\vec{u}\cdot\vec{n} = 0$ sur $\partial\Omega_i^0$ on ne tiendra pas compte de leurs contributions dans la suite des calculs. La deuxième intégrale dans la parenthèse se décompose de la manière suivante :

$$\int_{\partial\Omega_i} f\,g\,(\vec{u}.\vec{n})\,dS = \int_{\partial\Omega^- \cap \partial\Omega_i} f\,g\,(\vec{u}.\vec{n})\,dS + \int_{\partial\Omega^+ \cap \partial\Omega_i} f\,g\,(\vec{u}.\vec{n})\,dS + \int_{\partial\Omega_i,\ interne} f\,g\,(\vec{u}.\vec{n})\,dS$$



Dans le cas des portions de frontière $\partial\Omega_i$ situées complètement à l'intérieur de Ω (comme le cas des sous-domaines situés complètement à l'intérieur de Ω montrés sur la figure ci-dessus) les intégrales

$$\int_{\partial\Omega_i^- \cap \partial\Omega} f\, g\,(\vec{u}.\vec{n})\, dS + \int_{\partial\Omega_i^+ \cap \partial\Omega} f\, g\,(\vec{u}.\vec{n})\, dS$$

sont identiquement nulles. En introduisant cette décomposition dans l'équation précédente et en multipliant par -1 on obtient :

$$\sum_i \left[\int_{\Omega_i} (\vec{u}.\nabla f)\, g - \left(\int_{\partial\Omega^- \cap \partial\Omega_i} f\, g\,(\vec{u}.\vec{n})\, dS + \int_{\partial\Omega^+ \cap \partial\Omega_i} f\, g\,(\vec{u}.\vec{n})\, dS + \int_{\partial\Omega_i \text{ int ernes}} f\, g\,(\vec{u}.\vec{n})\, dS \right) \right]$$

$$= -\left(\sum_i \int_{\partial\Omega^- \cap \partial\Omega_i \neq \phi} f^-\, g\,(\vec{u}.\vec{n})\, dS + \sum_i \int_{\partial\Omega^+ \cap \partial\Omega_i \neq \phi} f\, g\,(\vec{u}.\vec{n})\, dS \right)$$

Maintenant en regroupant les termes constituant les frontières $\partial\Omega^-$ et $\partial\Omega^+$, on obtient :

$$\int_{\Omega_i} (\vec{u}.\nabla f)\, g d\Omega - \sum_{\partial\Omega_i \text{ int ernes}} \int_{\partial\Omega_i} f\, g\,(\vec{u}.\vec{n})\, dS - \sum_i \int_{\partial\Omega^- \cap \partial\Omega_i \neq \phi} (fg\,(\vec{u}.\vec{n})\, dS = -\sum_i \int_{\partial\Omega^- \cap \partial\Omega_i \neq \phi} f^-\, g\,(\vec{u}.\vec{n})\, dS ,$$

où

$$\int_{\Omega_i} (\vec{u}.\nabla f)\, g\, d\Omega = \sum_i \int_{\Omega_i} (\vec{u}.\nabla f)\, g d\Omega$$

et finalement,

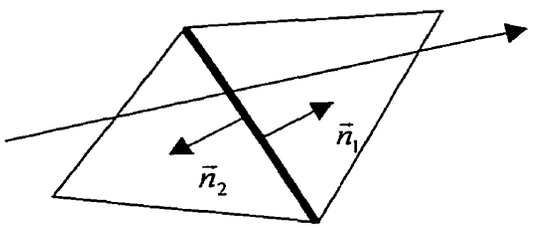
$$\int_{\Omega_i} (\vec{u}.\nabla f)\, g d\Omega - \sum_{\partial\Omega_i \text{ int ernes}} \int_{\partial\Omega_i} f\, g\,(\vec{u}.\vec{n})\, dS - \int_{\partial\Omega^-} fg\,(\vec{u}.\vec{n})\, dS = -\int_{\partial\Omega^-} f^-\, g\,(\vec{u}.\vec{n})\, dS .$$

Or

$$\sum_{\partial\Omega_i \text{ int ernes}} \int_{\partial\Omega_i} f\, g\,(\vec{u}.\vec{n})\, dS = \sum_{\partial\Omega_i \text{ int ernes}} \sum_{F\in\partial\Omega_i} \int_F f\, g\,(\vec{u}.\vec{n})\, dS = \sum_{F\in\partial\Omega_i} \sum_{\partial\Omega_j \cap \partial\Omega_i = F} \int_F f\, g\,(\vec{u}.\vec{n})\, dS$$

où F sont les faces de $\partial\Omega_i$ Or dans un maillage d'éléments finis il n'y a que deux éléments adjacents à chaque face. On a donc:

$$\sum_{F\in\partial\Omega_i} \sum_{\partial\Omega_j \cap \partial\Omega_i = F} \int_F f\, g\,(\vec{u}.\vec{n})\, dS = \sum_{F\in\partial\Omega_i} \left(\int_F f\, g\,(\vec{u}.\vec{n}_1)\, dS + \int_F f\, g\,(\vec{u}.\vec{n}_2)\, dS \right)$$



où $\vec{n}_1$ et $\vec{n}_2$ sont les normales aux éléments adjacents à F qui pointent dans des directions opposées (voir figure). Si l'on choisit sur F la normale de sorte que $\vec{u}\cdot\vec{n} < 0$, alors on obtient finalement :

$$\int_\Omega (\vec{u}.\nabla f)\,|_{L^2}\; g d\Omega - \sum_{F\in\partial\Omega_i} \int_F \left(f - f^-\right) g\,(\vec{u}.\vec{n}_1)\, dS - \int_{\partial\Omega^-} f g\,(\vec{u}.\vec{n})\, dS = - \int_{\partial\Omega^-} f^- g\,(\vec{u}.\vec{n})\, dS\,.$$

Comme on a éliminé la normale correspondant à un coté sortant il ne reste plus qu'à sommer sur les $\partial\Omega_i^-$ pour obtenir :

$$\int_\Omega (\vec{u}.\nabla f)\,|_{L^2}\; g d\Omega - \sum_i \int_{\partial\Omega_i^- \cap \partial\Omega^- \neq \phi} \left(f - f^-\right) g\,(\vec{u}.\vec{n}_1)\, dS - \int_{\partial\Omega^-} f g\,(\vec{u}.\vec{n})\, dS = - \int_{\partial\Omega^-} f^- g\,(\vec{u}.\vec{n})\, dS$$

En choisissant des

$$g \in \prod_i H^1(\Omega_i)$$

convenablement on peut résoudre successivement d'élément en élément cette dernière équation ([4]) pour obtenir finalement :

Trouver $f_h \in V_h$ solution de

$$P'': \begin{cases} \forall \Omega_i \in T(\Omega), \\ \int_{\Omega_i} (\vec{u}_h . \nabla f_h) g_h \, d\Omega - \int_{\partial\Omega_i^-} f_h g_h \, (\vec{u}_h . \vec{n}) \, dS = - \int_{\partial\Omega_i^-} f^- g_h \, (\vec{u}_h . \vec{n}) \, dS, \forall g_h \in V_h \end{cases}$$

Équation 9 – Forme faible de l'équation de Lesaint-Raviart calculable élément par élément

qui est l'équation que nous auront à résoudre successivement, élément par élément. Ici f désigne la valeur de la fonction f sur $\partial\Omega_i$. Les arêtes constituant $\partial\Omega_i^-$ peuvent être soit des arêtes entrantes supportant la condition initiale $\partial\Omega_i^-$, soit des arêtes d'un élément interne touchant à des éléments adjacents déjà calculés. Il sera par conséquent nécessaire pour calculer un élément d'avoir au préalable calculé toutes ses arêtes entrantes, et donc tous les éléments voisins connectés par une arête entrante de l'élément considéré.

9.4.3 Problèmes posés par Lesaint-Raviart

L'implémentation de la formulation Lesaint-Raviart pose deux problèmes :

1. Un élément ne pouvant être calculé que lorsque toutes ses arêtes entrantes l'ont été, il est nécessaire de définir un ordre de calcul des éléments (on définit comme entrante une arête dont le flux de vitesse est positif).

2. La présence d'un terme de bord inusuel, pour lequel il est nécessaire de récupérer des valeurs précédemment calculées sur l'élément voisin ou dans la condition initiale.

9.4.3.1 Définition d'un ordre de calcul des éléments

On doit pour cela commencer par résoudre une formulation permettant de calculer la vitesse du fluide en tout point de la géométrie, celle de Navier-Stokes par exemple.

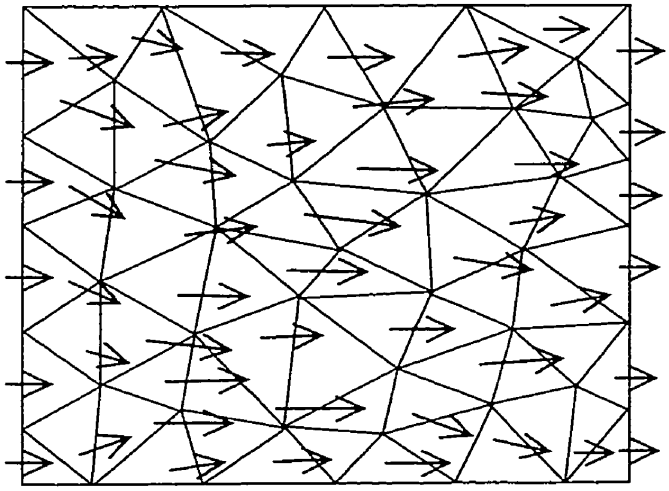


Figure 9.6 - Exemple de champ de vitesse

Une fois que l'on dispose du champ de vitesse il faut, en partant de tous les bords entrants, déterminer l'ordre de calcul des éléments selon le critère suivant : ***un élément peut être calculé quand toutes ses arêtes entrantes ont été calculées***. On marque alors toutes les arêtes de cet élément comme ayant été calculées.

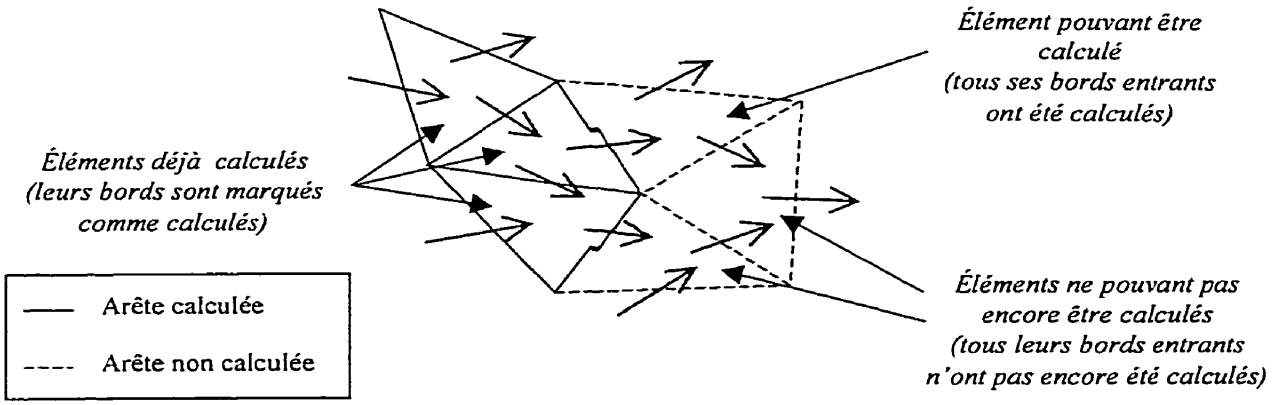
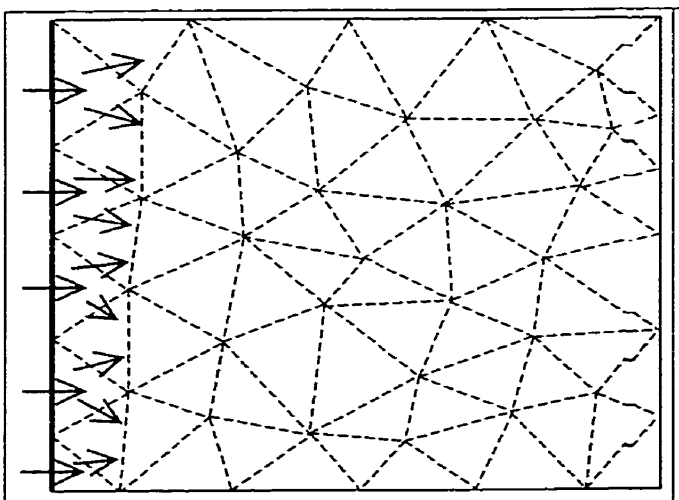
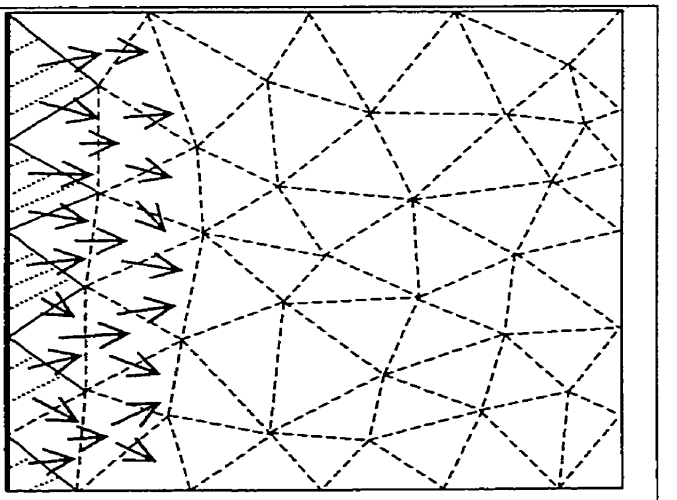


Figure 9.7 - Critère de calcul d'un élément



Départ : on impose une vitesse constante et entrante sur le bord gauche de la boîte. On définit également sur le bord la condition aux limites de la quantité à transporter



Tous les éléments ayant tous leurs bords entrants calculés (les arêtes où est fixé la condition aux limites sont considérées comme calculées) sont calculés

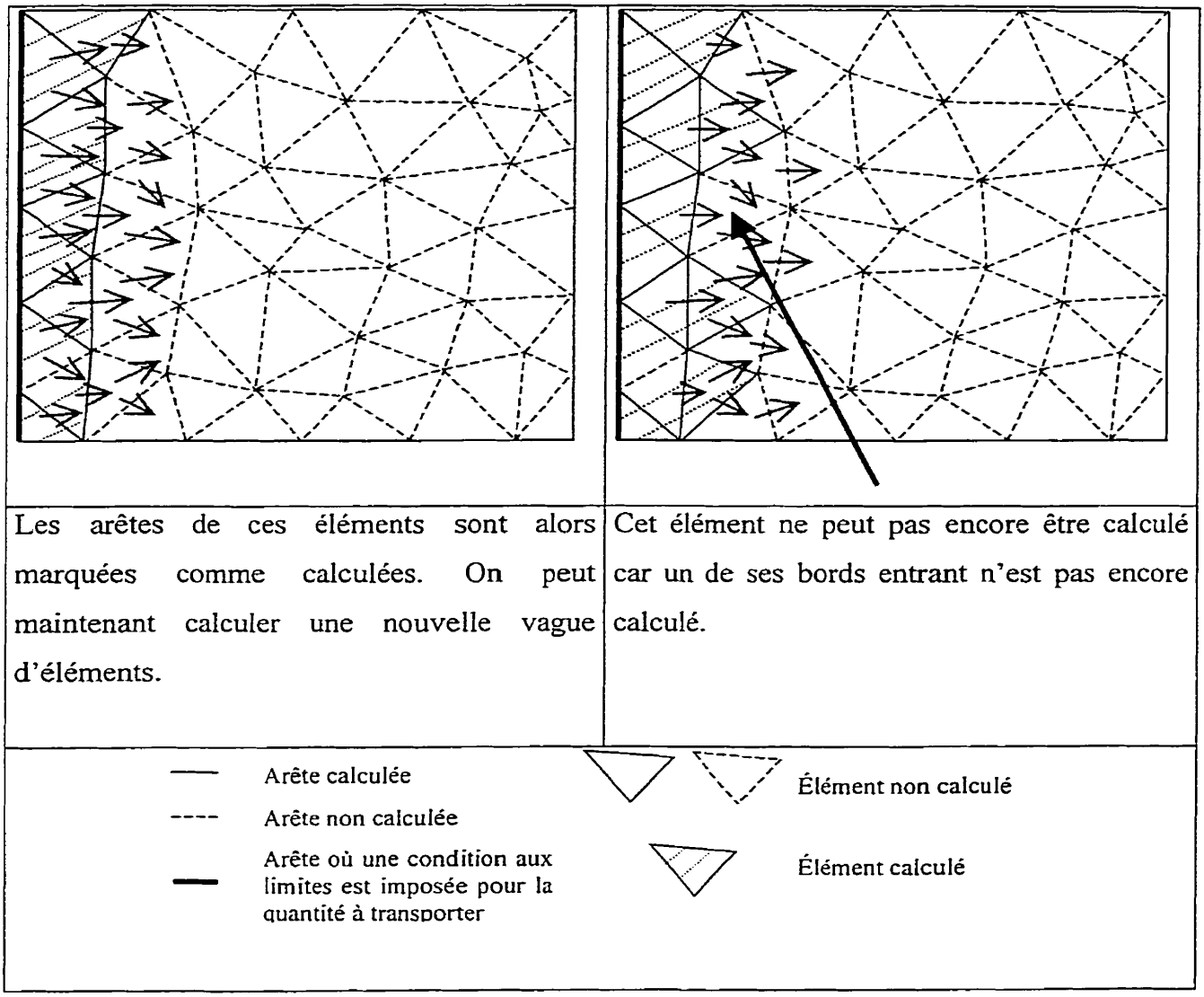


Figure 9.8 - Déroulement partiel d'un exemple de remplissage

On retient ici une approche où l'ordre de calcul des éléments n'est pas prédéterminé, mais est plutôt géré par la notion de **front**. On définit le front comme étant l'ensemble des arêtes entrantes ayant été calculées d'un élément non calculé. Pour des raisons d'efficacité, on définit l'associations éléments non calculés – arêtes entrantes calculées dans une table de hashage dont la clé est l'élément et dont le contenu est la liste des arêtes de cet élément ayant été calculées. L'initialisation du front est effectuée par

concaténation des couples élément – arête supportant géométriquement une condition initiale de la quantité à transporter et pour lesquels le flux dans l'élément est entrant. On rappelle que les conditions initiales sont définies sur des bords et que les bords sont des ensembles de paires élément de volume – face de l'élément de volume. On lance ensuite le calcul. Pour chacun des éléments du front, on teste si toutes ses arêtes entrantes sont situées sur le front. Si oui, on calcule l'élément, sinon on insère cet élément dans un second front contenant les éléments retardés. Une fois le front courant calculé, on doit définir le nouveau front. Pour cela, on commence par insérer dans le nouveau front les éléments dont le calcul a été retardé. On insère ensuite dans ce nouveau front les arêtes sortantes des éléments venant d'être calculés dans l'entrée correspondant à l'élément voisin.

On itère ensuite jusqu'à stabilité du front. Deux problèmes peuvent surgir lorsque l'on utilise cet algorithme :

- certains éléments ne seront jamais calculés si toutes leurs arêtes sont sortantes. Il est donc nécessaire de les détecter préalablement et de les signaler ;

- l'algorithme peut boucler dans le cas de recirculations. Comme l'on désire ici uniquement démontrer la facilité d'adaptation du code élément fini, on ignore délibérément ce problème .

On choisit ici délibérément des écoulements laminaires pour éviter ces problème.

```
front courant,nouveau,éléments_retardés
courant = somme(bords entrants)
faire toujours
      pour tous les éléments du front courant
            si toutes les arêtes entrantes ont été calculées
```

```
                calculer l'élément
            sinon
                insérer l'élément dans éléments_retardés
            fin si
        fin pour tous
        nouveau = éléments_retardés
        pour tous les éléments de courant
            si l'élément n'est pas dans le front des éléments retardés
                insérer ses arêtes sortantes et ses éléments voisins
                correspondants dans nouveau
            finsi
        fin pour tous
        si courant est différent de nouveau
            courant = nouveau
            continuer
        sinon
            arrêter
fin faire toujours
```

Algorithme de gestion du front

Note : Afin de ne pas calculer plusieurs fois le flux à travers une arête, on calcule, avant d'effectuer toute autre action, les flux à travers toutes les arêtes.

9.4.3.2 Terme de bord inusuel

Le terme de bord LesaintRaviartBoundaryFunctionSourceTerm de la formulation Lesaint-Raviart requiert le calcul du flux entrant d'une quantité dans un élément. Deux cas se présentent :

- il est nécessaire de récupérer la valeur de cette quantité en un point d'un élément voisin (1^{er} cas);

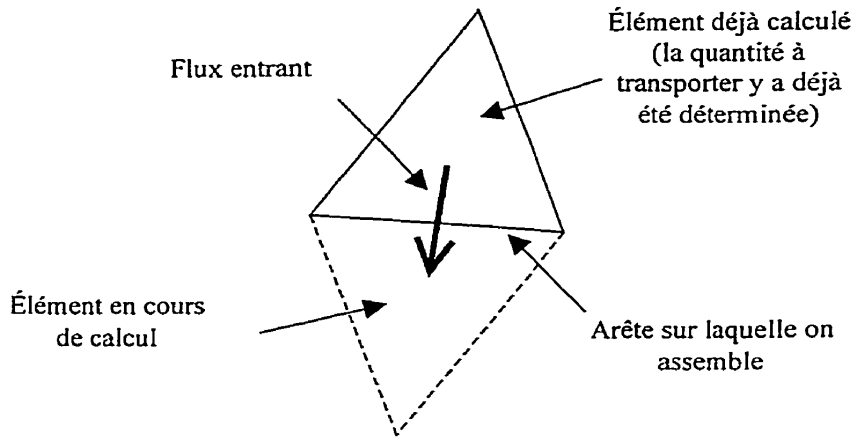


Figure 9.9 – 1^{er} case : il faut extraire la valeur de la quantité à transporter d'un élément voisin

- il faut utiliser la valeur d'une condition aux limites imposée sur cette arête (2^{nd} cas).

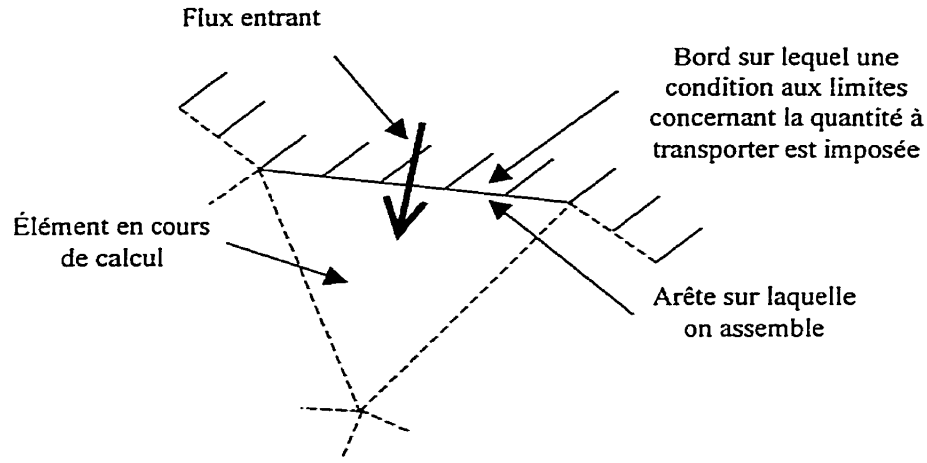


Figure 9.10 - 2ième cas : il faut utiliser la valeur de la condition aux limites

Le 1er cas est un assemblage de terme de bord triliniéaire doublement contracté faisant intervenir deux contracteurs. Le premier est `QuantityScalarNormal`, utilisé ici pour extraire la vitesse du champ que l'on utilise pour transporter la quantité et calculer $\vec{U}.\vec{n}$. Le second est `QuantityLastStep` récupérant la valeur de la quantité dans l'élément voisin.

Le 2nd cas est également un assemblage de terme de bord trilinéaire doublement contracté. Cependant, si le contracteur `QuantityScalarNormal` est toujours utilisé pour calculer $\vec{U}.\vec{n}$, la seconde contraction est effectuée non pas par une multiplication avec le résultat de l'évaluation d'un contracteur, mais par une multiplication par l'évaluation de la condition aux limites (une instance d'une classe dérivée de `BoundaryCharacteristic`) en cet élément.

On définit donc deux assembleurs de matrices élémentaires, un pour chaque cas.

Le code de ce terme particulier est le suivant :

```
integratorCurrent.SetCurrentElement(&geoelemCurrentVolume);
```

Notification de l'élément de volume courant à l'intégrateur.

```
integratorCurrent.SetCoordChangeElement( \\
      seCurrentFace.lpgeoelemSimplexedXYZ);
```

Notification de l'élément à utiliser pour calculer le changement de coordonnées. On passe l'élément de bord puisqu'il est le support de l'intégration

```
integratorCurrent.SetFunctionSpace(&fsdCurrent);
```

Notification de l'espace fonctionnel courant à l'intégrateur

```
integratorCurrent.SetBoundaryIntegrationPoints(&seCurrentFace);
```

Demande à l'intégrateur de générer la liste des points d'intégration sur l'élément de bord

```
      QuantityScalarNormal qsnCurrentOnBoundary(lpDofDataSpeed,
lppqSpeed,
            -1);
      QuantityScalarNormal
qsnCurrentOnElement(lpDofDataSpeed,lppqSpeed,1);
      QuantityLastStep qlsCurrent(lpDofData,lppqQuantity);
```

Déclaration des contracteurs. Le premier récupérera la vitesse et calculera son produit scalaire par la normale à l'arête dans l'élément de volume que l'on calcule actuellement : il sera utilisé dans le cas où l'arête supportant l'intégrale de bord fait partie du support géométrique d'une condition aux limites. Le second servira pour récupérer la vitesse dans l'élément voisin puis calculer sa normale à l'arête. Le troisième paramètre du constructeur de QuantityScalarNormal est un multiplicateur appliqué au résultat du produit scalaire. Enfin, le troisième contracteur récupère la valeur de la quantité dans l'élément géométrique voisin.

```
vector<geometrical_element*>::iterator veclpgeoelemitParent =
      seCurrentFace.lpgeoelemSimplexedXYZ->parents_begin();
vector<geometrical_element*>::iterator veclpgeoelemitParentsEnd =
            seCurrentFace.lpgeoelemSimplexedXYZ->parents_end();
if((*veclpgeoelemitParent != &geoelemCurrentVolume) ||
      ((++veclpgeoelemitParent != veclpgeoelemitParentsEnd) &&
      (*veclpgeoelemitParent != &geoelemCurrentVolume)))
```

Détermination de l'élément géométrique voisin.
S'il y en a un :

```
{
```

```
qsnCurrentOnElement.SetCurrentElement(*veclpgeoelemitParent
);
qsnCurrentOnElement.SetCurrentFaceElement(
      seCurrentFace.lpgeoelemSimplexedXYZ);
qlsCurrent.SetCurrentElement(*veclpgeoelemitParent);
```

Notification de l'élément voisin aux deux contracteurs et de l'arête courante au contracteur de type QuantityScalarNormal.

```
ElementaryTrilinearTwiceContracted
ettcCurrent(*lppqQuantity,
      &qsnCurrentOnElement,
      &qlsCurrent,
      genparamsCurrent.dTime);
```

Déclaration de l'assembleur de matrices élémentaires.

```
integratorCurrent.AssembleLinearMatrix(&ettcCurrent,
      &veccoefRetvalue,genparamsCurrent.dTime);
```

Calcul de la matrice élémentaire.

```
}

else{
```

S'il n'y a pas d'élément voisin :

```
BCNamesRegistry::iterator beginBoundaries =
      GlobalBCNamesRegistry.begin();
BCNamesRegistry::iterator endBoundaries =
      GlobalBCNamesRegistry.end();

for(;beginBoundaries != endBoundaries;beginBoundaries++){
      if((*beginBoundaries).GetPhysType() ==
            lppqQuantity->GetCode())
      {
            Boundary* lpboundaryToTest =
                  meshCurrent.GetBoundaryWithId(\\
                        (*beginBoundaries).GetGeomId());
            if(find(lpboundaryToTest->volumes_begin(),
                  lpboundaryToTest->volumes_end(),
                  &geoelemCurrentVolume) !=
                        lpboundaryToTest->volumes_end())
            {
```

Il faut retrouver la bonne condition aux limites. Pour cela on cherche les conditions aux limites concernant la quantité physique à transporter `(if((*beginBoundaries).GetPhysType() == lppqQuantity->GetCode())` puis l'on teste pour chacune de ces quantités si elle contient l'élément courant.

```
                                qsnCurrentOnBoundary.SetCurrentElement(
                                    &geoelemCurrentVolume);

            qsnCurrentOnBoundary.SetCurrentFaceElement(

            seCurrentFace.lpgeoelemSimplexedXYZ);
```

Notification de l'élément de volume et de l'arête courante au contracteur calculant $\bar{U}.\bar{n}$

```
                        ElementaryTrilinearTwiceContractedWithBou
                ndaryCharacteristic
                ettcwbcCurrent(*lppqQuantity,
                        *lpboundaryToTest,
                        (*beginBoundaries),
                        &qsnCurrentOnBoundary,
                        genparamsCurrent.dTime);
```

Déclaration de l'assembleur de matrice élémentaire. Noter le passage en paramètre du contracteur (`qsnCurrentOnBoundary`), de la limite (`lpboundaryToTest`) et de la caractéristique de condition aux limites à utiliser (`*beginBoundaries`).

```
                                integratorCurrent.AssembleLinearMatrix(
                                    &ettcwbcCurrent, &veccoefRetvalue,
                                    genparamsCurrent.dTime);
```

Calcul de la matrice élémentaire.

```
                                break;
```

Arrêt de la recherche dans les conditions aux limites.

```
                            }

                        }

                }

        }
```

L'assemblage dans le gestionnaire de degrés de liberté est tout à fait classique et n'est donc pas reproduit ici.

9.4.4 Formulation Lesaint-Raviart

On ne reproduit pas ici le code de la formulation Lesaint-Raviart. Il est en effet constitué d'assemblages de termes, comme toutes les autres formulations, d'impositions de

conditions aux limites avec en prime le code de gestion du front, dont la présentation du code source n'apporterait rien à la compréhension. Un détail en revanche doit être souligné : la *formulation Lesaint-Raviart* a cela d'intéressant qu'elle *peut être calculée localement sur chaque élément*. Il faut donc assembler les termes non pas tour à tour sur l'ensemble des éléments mais élément par élément, en résolvant un petit système d'équation pour chaque élément. Heureusement, la fonction principale des termes (celle qui est appelé pour demander un assemblage) est fournie en deux versions, dont une ne fait l'assemblage que sur un unique élément passé en paramètre en lieu et place d'un domaine. Il suffit donc d'appeler cette version pour tous les termes pour un élément donné puis de demander au résoluteur de solutionner le système ainsi assemblé. Le gestionnaire de degrés de liberté disposant d'un mode spécial permettant de ne mettre dans la matrice globale que les degrés de liberté récemment assemblés tout en conservant les degrés de liberté calculés lors des résolutions précédentes, on résout ainsi for élégamment notre problème.

9.5 La méthode d'intégration récursive de Romberg

9.5.1 Introduction

La méthode de Romberg est une méthode d'intégration numérique permettant d'obtenir une grande précision. Après l'avoir brièvement présentée dans le cas 1D, et ainsi constaté à quel point elle est intrinsèquement récursive, on présentera les grandes lignes de son implémentation.

9.5.2 Brève description mathématique

La méthode de Romberg est basée sur la méthode des trapèzes composée et la méthode d'extrapolation de Richardson. On pourra trouver une description complète de ces deux

méthodes ainsi que de celle de Romberg dans « Analyse Numérique pour Ingénieurs » d'André Fortin [5].

On note $T_{1,i}$ le résultat obtenu à l'aide de la méthode des trapèzes composée avec 2^{i-1} intervalles. Les $T_{1,i}$ sont des approximations d'ordre 2. En utilisant l'extrapolation de Richardson on définit alors :

$$T_{2,i} = \frac{2^2 T_{1,i+1} - T_{1,i}}{2^2 - 1}$$

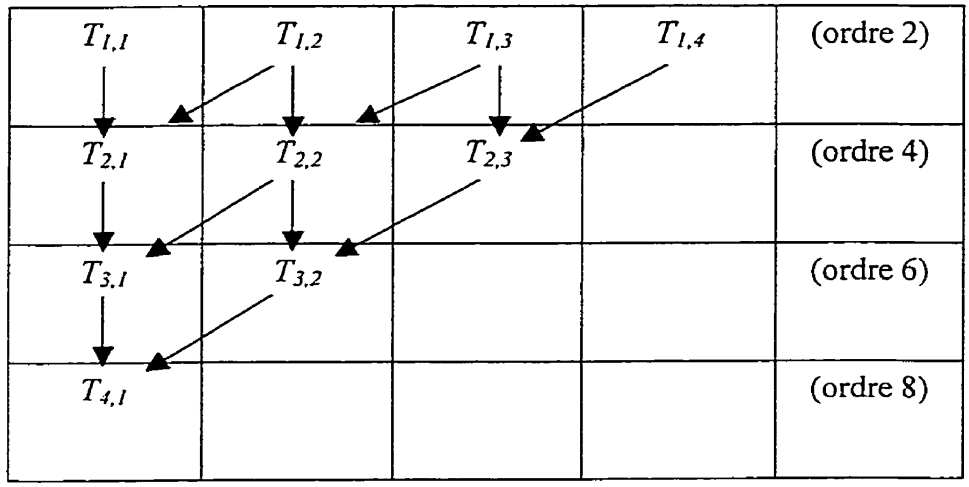
Où les $T_{2,i}$ sont des interpolations d'ordre 4.

De manière plus générale :

$$T_{n,i} = \frac{2^{2(n-1)} T_{n-1,i+1} - T_{n-1,i}}{2^{2(n-1)} - 1}$$

Ce qui définit le triangle d'évaluation suivant :

Tableau 9.1 - Triangle d'évaluation par la méthode de Romberg



On peut constater à quel point la méthode de Romberg est bien adaptée à l'intégration récursive. Il suffit en effet, si le critère de convergence n'est pas atteint, de calculer avec la méthode des trapèzes l'intégrale sur deux fois plus d'éléments et d'utiliser ce résultat pour calculer un nouveau résultat d'ordre 4, puis 6, etc. jusque l'ordre N souhaité.

Cette méthode se généralise de deux manières. Tout d'abord, on peut utiliser n'importe quel intégrateur sur une partition contenant deux fois plus d'éléments. Ensuite, on peut généraliser cette méthode à autre chose que des éléments unidimensionnels, en utilisant une simplexation de l'élément géométrique. Ainsi, dans le cas d'un triangle, on procédera à la division suivante :

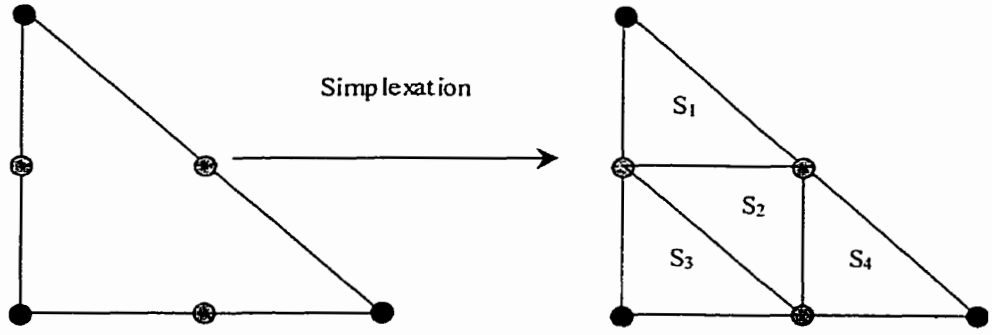


Figure 9.11 – Simplexation d'un élément triangulaire

On intégrera ici sur les éléments S_1, S_2, S_3 et S_4 pour calculer $S_{n,i+1}$. On procède de même quelque soit le type d'élément.

9.5.3 Implémentation

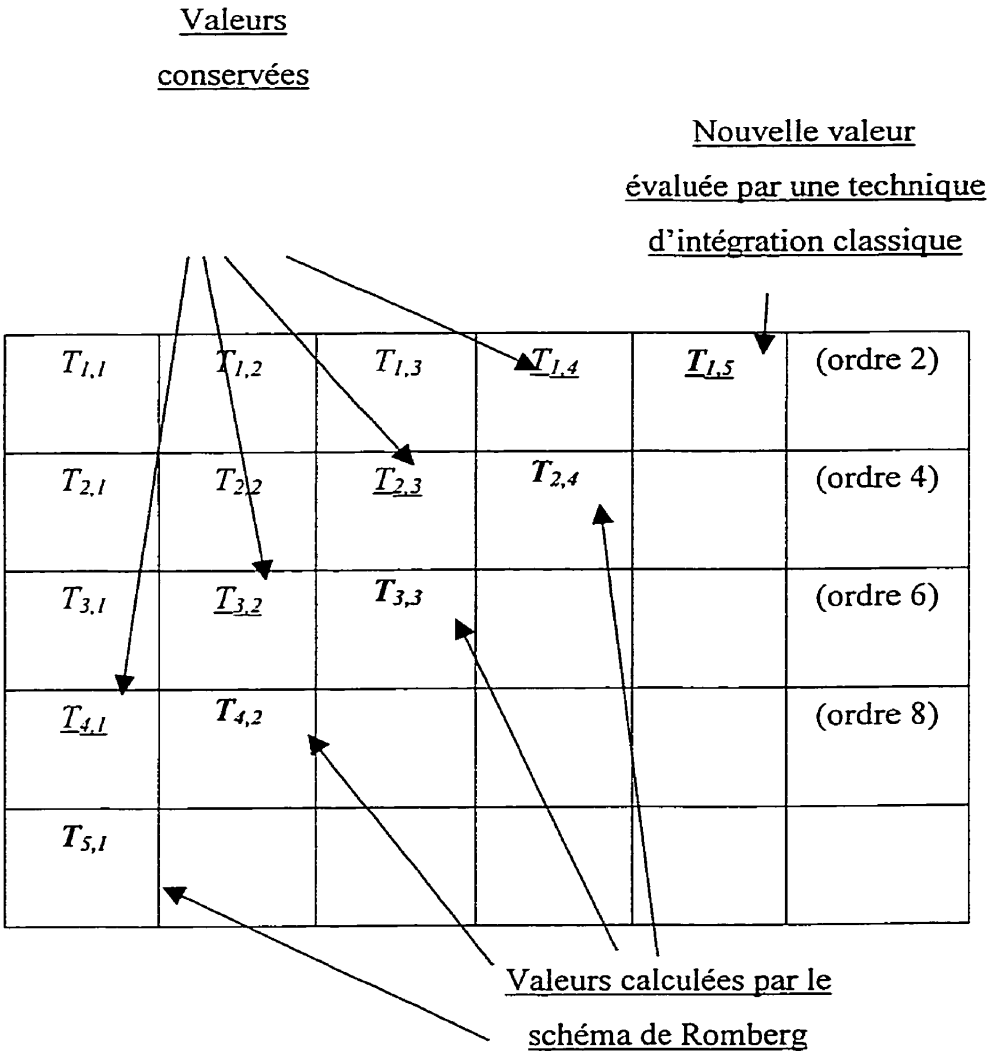
L'intégrateur de Romberg est implémenté, comme tous les intégrateurs récursifs doivent l'être, en quatre parties :

- un évaluateur
- un critère d'arrêt
- un schéma de récursion
- une simplexation de chaque type d'élément géométrique

Comme expliqué dans la partie décrivant les intégrateurs récursifs d'une manière générale, la simplexation est directement codée dans les éléments géométriques. Il ne semble pas exister en effet beaucoup de simplexations correctes des différents types

d'éléments. Il nous reste donc à écrire à priori un critère d'arrêt, une évaluation et un schéma de récursion. Or, la méthode de Romberg ne concerne que le schéma de récursion. Le mode d'évaluation et la nature exacte du critère d'arrêt nous importent peu.

On constate en étudiant attentivement le triangle d'évaluation de Romberg que seule une donnée par ligne doit être conservée : celle correspondant à la cellule la plus à droite. Lors du calcul d'un nouvel ordre de précision, on ne devra ainsi calculer que la nouvelle valeur $T_{1,N/2}$ puis calculer $T_{2,N/2-1}, \ldots, T_{N/2,1}$, N étant le degré de précision.



Le code de ce schéma de récursion est très simple, mais est cependant compliqué par de nombreuses considérations quant à la gestion efficace de la mémoire. Aussi préfère-t-on ne pas le détailler ici. Une fois ce schéma codé, on peut l'utiliser avec n'importe quel critère d'arrêt et n'importe quel évaluateur.

9.5.4 Intégration récursive de la fonction de Gauss

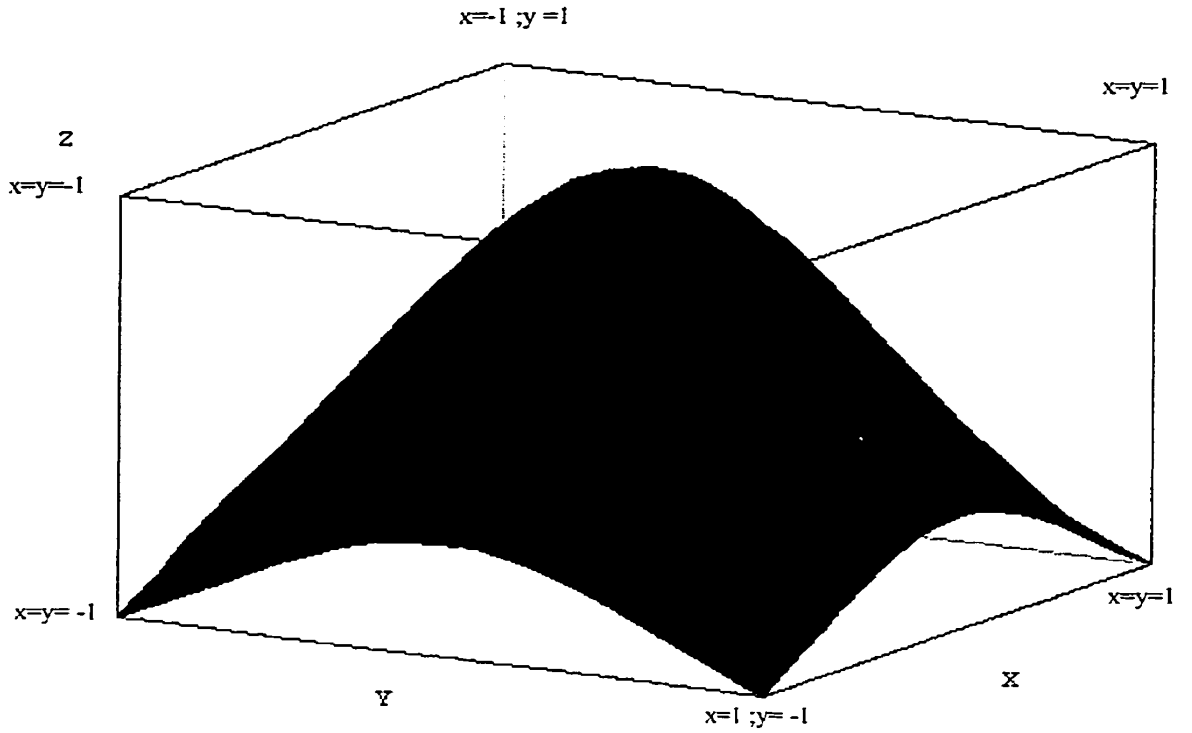


Figure 9.12 – $e^{-x^2-y^2}$

On désire calculer l'intégrale de $e^{-x^2-y^2}$ sur [-1;1]x[-1;1]. On va pour cela intégrer sur un élément triangulaire de coordonnées ((-1,-1);(1,-1);(-1,1)). On calcule donc en fait la moitié de cette intégrale. La fonction exponentielle étant analytique, il est nécessaire d'utiliser un schéma adaptatif pour calculer son intégrale. On utilise ici le schéma de Romberg pour piloter la récursion et l'intégrateur de Gauss pour calculer l'intégrale sur chaque élément de la simplexation. On procède de deux manières différentes. La première consiste à définir la fonction $e^{-x^2-y^2}$ comme une fonction analytique et à utiliser le maximum de points d'intégration de Gauss à notre disposition, soit 12 points

de Gauss. La seconde consiste à ne prendre qu'un seul point par élément lors du calcul de l'intégrale d'un élément.

Afin de procéder à l'évaluation de cette intégrale, on a choisi d'utiliser un terme bilinéaire avec loi (la loi étant unitaire), de définir une fonction de forme Gaussienne, d'utiliser la fonction de forme constante comme fonction test et les fonctions de forme linéaires de Lagrange comme fonctions de forme géométrique. On utilise, comme expliqué précédemment, un intégrateur de Gauss pour procéder à l'évaluation de l'intégrale sur un élément de la partition et le schéma récursif de Romberg pour piloter la récursion. L'utilisation d'un intégrateur récursif implique la définition de deux autres objets : un critère d'arrêt et un évaluateur. L'évaluateur est ici un simple enrobage de l'intégrateur de Gauss Il est capable d'appeler de dernier pour tous les éléments d'un ensemble d'éléments à calculer et d'additionner les résultats. Le critère d'arrêt retenu est très simple : on impose que la différence entre $T_{N,I}$ et $T_{N-I,I}$ soit inférieure à une valeur paramétrable. Afin d'obtenir la plus grande précision possible, on fixe cette différence à une valeur inférieure à la précision machine (10^{-12}).

```
GaussFunction gfTest;
```

instanciation de la fonction de forme Gaussienne

```
constant_shape_function csfCurrent;
```

instanciation de la fonction de forme constante

```
FunctionSpaceData_c fsdCurrent(&gfTest,
                               false,
                               &csfCurrent,
                               false,
                               &gsfLinear);
```

instanciation de l'espace fonctionnel

```
geometrical_element* lpgeoelemTriangle = NULL;
lpgeoelemTriangle  = *maillage.GetAllElements()->begin();
```

on prend le premier élément du maillage

```
GaussIntegrator MyVeryVeryBeautifulGaussIntegrator(lpgeoelemTriangle,
                                    &fsdCurrent);
```

instanciation de l'intégrateur de Gauss

```
PhysicLaw plCurrent(1,1);
plCurrent.GetCoefficient()[0] =  1.;
```

instanciation et initialisation de la loi physique

```
ElementaryBilinearWithLaw ebwlCurrent(&fsdCurrent);
```

instanciation de l'assembleur de matrice élémentaire à utiliser

```
GaussEvaluator geCurrent(&MyVeryVeryBeautifulGaussIntegrator);
```

instanciation d'un évaluateur enrobant l'intégrateur de Gauss

```
GaussStopCriterion gscCurrent(1e-14);
```

déclaration du critère d'arrêt. On l'initialise avec un écart à respecter inférieur à la précision machine

```
integration_result result;
```

matrice élémentaire résultant de l'intégration

```
RombergSimplexation<GaussEvaluator,GaussStopCriterion>
simplexCurrent(geCurrent,
                  gscCurrent,
                  maillage,
                  iProfondeurMax);
```

instanciation d'un schéma récursif de Romberg. Le dernier paramètre passé est la profondeur maximale de récursion

```
simplexCurrent.SetFunctionSpace(&fsdCurrent);
simplexCurrent.SetCurrentElement(lpgeoelemTriangle);
```

notification de l'espace fonctionnel courant et de l'élément courant au schéma de Romberg. Celui-ci notifie à son tour l'évaluateur afin qu'il puisse notifier l'intégrateur

```
simplexCurrent.AssembleBilinearMatrix(&plCurrent,
                                  &ebwlCurrent,
                                  &result);
```

enfin, évaluation de l'intégrale

On obtient ainsi les résultats suivants :

Tableau 9.2 – Valeurs obtenues pour l'intégration de la gaussienne en utilisant un maximum de points d'intégration

Niveau de récursion	Valeur obtenue	Temps d'exécution	Erreur
2	1.4641862025297	0.001	$1.76*10^{-7}$
3	1.4641861997805	0.006	$1.11*10^{-9}$
3	1.4641861997805	0.006	1.11*10-9
4	1.4641861999451	0.025	$1.36*10^{-10}$
5	1.4641861999431	0.1042	$<10^{-12}$

Tableau 9.3 – Valeurs obtenues pour l'intégration de la gaussienne en utilisant un point d'intégration par élément

Niveau de récursion	Valeur obtenue	Temps d'exécution	Erreur*
2	1.4642364173854	0.0001	$5*10^{-3}$
3	1.4641861940610	0.003	$4.01*10^{-7}$
4	1.4641861997365	0.011	$1.41*10^{-8}$
5	1.4641861999959	0.0451	$3.6*10^{-9}$
6	1.4641861999959	0.2063	$3.6*10^{-9}$
7	1.4641861999958	0.2103	$3.59*10^{-9}$

Les résultats semblent plus précis avec la méthode utilisant un maximum de points de Gauss et ce, en un mininum de temps. Cela est très certainement dû au fait que le rassemblement des résultats sur une récursion de niveau 7 requiert 4^2 additions de matrices de plus que le résultat de niveau 5. On doit également obtenir des valeurs extrêmement petites, et comme l'addition n'est plus commutative lorsqu'elle est effectuée numériquement pour des valeurs très petites, cela induit la légère erreur constatée.

CONCLUSION

L'objectif de ce travail était la conception et le développement d'un moteur de calcul par éléments finis plus généraliste que les codes classiques. On peut affirmer qu'on l'a atteint puisque le résultat de ce travail est un code de calcul original, rapide et aisément extensible, capable à partir d'une même formulation, à l'écriture compacte, de traiter un cas en une dimension quelconque. Ce pour des cas simples, comme le Laplacien, des cas complexes, comme la formulation de Lesaint-Raviart, ou encore des cas usuels, mais pouvant nécessiter de longues périodes de développement, comme Navier-Stokes. On ouvre ainsi la voie à un développement plus rapide et plus facile de formulations diverses, ce qui ne peut que bénéficier à l'équipe de recherche de M. Trochu et, nous l'espérons, à tous ceux qui s'intéresseront à FEMView. Afin d'assurer le maximum de souplesse dans la diffusion de FEMView tout en préservant les droits intellectuels de ses auteurs, le moteur de calcul sera bientôt disponible gratuitement sous une licence GNU. Ainsi, toute personne désirant écrire des formulations spécifiques pourra utiliser le canevas développé dans le cadre de cette maîtrise et contribuer à l'enrichissement de FEMView tout en conservant complètement la propriété intellectuelle des formulations qu'elle a elle-même developpées.

Nous avons montré comment l'utilisation de certains outils mathématiques, notamment la géométrie différentielle, et d'une méthodologie de conception encore inusitée en ce domaine (l'approche multi-paradigmes) a permis d'atteindre notre but. L'approche multi-paradigmes, en exprimant les points communs et les variations entre les entités apparaissant dans un logiciel d'élément fini permet non seulement de justifier l'utilisation du paradigme générique, mais également de le laisser pleinement s'exprimer en ne le bridant pas par l'utilisation d'une méthode de conception inadaptée. Ce paradigme a un potentiel d'utilisation énorme. Qu'a-t-on démontré en effet dans ce

mémoire? Que l'approche objet n'est pas adaptée à la conception de logiciels purement algorithmiques. L'intérêt de ce résultat dépasse de loin la seule méthode des éléments finis. Aussi, une première direction de poursuite des travaux serait l'identification d'autres types de relations entres familles. On pourrait ainsi imaginer, au terme d'un travail important, de proposer une méthode de conception où l'on décrirait les constituants d'une application par une approche ensembliste complétée par la description des relations entre les ensembles. Une fois tout cela identifié, une correspondance optimale et systématique entre un ensemble et un paradigme d'implémentation pourrait être formulée.

Une seconde direction dans laquelle pourrait se diriger une suite à ce travail est le développement d'un langage de script. Le code en effet, permet très facilement la sélection de composants à utiliser lors de l'exécution, comme une fonction de forme par exemple, grâce à l'utilisation du mécanisme de liaison différée. Une fois l'implémentation de ce langage terminée, le développement d'une interface graphique pour FEMView ne consisterait plus qu'en l'écriture d'une application capable de représenter graphiquement les équations aux dérivées partielles, de définir les espaces fonctionnels et autres éléments à prendre en considération, puis de les traduire dans le langage de script. On justifierait ainsi le « View » de FEMView.

Enfin, on dispose dès à présent d'un outil facilement utilisable et pouvant servir de support à un cours d'éléments finis beaucoup plus appliqué et pratique que ne peuvent l'être les cours actuellement dispensés. On peut imaginer que les élèves ayant suivi un cours théorique en éléments finis et possédant une connaissance minimale en programmation reçoivent un enseignement condensé, adapté, sur les formes différentielles et certains principes de programmation. Ils pourront ensuite, après avoir étudié quelques formulations classiques, comme celles présentées dans ce mémoire, développer rapidement des codes basés sur FEMView satisfaisant leurs propres besoins

(On trouvera en annexe des énoncés d'exercices pouvant être utilisés lors de la définition de séances de travaux pratiques d'un tel cours).

RÉFÉRENCES

[1] James O. Coplien, « Multi-Paradigm Design for C++ », Addison Wesley

[2] Bernard Schutz, « Geometrical methods of mathematical physics », Cambridge University Press

[3] J.-C. Nedelec , « Notions sur les techniques d'éléments finis », Éditions Ellipses

[4] De Boor, « Mathematical Aspects of Finit Elements in Partial Differential Problems » (pages 89-119), Academic Press

[5] André Fortin, « Analyse Numérique pour Ingénieurs », Éditions de l'École Polytechnique de Montréal

[6] R. Temam, « Navier-Stokes Equations », North-Holland, Amsterdam

[7] « Problèmes aux limites non homogènes et applications », Dunod Paris (1968)

Annexe - Sujets de Travaux Pratiques

TP1 d'éléments finis appliqués

But

Prendre en main le moteur de calcul par éléments finis FEMView en analysant un exemple simple : le calcul d'un Laplacien.

Fourni

Le code source de FEMView et un projet Visual C++ le compilant

Description

Lire l'exemple complet « Laplacien » dans le mémoire de maîtrise de Ludovic Pénet

Travail préliminaire

Remplacer le fichier *main.cpp* par le fichier *exemple_laplacien.cpp* situé dans le sous-répertoire *samples\laplacien*.

Recompiler le projet.

Paramétrer Visual C++5 pour que les bons fichiers de configuration soient lus : dans le menu « Project », sélectionner « Settings ». Cliquer sur l'onglet « debug ». Dans la boîte « Program Arguments », spécifier le chemin des fichiers de configuration du Laplacien. Si par exemple les fichiers sont dans :

E:\Utilisateurs\Ludovic\FEMView\src\samples\laplacien

Taper :

E:\Utilisateurs\Ludovic\FEMView\src\samples\laplacien\laplacien

Ils sont au nombre de 6 :

- le fichier *.geo* est le fichier de définition de géométrie pour le mailleur GMSH
- le fichier *.cs* définit les caractéristiques. Dans le cas présent, une seule caractéristique constante est définie et est nommée *mu*

```
characteristic ConstantCharacteristic mu = {1.};
```

- le fichier *.fs* définit les espaces fonctionnels

```
function space fsLaplacien(GeomShapeFunction(1), true,
GeomShapeFunction(1), true,GeomShapeFunction(1));
function space fsInterpoPressure(GeomShapeFunction(1),
false,GeomShapeFunction(1),
false,GeomShapeFunction(1));
```

On définit ici deux espaces fonctionnels. Le premier sera utilisé pour assembler l'unique terme du Laplacien, le second sera utilisé pour interpoler la pression en utilisant les valeurs des degrés de liberté.

Les paramètres de l'espace fonctionnels sont les suivants : la fonction de forme (qui prend elle-même en paramètre l'ordre d'interpolation à utiliser) (1), si l'on doit directement évaluer la fonction de forme ou si l'on doit évaluer sa dérivée extérieure (2), la fonction de test (3), si l'on doit ou non prendre sa dérivée (4) et enfin la fonction de forme géométrique (5).

On remarque tout d'abord que l'on a bien spécifié un espace fonctionnel où l'on évalue la dérivée extérieure de la fonction de forme et la dérivée extérieure de la fonction test. On remarque ensuite que l'on ne prend pas la dérivée de la fonction de forme lorsque l'on interpole la quantité selon les résultats de la simulation. C'est en effet cette quantité et non sa dérivée que l'on veut évaluer.

- le fichier *.pq* définit les quantités physiques. Dans ce cas, une seule quantité physique est définie : la pression.

```
physical quantity
pressure(300,fsInterpoPressure,dof_laplacien,false);
```

Le paramètre 300 est la partie physique de la clé d'interaction avec le gestionnaire de degrés de liberté. `fsInterpoPressure` est l'espace fonctionnel que nous avons précédemment définit pour interpoler la pression.

`dof_laplacien` est le gestionnaire de degré de liberté où est (à priori) assemblée la pression

Enfin, `false` indique que la quantité physique est connectée. C'est à dire que, par exemple, la pression en un nœud est la même pour tous les éléments comportants ce nœud. En mettant ce paramètre à true, on définit un degré de liberté en ce nœud pour chacun des éléments comportant ce nœud.

- le fichier *.bc* définit les conditions aux limites.

```
boundary condition
bc1(11,0,pressure,fsInterpoPressure,ConstantScalarBound
aryCharacteristic,100);
boundary condition
bc2(12,0,pressure,fsInterpoPressure,ConstantScalarBound
aryCharacteristic,0);
```

On en définit ici deux : une sur le bord gauche et une sur le bord droit. La première valeur spécifiée en paramètre indique l'identificateur géométrique du support de la condition. On pourra remarquer que le fichier .geo définit deux physical lines d'identifiants 11 et 12. Le second paramètre indique l'ordre de la dérivée de la quantité que l'on impose. Ici la valeur 0 indique que l'on impose directement la quantité (condition de Dirichlet). Une valeur de 1 serait revenue à imposer la dérivée de la quantité. Le second paramètre est l'espace fonctionnel à utiliser lors de l'imposition de cette condition. Le troisième est le type de la condition aux limites à imposer. Les paramètres suivants dépendent du type de la condition aux limites. Dans le cas d'une constante, comme ici, on n'en précise qu'un : la valeur de la constante.

- Le fichier .unv contient le maillage. Il est généré par GMSH d'après le .geo.

À l'aide de l'éditeur de Visual C++ 5, repérer la ligne suivante :

```
genparamsCurrent.dTime = 0.;
```

Le code situé dans la fonction principale avant cette ligne effectue le parsing des paramètres passés en ligne de commande et charge les fichiers de configuration.

Les quelques lignes suivantes situées immédiatement après le point d'arrêt effectuent la récupération d'objets dans les registres globaux et la construction de vecteurs de paramètres pour la formulation Laplacien :

```
vector<PhysicalQuantity*> veclppq;
veclppq.push_back(&GlobalPhysicalQuantitiesNamesRegistr
y["pressure"]);
```

Instanciation d'un vecteur de quantités physiques. On y insère un unique élément : la quantité physique représentant la pression.

```
vector<FunctionSpaceData_c*> veclpfsdCurrent;
veclpfsdCurrent.push_back(GlobalFunctionSpacesNamesRegi
stry["fsLaplacien"]);
```

Instanciation d'un vecteur d'espaces fonctionnels. On y insère un unique élément : l'espace fonctionnel utilisé pour assembler l'unique terme du laplacien.

```
vector<Characteristic*> veclpcharac;
veclpcharac.push_back(GlobalCharacteristicsNamesRegistr
y["mu"]);
```

Instanciation d'un vecteur de caractéristiques. On y insère un unique élément : la caractéristique constante mu.

On passe ensuite tous ces vecteurs au constructeur de la class de formulation Laplacien_c, en plus du domaine support de la formulation :

```
Laplacien_c
laplacien(veclpfsdCurrent,maillage.GetAllElements(),vec
lpcharac,veclppq);
```

À ce point, on a défini une instance de la formulation Laplacien. La résolution de cette formulation est effectuée par l'unique appel suivant :

```
laplacien.TreatmentOfFormulation(maillage,GlobalBCNames
Registry,mrMateriaux,genparamsCurrent);
```

Travail à réaliser

Le travail à réaliser consiste à analyser le déroulement de la résolution de cette formulation.

Pour ce faire, on place un point d'arrêt sur la ligne de code appelant la méthode TreatmentOfFormulation de la classe Laplacien_c. On utilise ensuite le débogueur pour tracer l'exécution du programme.

On demande de :

1. repérer les lignes de code provoquant l'imposition des conditions aux limites de Dirichlet et de Robin

2. repérer la ligne de code définissant l'intégrateur

3. repérer la ligne de code provoquant l'assemblage du terme

4. décrire l'ordre d'utilisation du terme, de l'intégrateur, de l'assembleur de matrice élémentaire et des fonctions de forme. Qui utilise qui ?

5. décrire comment est calculée une matrice élémentaire sur un élément

L'assemblage de la matrice élémentaire est ensuite effectué par le terme par les quelques lignes de codes suivantes :

```
iCurrentCell = 0;
for(j=0;j<vecdnConnToAddToDofManager.size();j++){
```

```
        for(k=0;k<vecdnEqToAddToDofManager.size();k++){
         lpDofData-
>AssembleTerm(vecdnEqToAddToDofManager[k],
                              veciEqToAddToDofManager[k],

veciDisconnectorsEqToAddToDofManager[k],
                              vecdnConnToAddToDofManager[j],
                              veciConnToAddToDofManager[j],

veciDisconnectorsConnToAddToDofManager[j],

integresRetValue[iCurrentCell++]);
        }
}
```

Les clés d'interaction ayant préalablement été générées lors des appels aux fonctions SetNat de la base de fonction de forme et de la base de fonctions test :

```
geoelemCurrent.SetNat(fsdCurrent.GetShapeBase(),
        fsdCurrent.IsShapedF(),
        lppqShape->GetCode(),
        lppqShape->Disconnected(),
        &vecdnConnToAddToDofManager,
        &veciConnToAddToDofManager,
        &veciDisconnectorsConnToAddToDofManager);

geoelemCurrent.SetNat(fsdCurrent.GetTestBase(),
        fsdCurrent.IsTestdF(),
        lppqTest->GetCode(),
```

```
lppqTest->Disconnected(),
&vecdnEqToAddToDofManager,
&veciEqToAddToDofManager,
&veciDisconnectorsEqToAddToDofManager);
```

6. repérer la ligne de code appelant le résoluteur afin qui résolve la matrice globale assemblée par le terme

7. repérer la ligne de code provoquant la sauvegarde des résultats au format .pos (lisible par GMSH).

TP2 d'éléments finis appliqués

But

Étudier le fonctionnement de la formulation Navier-Stokes, des adapteurs et des contracteurs. Utiliser un schéma de résolution non-linéaire.

Fourni

Le code source de FEMView et un projet Visual C++ le compilant

Description

Lire l'exemple complet « Convection naturelle dans une cavité » dans le mémoire de maîtrise de Ludovic Pénet

Travail préliminaire

Avoir terminé et bien compris le TP1.

Remplacer le fichier *main.cpp* par le ficher *exemple_navier_stokes.cpp* situé dans le sous-répertoire samples\navier-stokes.

Recompiler le projet.

Paramétrer Visual C++5 pour que les bons fichiers de configuration soient lus : dans le menu « Project », sélectionner « Settings ». Cliquer sur l'onglet « debug ». Dans la boîte « Program Arguments », spécifier le chemin des fichiers de configurations de Navier-Stokes. Si par exemple les fichiers sont dans :

E:\Utilisateurs\Ludovic\FEMView\src\samples\navier-stokes\

Taper :

E:\Utilisateurs\Ludovic\FEMView\src\samples\navier-stokes\navier-stokes

Les fichiers de configuration sont similaires à ceux du laplacien. Comme on utilise cependant ici une 3*0-forme (la vitesse), il est nécessaire d'avoir à notre disposition une syntaxe permettant, lors de la définition des conditions aux limites, de spécifier quelle composante de la vitesse doit être imposée. Elle consiste à écrire, juste après la spécification de l'espace fonctionnel à utiliser lors de l'imposition de cette condition, l'indice de la composante concernée :

```
boundary condition bcNS2b1C1(11, 0, speed,
    fsInterpoSpeed, 1,
    ConstantScalarBoundaryCharacteristic, 0);
```

Le numéro de la composante est compris entre 0 et D-1, D étant la dimension de l'espace du problème.

Un autre fichier comporte une nouvelle syntaxe : le fichier de définition des espaces fonctionnels :

```
function space fsNS1(GeomShapeFunction(1), false,
    GeomShapeFunction(2, true, trace_adapter), true,
    GeomShapeFunction(1));
function space fsNS2(GeomShapeFunction(2, true), true,
    GeomShapeFunction(1));
...
```

On remarque que l'on spécifie plusieurs paramètres pour certaines fonctions de formes. On sait que le premier paramètre est l'ordre de la base d'interpolation. Le second indique si la fonction de forme doit interpoler une forme multiple ou non (sa valeur par défaut est *« false »*). Le troisième est un adapteur, en l'occurrence ici l'adapteur de trace (la lecture de la partie « Navier-Stokes » de l'exemple complet « Convection naturelle dans une cavité » explicite cet adapteur).

La formulation Navier-Stokes requiert l'utilisation d'un schéma de résolution non-linéaire. Ceci est fait dans la formulation Navier-Stokes de FEMView par l'insertion des lignes de code suivantes :

```
do{
    resid = 1.0;
```

```
            it = 0;
            if(it2++ == 0)Convergence_Factor = 1.0;
            do{
            //   Put actual solution on precedent one
            //   and zero the matrix
                 ((DofManager_c*&)*veclppqCurrent[1])-
>NonLinearStep();
et

                 ((DofManager_c*&)*veclppqCurrent[1])-
>NonLinearSolveMatrix();
                 // Calculate residual
                 residex = resid;
                 resid =
((DofManager_c*&)*veclppqCurrent[1])->NonLinearResidual();
                 printf("Non Linear Residual ->
%12.5E\n",resid);
                 //if(residex<resid)break;
                 it ++;
            }while( (resid > 5.e-4) && (it < 50));
            Convergence_Factor = sqrt(Convergence_Factor);
            printf("cf = %12.5E\n",Convergence_Factor);
      }while(Convergence_Factor > 1.001);
 ((DofManager_c*&)*veclppqCurrent[1])-
>NonLinearEndResolution();
```

On remarque que l'appel à NonLinearSolveMatrix remplace l'appel à SolveMatrix.

Travail à réaliser

Le travail à réaliser consiste à analyser le déroulement de la résolution de cette formulation.

Pour ce faire, on place un point d'arrêt sur la ligne de code appelant la méthode TreatmentOfFormulation de l'instance de la classe NavierStokes_c. On utilise ensuite le débogueur pour tracer l'exécution du programme.

On demande de :

1. Décrire le fonctionnement de l'adapteur trace. Quand est-il appelé ? Par quelle fonction ? Par quelle classe est appelée cette fonction ? (Indication : mettre un point d'arrêt dans l'opérateur de fonction de la classe trace_adapter sera très utile)

2. Décrire le fonctionnement du contracteur QuantityLastStep. Quand est-il appelé ? Par quelle fonction ? Par quelle classe est appelée cette fonction ? (Indication : mettre un point d'arrêt dans l'opérateur de fonction de la classe QuantityLastStep sera très utile). Il est obligatoire d'observer le fonctionnement du contracteur à la première itération ainsi qu'à une autre itération pour bien comprendre son fonctionnement et en déduire une caractéristique du gestionnaire de degrés de liberté.

3. Ouvrir le fichier de définition des quantités physiques. On remarque que la pression et la vitesse utilisent le même gestionnaire de degrés de liberté. Pourquoi ?

4. Expliquer pourquoi on peut placer plusieurs quantités physiques dans le même gestionnaire de degrés de liberté sans que cela pose problème.