MASTER EN SCIENCES INFORMATIQUES

Conception d'un système de génération automatique de données pour les programmes d'éléments finis
Étude de l'influence du mécanisme de mémoire virtuelle sur les performances de programmes

Bohon, Alain; Debois, Philippe

Award date:
1988

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la paix de Namur
Institut d'Informatique
Année académique 1987 - 1988

Conception d'un système de génération automatique de données pour les programmes d'éléments finis

Etude de l'influence du mécanisme de mémoire virtuelle sur les performances de programmes

Alain BOHON - Philippe DEBOIS

Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique

Promoteur : B. LECHARLIER

Abstract

Pour étudier le comportement de structures diverses soumises à certaines sollicitations, les ingénieurs de l'université de Liège utilisent la méthode des éléments finis. Celle-ci consiste à découper la structure qu'ils désirent étudier en éléments finis connectés entre eux en certains points appelés noeuds. L'ensemble des éléments finis et des noeuds est appelé maillage. Lorsque la structure atteint une certaine complexité, le maillage de celle-ci constitue un travail très fastidieux, c'est pourquoi on est amené à développer le maillage de façon plus automatique. La première partie de ce mémoire explique de façon détaillée le système de maillage automatique que nous avons conçu et implémenté. Les programmes d'éléments finis mettant en oeuvre la méthode ont été conçus à une époque où les machines à mémoire virtuelle n'existaient pas encore. En tenant compte du mécanisme de gestion de mémoire virtuelle, on peut s'attendre à améliorer les performances de ces programmes. C'est à cette étude comparative qu'est consacrée la deuxième partie de ce mémoire.

To study the behaviour of some structures submitted to some sollicitations, engineers of the university of Liège use the finite elements method. It consists in splitting the structure they want to study in finite elements connected with each other in some points called nodes. The set of finite elements and nodes is called mailing. When the stucture gains in complexity, the mailing of that structure becomes a very fastidious work, so that we are induced to develop this mailing in a more automatic manner. The first part of this dissertation explains in detail the automatic mailing system we have designed and implemented. Finite elements programs which work up the finite elements method have been designed in a period where virtual memory computers didn't exist yet. By taking into account the virtual memory mechanism, we can expect to improve performances of such programs. The second part of this dissertation is entirely devoted to that comparative study.

Remerciements

Avant toutes choses, nous tenons à remercier vivement Monsieur Lecharlier, promoteur de ce mémoire, pour l'aide et les conseils apportés lors de l'élaboration de ce travail.

Nous exprimons également notre reconnaissance envers Monsieur de Marneffe, professeur à l'Université de Liège, qui a proposé les sujets de ce mémoire. Nous le remercions également de nous avoir enseigné les cours relatifs aux techniques de compilation ainsi qu'au mécanisme de mémoire virtuelle.

Pour terminer, nous remercions Monsieur de Saxcé, ingénieur civil, avec qui nous avons eu des rapports très amicaux, et qui a pour une large part, contribué à la réalisation de ce projet. Nous remercions également toutes les personnes de la faculté de mécanique de l'Université de Liège pour l'accueil et l'aide qu'ils nous ont apportés.

Préface

Ce mémoire s'inscrit dans le cadre d'un stage effectué à la faculté de mécanique de l'Université de Liège, où nous avons travaillé sur deux projets distincts. Ce travail se compose donc de deux parties distinctes l'une de l'autre.

La première est consacrée à la conception d'un système de génération automatique de données pour les programmes d'éléments finis. La seconde étudie l'influence du mécanisme de mémoire virtuelle sur les temps d'exécution des programmes d'éléments finis.

Table des matières

Partie 1

Conception d'un système de génération automatique de données pour les programmes d'éléments finis

Chapitre 1
Exposé du problème

1.0 Introduction

L'évolution actuelle des technologies amène l'ingénieur à réaliser des projets de plus en plus complexes, coûteux, et soumis à des contraintes de sécurité de plus en plus sévères. Nous pensons bien sûr aux projets spatiaux, aéronautiques et nucléaires dans lesquels la sécurité est vitale. D'autres types de projets d'envergure sont liés à notre environnement : contrôle de la pollution thermique, acoustique ou chimique, aménagement des cours d'eau, gestion des nappes souterraines, prévision météorologique. Pour dominer ces projets, l'ingénieur a besoin de modèles qui lui permettent de simuler le comportement de systèmes physiques complexes. Il peut ainsi prévoir l'influence de ses décisions au moment de la conception du système.

Les sciences de l'ingénieur (mécanique des solides et des fluides, thermique, ...) permettent de décrire le comportement de systèmes physiques grâce à des équations aux dérivées partielles. La méthode des éléments finis est l'une des méthodes les plus utilisées aujourd'hui pour résoudre effectivement ces équations. Elle nécessite l'utilisation intensive de l'ordinateur pour mettre en oeuvre les méthodes numériques utilisées pour construire et résoudre ces équations. C'est une méthode très générale qui s'applique à la majorité des problèmes rencontrés dans la pratique par l'ingénieur.

Ce chapitre est entièrement consacré à l'exposé du problème que nous avons été amené à résoudre afin d'aider l'ingénieur dans l'étude du comportement de certains systèmes physiques. Une présentation préalable de la méthode des éléments finis utilisée par l'ingénieur dans son étude sera cependant nécessaire à la compréhension du problème qui nous préoccupe.

1.1 La méthode des éléments finis

Parmi les différentes tâches auxquelles s'attachent les ingénieurs de l'université de Liège, il en est une qui consiste à étudier le comportement de certains milieux continus, solides ou fluides, représentables par un système aux dérivées partielles.

Comme exemples, on peut citer :

- l'étude de la déformation d'un pont soumis à une certaine charge;
- l'étude de la déformation d'une aile d'avion ou d'une partie de fusée soumise à une série de sollicitations.

Essentiellement, les problèmes étudiés sont de deux types :

- étant donné une structure et un certain nombre de forces appliquées à différents endroits de sa frontière, on voudrait calculer le déplacement de l'ensemble des points de la structure résultant de l'application de ces forces;
- étant donné une structure, on impose certains déplacements en certains points de la structure et on calcule le déplacement des autres points.

Pour résoudre de tels problèmes, les ingénieurs font appel à la méthode des éléments finis. Notre connaissance de cette méthode étant assez limitée, nous n'en ferons qu'une description brève et assez générale.

Le principe de la méthode des éléments finis consiste à découper le milieu que l'on désire étudier en éléments, appelés éléments finis, connectés entre eux en certains points, appelés noeuds, auxquels sont associées les variables discrètes du problème (par exemple, les déplacements inconnus). L'ensemble des éléments finis et des noeuds est appelé maillage.

Les éléments servant à la discrétisation sont choisis parmi une bibliothèque d'éléments existants (segments, triangles, quadrangles, parallélipipèdes, etc ...). La façon

dont la structure est découpée en éléments du type de ceux choisis c'est-à-dire la façon dont le maillage est effectué nécessite l'expérience de l'ingénieur ainsi que certaines connaissances spécifiques.

Pour des structures bien définies qui reviennent souvent, des programmes de maillage automatique peuvent être réalisés, mais l'intervention de l'homme est toujours nécessaire pour décider du niveau de raffinement à utiliser.

Une fois le maillage de la structure effectué, chaque élément est étudié séparément. En étudiant le comportement de chacun des éléments, on espère plus tard pouvoir dériver le comportement global de la structure toute entière. Pour chaque élément impliqué dans la discrétisation, des équations d'équilibre sont dérivées. Ces équations sont ensuite assemblées pour donner lieu à un système d'équations linéaires associé à la structure toute entière. La résolution de ce système d'équations permet de trouver les inconnues du problème.

Etant donné la taille des systèmes d'équations mis en oeuvre, on ne peut espérer utiliser cette méthode sans faire usage de l'ordinateur. C'est pourquoi les ingénieurs utilisent des logiciels de calcul par éléments finis permettant de calculer les inconnues à partir d'une description du maillage de la structure et des sollicitations (forces ou déplacements) imposées.

1.2 Le problème

Les logiciels de calcul par éléments finis apportent une aide précieuse pour l'étude entreprise par l'ingénieur. Toutefois, ils ne couvrent pas toutes les étapes de la méthode des éléments finis. En effet, c'est à l'ingénieur qu'incombe de faire manuellement le maillage de la structure. Ce maillage manuel s'avère souvent être un travail considérable et fastidieux et peut constituer une limitation à la

complexité et à la taille des structures qui peuvent être étudiées.

Pour remédier à cela, on est amené à automatiser le travail de maillage. Le problème consiste donc à réaliser un système permettant de générer automatiquement le maillage de la structure en donnant à l'utilisateur la possibilité d'influencer la façon dont la génération est accomplie. Selon les responsables de ce projet, l'utilisation d'un langage constitue un moyen élégant pour résoudre ce genre de problème.

Grâce à l'utilisation d'un tel système, on espère :
- supprimer l'aspect fastidieux du travail de maillage;
- alléger le travail de l'ingénieur;
- pouvoir étudier des structures de plus grande complexité.

Avant d'aller plus loin dans l'exposé du problème, il est important de faire la remarque suivante : la méthode utilisée par les ingénieurs de l'université de Liège pour étudier le comportement de structures est en fait une variante de la méthode des éléments finis, à savoir, la méthode des éléments frontières.

Cette méthode diffère essentiellement de la méthode des éléments finis par les caractéristiques suivantes :

- seule la frontière de la structure est découpée en éléments et non la totalité de la structure. Les éléments ne se trouvent donc que sur la frontière de la structure, d'où le nom de la méthode;
- la méthode des éléments frontières fait usage d'équations intégrales (équations dont les inconnues se trouvent sous une intégrale) au lieu d'équations aux dérivées partielles dans le cas de la méthode des éléments finis;
- les systèmes d'équations linéaires à résoudre dans la méthode des éléments frontières sont non-symétriques alors qu'ils sont symétriques dans le cas de la méthode des éléments finis.

Etant donné ces considérations, seul le maillage de la frontière de la structure doit être pris en charge par le système à mettre en oeuvre.

Le logiciel de calcul par éléments frontières BEM (Boudary Element Method) utilisé par les ingénieurs de l'université de Liège pour calculer les inconnues ne peut en fait traiter que des problèmes plans. C'est pourquoi notre système se limitera au maillage de la frontière de structures planes. On peut considérer que le fait de se limiter à deux dimensions constitue une simplification non négligeable du problème, le cas à trois dimensions étant un problème beaucoup plus complexe.

1.3 Le langage

Le moyen retenu par les responsables de ce projet pour exprimer les données nécessaires au maillage automatique d'une structure est un langage. Nous passerons en revue dans cette les différentes choses que le langage devrait permettre d'exprimer afin que le maillage automatique de la frontière d'une structure puisse avoir lieu.

Ce langage devrait offrir la possibilité de décrire la géométrie d'une structure, les sollicitations (forces ou déplacements) qui sont imposées sur sa frontière ainsi que les caractéristiques propres à la pièce.

Outre cela, il devrait posséder, selon les utilisateurs, les caractéristiques suivantes :

- simplicité;
- facilité d'utilisation;
- concision des descriptions : un langage basé sur une représentation CSG serait plus souhaitable qu'un langage basé sur une représentation frontière (cfr chapitre 2).

La géométrie de la structure

Pour pouvoir générer les noeuds et les éléments sur la frontière d'une structure, une description de la géométrie de cette structure est nécessaire. On voudrait que celle-ci soit concise et simple à valider.

Le langage devrait idéalement offrir certains moyens pour faciliter cette description en tenant compte des répétitions, des symétries, etc ...

Afin de pouvoir générer des noeuds à l'intérieur de la structure (cfr chapitre 5), celle-ci (aussi appelée domaine) est partitionnée en sous-régions (appelées sous-domaines). C'est en fait la géométrie de chacun de ces sous-domaines qu'il faudra décrire et non la géométrie du domaine tout entier.

Afin de pouvoir vérifier que la structure décrite est correcte, des ordres d'affichage doivent également être prévus dans le langage.

Les sollicitations

Essentiellement, les problèmes qu'étudie l'ingénieur sont de deux types. Etant donné une structure :

- on impose des forces à certains noeuds et on veut obtenir, pour chacun des noeuds de la structure, le déplacement résultant de l'application de ces forces;
- on impose des déplacements à certains noeuds et on veut obtenir les déplacements résultant aux autres noeuds;
- on peut combiner les deux premiers types de problèmes en imposant à la fois des forces à certains noeuds et des déplacements à d'autres et voir comment se comporte l'ensemble des noeuds de la structure.

Le langage devrait permettre de décrire de façon simple les sollicitations c'est-à-dire les forces et/ou les déplacements imposés à la frontière de la structure. Ces

sollicitations devraient pouvoir être spécifiées en tenant compte des répétitions et des symétries de la structure ...

Les sollicitations peuvent être de deux types :

- ponctuelles c'est-à-dire localisées en un endroit bien précis de la frontière de la structure (par exemple en un noeud);
- réparties c'est-à-dire distribuées sur un segment de la frontière de la structure (par exemple sur un côté de la structure).

Dans le cas réparti, seules les sollicitations aux extrémités du segment devraient être spécifiées, les sollicitations à l'intérieur étant générées automatiquement par interpolation.

Autres caractéristiques

Le langage devrait permettre de définir certaines caractéristiques du matériau de la structure nécessaires au programme BEM de calcul par éléments frontières pour calculer les inconnues. On peut citer par exemple le module d'élasticité permettant de savoir de quelle quantité s'étire la structure lorsqu'on lui applique une certaine force.

Il devrait également permettre d'exprimer des indications quant à la façon de faire la génération :

- niveau de raffinement à utiliser (nombre de noeuds et nombre d'éléments à générer);
- indications spécifiant s'il faut générer les noeuds et les éléments de façon uniforme sur l'entièreté de la frontière ou bien générer plus de noeuds et d'éléments à certains endroits de la frontière de la structure.

Ceci permet à l'ingénieur de garder un certain contrôle sur la manière dont est faite la génération.

1.4 Le système

Nous allons dans cette dernière section présenter la façon dont notre système de génération automatique a été décomposé en sous-systèmes. Il faut remarquer que la décomposition que nous avons choisie n'est pas la seule décomposition possible et nous n'avons nullement la prétention d'affirmer que nous avons fait le meilleur choix.

A partir d'un texte source écrit dans notre langage exprimant toutes les données nécessaires au maillage automatique de la frontière d'une structure, les opérations suivantes doivent être réalisées :

- vérifier que le texte source constitue bien une chaîne appartenant à notre langage;
- afficher la structure afin que l'utilisateur puisse vérifier qu'elle est correcte, c'est à dire qu'elle correspond bien à la structure qu'il veut étudier;
- générer de façon automatique le maillage de la frontière de la structure ainsi que les autres données nécessaires au programme BEM de calcul par éléments frontières;
- En outre, ces deux dernières étapes ne peuvent être réalisées que si une représentation frontière de la structure est disponible. Or, le langage étant basé sur une représentation CSG, un changement de représentation sera donc nécessaire (cfr chapitre 2).

Ces quatre opérations étant relativement différentes de par leur nature et assez indépendantes, nous avons décidé de baser la décomposition de notre système sur celles-ci.

Ainsi, on peut d'ores et déjà identifier les modules suivants :

- un compilateur traduisant un texte de description d'une structure en une représentation CSG de cette structure (plus d'autres informations nécessaires à la génération);
- un module de conversion permettant de passer de la représentation CSG de la structure à une représentation frontière.
- un module d'affichage de la structure;

- un module de génération effectuant le maillage proprement dit de la frontière de la structure.

Les chapitres suivants sont consacrés à l'exposé de chacun de ces sous-systèmes. Mais avant d'exposer ceux-ci, il est bon d'expliquer les différents schémas de représentation de solides existants et d'approfondir ceux que nous avons choisis. C'est l'objectif du chapitre 2.

Chapitre 2
Schémas de représentation des solides

2.0 Introduction

La question de savoir comment représenter la géométrie d'objets solides rigides est d'une importance capitale dans les systèmes nécessitant de telles modélisations. On peut rencontrer de tels systèmes de modélisation dans les domaines d'application suivants : conception de pièces assistée par ordinateur, fabrication de pièces assistée par ordinateur, architecture, ingénierie civile, simulation, graphismes sur ordinateur, vision des robots, etc ...

Dans ce chapitre nous verrons, dans un premier temps, comment appréhender la notion de solide de façon mathématique et donner une définition de ce qu'est un schéma de représentation de solides. Nous présenterons également les propriétés formelles et informelles qui permettent de caractériser un schéma de représentation.

Nous exposerons ensuite les différents schémas de représentation les plus couramment utilisés, ainsi que les propriétés formelles et informelles de chacun de ceux-ci.

Pour terminer, nous expliquerons les schémas de représentation qui ont été choisis dans notre système de génération et nous verrons en quoi ces schémas sont particulièrement bien adaptés aux besoins de celui-ci.

Les généralités présentées dans ce chapitre sont principalement issues de [Requicha 1980], [Meier 1986] et [Requicha et Voelcker 1977]. Le lecteur désireux d'approfondir le sujet peut se référer à ces articles.

Dans ce chapitre, nous emploierons parfois l'expression "schéma de représentation" en lieu et place de l'expression "schéma de représentation de solides".

2.1 Le modèle mathématique

Les solides physiques vont être modélisés par des entités géométriques abstraites qui sont des sous-ensembles de l'espace euclidien à trois dimensions E^3. Pour modéliser des objets physiques, ces entités géométriques abstraites devraient vérifier les propriétés évidentes décrites ci-dessous (ces propriétés seront formalisées ultérieurement).

1° la rigidité :
Tout solide abstrait qui modélise un solide physique doit posséder une forme qui est indépendante de la localisation et de l'orientation du solide.

2° l'homogénéité tri-dimensionnelle :
Un solide doit avoir un intérieur et la frontière du solide ne peut avoir de partie isolée ou pendante (ex : sommet isolé, arête ou face pendante).

3° occupation finie :
Un solide doit occuper une partie finie de l'espace. L'ensemble des points qui le constituent doit donc être borné.

4° fermeture sous certaines opérations :
L'application d'opérateurs de mouvement (translation, rotation) ou d'opérateurs ensemblistes (union, intersection, différence) à des solides doit fournir d'autres solides (respectant les mêmes propriétés).

5° déterminisme de la frontière :
La frontière d'un solide doit déterminer de façon non ambiguë le solide et l'intérieur de celui-ci.

De façon plus formelle, les modèles les plus appropriés pour les solides sont des sous-ensembles de E^3 qui sont bornés, fermés et réguliers. On appellera de tels ensembles des r-ensembles.

On peut expliquer cela intuitivement de la façon suivante : un ensemble borné est un ensemble qui occupe une partie finie de l'espace. Un ensemble fermé est un ensemble qui contient sa frontière. Un ensemble régulier est un ensemble dont la totalité de sa frontière est adjacente à l'intérieur de l'ensemble (aucune partie isolée ou pendante).

De façon plus formelle :
soit X un sous-ensemble de l'espace euclidien E^3.

- Un point p est un élément de l'intérieur de X, iX, ssi il existe un voisinage de p (par exemple, une boule ouverte de centre p et de rayon positif) qui est contenu dans X.
- Un point p est un élément de la fermeture de X, kX, ssi tout voisinage de p contient un point de X.
- Un point p est un élément de la frontière de X, bX, ssi p est à la fois un élément de kX et de k(cX) où cX dénote le complément de X.
- L'ensemble X est dit régulier ssi X = kiX c'est-à-dire ssi il est égal à la fermeture de son intérieur. La régularisation de l'ensemble X, notée rX, est définie par rX = kiX.

Rendre un ensemble régulier a donc pour effet de supprimer les parties isolées ou pendantes de cet ensemble. Il faut bien voir que tout ensemble régulier est fermé mais que l'inverse n'est pas nécessairement vrai. Ceci peut être illustré au moyen du contre-exemple de la figure 2.1. L'ensemble de la figure 2.1a est fermé car il contient sa frontière, mais il n'est pas régulier car sa frontière possède des parties qui ne sont pas adjacentes à son intérieur (en l'occurrence, la face pendante et le côté pendant). La régularisation de l'ensemble de la figure 2.1a produit l'ensemble (régulier et fermé) de la figure 2.1b.

Les r-ensembles doivent être fermés (au sens expliqué ci-après) sous des mouvements rigides (c'est-à-dire des mouvements qui conservent la forme des solides) et sous des opérations ensemblistes. En d'autres mots, si on fait subir un mouvement rigide à un r-ensemble ou si on applique une opération ensembliste à deux r-ensembles, on obtient encore un

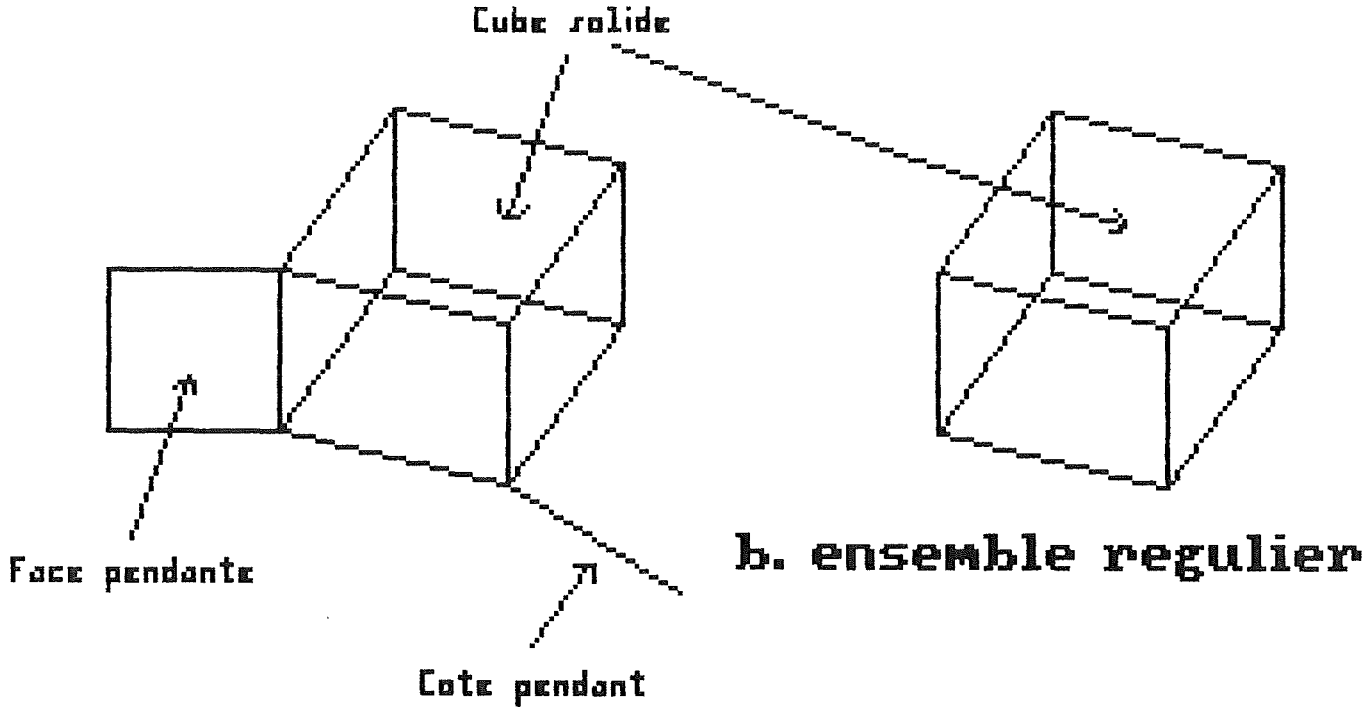


Fig 2.1 Regularisation d'un ensemble

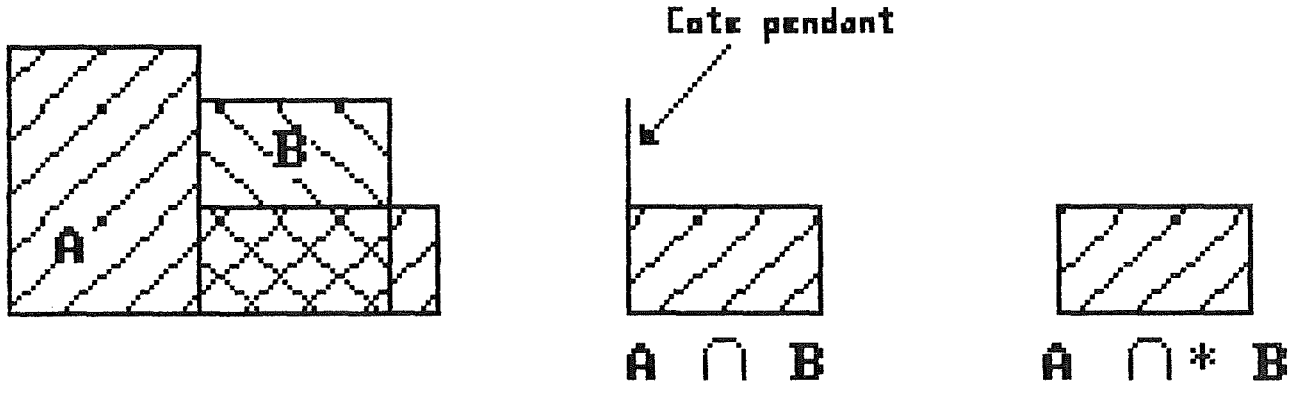


Fig 2.2 L'intersection regularisee de deux r-ensembles est un r-ensemble

r-ensemble. Ceci traduit le fait que, si on fait subir une translation ou une rotation à un solide, on obtient encore un solide. De même, si on applique des opérations ensemblistes à des solides (union, intersection, différence), on obtient toujours un solide. Il est à noter que l'ensemble vide est un r-ensemble et correspond au solide nul.

Les opérateurs ensemblistes traditionnels d'union, d'intersection, de différence et de complémentarité ne sont pas admissibles dans le sens où ils détruisent la propriété de régularité. Ceci peut être illustré par la figure 2.2 où l'intersection traditionnelle des solides A et B fournit un solide possédant un côté pendant (c'est-à-dire un ensemble non régularisé). Il faut donc utiliser des opérateurs ensemblistes régularisés définis par

$$A \cup^* B = r(A \cup B)$$
$$A \cap^* B = r(A \cap B)$$
$$A -^* B = r(A - B)$$
$$c^*A = rcA$$

plutôt que les opérateurs ensemblistes traditionnels. Dans la figure 2.2, l'utilisation de l'intersection régularisée élimine le côté pendant en rendant l'ensemble résultat régulier.

2.2 Définitions

Nous allons dans cette section définir de façon mathématique ce qu'on entend par schéma de représentation d'un solide.

L'ensemble des représentations syntaxiquement correctes est appelé l'espace de représentation et sera noté R. Cet ensemble de représentation peut être vu comme un langage formel c'est-à-dire un langage construit à partir d'un alphabet selon certaines règles de grammaire. Une représentation est syntaxiquement correcte si la structure de symboles qui lui correspond appartient au langage.

La sémantique des représentations est définie en associant ces représentations avec les entités géométriques abstraites définies dans la section 2.1. L'espace de modélisation est défini comme étant l'ensemble des solides abstraits (r-ensembles) et sera noté M. On établira une correspondance entre les éléments de R et les éléments de M au moyen d'un schéma de représentation (cfr fig 2.3).

Un schéma de représentation est donc une relation s de M vers R. Nous dénoterons le domaine de s par D et l'étendue ou l'image de D via s par V. Toute représentation dans V est dite valide puisqu'elle est syntaxiquement correcte et sémantiquement correcte (dans le sens où elle possède des éléments correspondants dans le domaine D).

Nous noterons que :

- il y a des solides qui ne sont pas représentables dans un schéma (ceux de M \ D). En d'autres termes, tout schéma de représentation possède une puissance de description limitée;
- il y a des représentations syntaxiquement correctes qui ne sont pas valides (celle de R \ V). En d'autres mots, il y a des représentations qui ne correspondent à aucun solide de l'espace de modélisation.

2.3 Propriétés formelles des schémas de représentation

Cette section présente quelques propriétés formelles des schémas de représentation et leurs implications.

2.3.1 Le domaine

Le domaine d'un schéma de représentation (D) caractérise la puissance descriptive de ce schéma. En d'autres mots, le domaine d'un schéma de représentation est l'ensemble des solides représentables dans ce schéma.

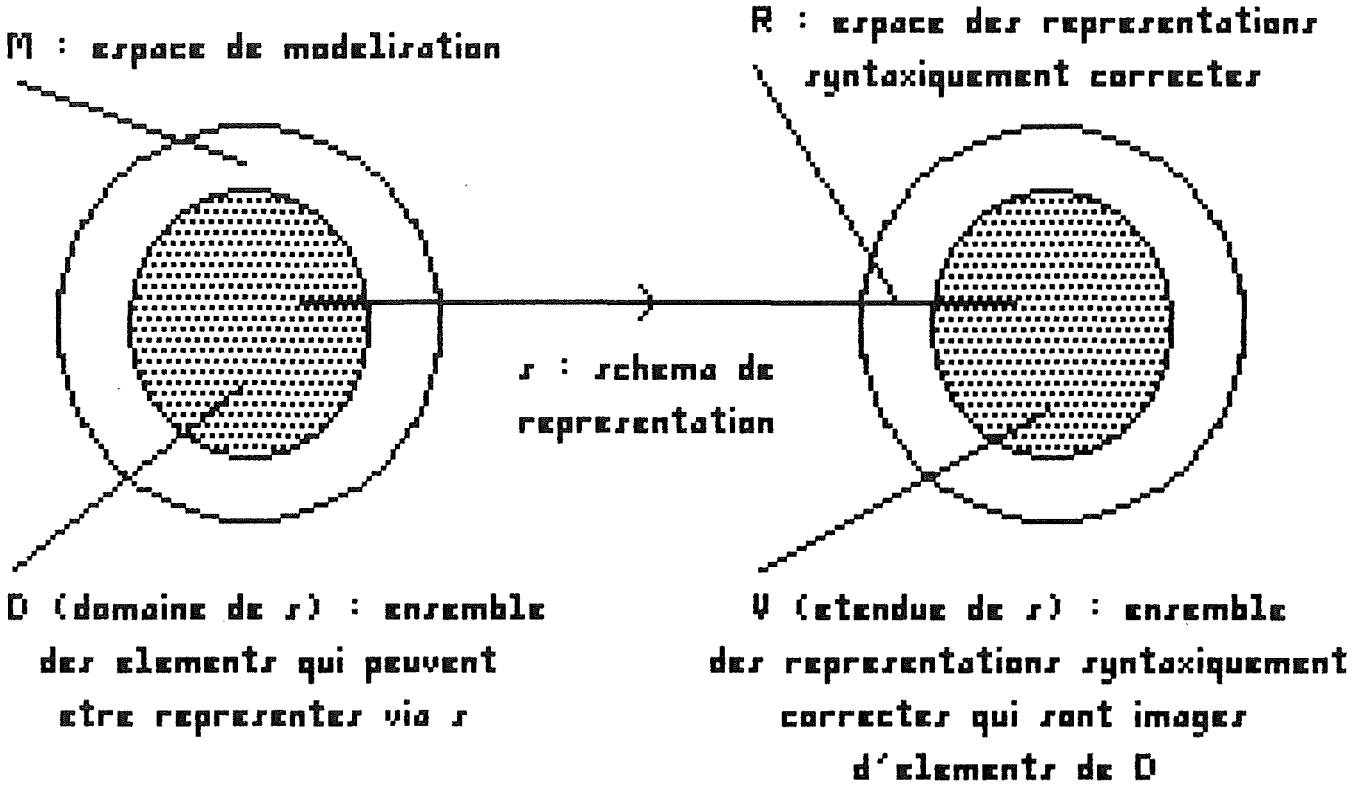


Fig 2.3 Schema de representation

Pour qu'un schéma de représentation soit adopté, il faut que le domaine de ce schéma soit suffisamment grand que pour pouvoir représenter les solides que l'on veut modéliser.

2.3.2 La validité

Pour pouvoir être utilisée, la représentation d'un solide doit être valide, c'est à dire que cette représentation doit correspondre à un solide de l'espace de modélisation. En d'autres mots, une représentation est valide si elle est un r-ensemble. Une représentation est invalide si elle ne correspond à aucun solide appartenant au domaine du schéma de représentation. L'étendue d'un schéma de représentation (V) est l'ensemble des représentations qui sont valides.

Il faut bien voir la différence qui existe entre une représentation valide et une représentation correcte. Une représentation peut très bien être valide c'est-à-dire correspondre à un solide de l'espace de modélisation mais ne pas être correcte dans le sens où elle ne correspond pas au solide que l'utilisateur voulait définir.

Il est évident que la validité d'une représentation est une propriété importante dans le sens où on ne peut faire aucune opération sur une représentation invalide. Invoquer un algorithme sur une représentation invalide peut provoquer la production de résultats visiblement suspects ou, dans le pire des cas, des résultats qui semblent corrects mais qui sont en fait erronés.

Par le passé, la responsabilité d'assurer la validité des représentations était assumée par l'utilisateur lui même. Typiquement, l'utilisateur créait la représentation d'un solide et ensuite vérifiait au moyen d'un affichage graphique que celle-ci était correcte. Il est facile de voir que cette méthode favorise l'apparition d'erreurs, la vérification visuelle étant parfois difficile, et est impossible à mettre en oeuvre lorsque les objets atteignent une relative complexité.

De plus en plus, on se tourne vers des systèmes qui prennent en charge automatiquement la vérification de la validité des représentations. Deux approches peuvent être utilisées. Soit la vérification de validité est effectuée par algorithme après que la représentation du solide ait été construite, soit cette vérification est faite lors de la construction de la représentation. Il faut noter qu'une vérification visuelle de la correction de la représentation est malgré tout nécessaire puisque rien n'assure qu'une représentation valide corresponde au solide que l'on veut représenter.

La façon la plus élégante de procéder est de concevoir un schéma de représentation dans lequel toutes les représentations jugées correctes d'après la syntaxe sont valides. Le problème de vérification de validité est alors réduit à un simple problème d'analyse. Cependant, il faut faire attention à ce qu'un schéma possédant cette propriété ne soit pas trop difficile à utiliser ou ne soit pas trop restrictif. Nous verrons par la suite que le schéma CSG possède cette propriété tout en restant facile à l'utilisation.

2.3.3 La complétude ou non ambiguïté

Une représentation de solide est complète ou non ambiguë ssi à cette représentation ne correspond qu'un et un seul solide (la représentation $r \in V$ est non ambiguë ssi $s^{-1}(r)$ est un singleton (m) de D). Un schéma de représentation est non ambigu ou complet ssi toutes ses représentations valides sont non ambiguës (c'est-à-dire si s^{-1} est une fonction).

Par définition, une représentation non ambiguë contient suffisamment d'informations pour pouvoir différencier un solide d'un autre et constitue une source d'informations suffisante pour les opérations à effectuer sur celle-ci. Dans notre système par exemple, une représentation frontière d'une

pièce est suffisante pour pouvoir générer les noeuds et les éléments.

Par la suite, nous nous restreindrons aux schémas de représentation non ambigus.

2.3.4 L'unicité

La représentation d'un objet est unique ssi l'objet correspondant ne possède pas d'autres représentations que celle-là (la représentation $r \in V$ est unique ssi $s(s^{-1}(r)) = \{r\}$). Un schéma de représentation est unique ssi toutes ses représentations valides sont uniques (c'est-à-dire ssi s est une fonction).

Cette propriété est surtout importante pour l'évaluation de l'égalité de solides. La nécessité de pouvoir évaluer l'égalité de deux solides peut être illustrée par l'exemple suivant : dans une base de données stockant des représentations de géométrie d'objets, les représentations redondantes ne peuvent être éliminées que s'il est possible de déterminer si deux représentations dans un même schéma correspondent au même objet.

Des schémas de représentation qui sont à la fois non ambigus et uniques sont hautement désirables puisque ils établissent une correspondance biunivoque entre les objets et leur représentation (entre D et V). Dans de tels schémas, des représentations distinctes correspondent à des objets distincts et inversément. C'est pourquoi l'égalité des solides peut être déterminée au moyen d'algorithmes qui travaillent sur leur représentation.

Le test d'égalité dans des schémas non ambigus mais non uniques nécessite la mise en oeuvre de techniques beaucoup plus élaborées.

Il faut remarquer que la plupart des schémas de représentation ne vérifient pas cette propriété d'unicité et ce pour au moins deux raisons :

- non unicité permutationnelle : les sous-structures d'une représentation peuvent être permutées;
- non unicité positionnelle : deux représentations distinctes peuvent correspondre à des copies conformes d'un même objet mais positionnées à des endroits différents.

2.4 Propriétés informelles des schémas de représentation

Cette section présente quelques propriétés des schémas de représentation qui sont d'une grande importance mais qui ne peuvent être formalisées.

2.4.1 La concision

La concision d'un schéma se mesure à la taille des représentations de ce schéma. Un schéma qui se veut concis devrait posséder des représentations de taille relativement réduite. Les représentations d'un tel schéma devraient contenir le moins d'informations redondantes possible.

Il faut remarquer que la concision ne constitue pas nécessairement la meilleure solution. En effet, une représentation non concise contenant des informations redondantes peut constituer un avantage pratique. On peut ainsi améliorer les performances des algorithmes qui travaillent sur ces représentations en stockant des informations dérivables qui, autrement, devraient être recalculées à chaque fois.

2.4.2 La facilité de création

La facilité avec laquelle des représentations valides peuvent être créées par un utilisateur est d'une importance capitale, particulièrement si l'utilisateur est humain.

Les représentations concises sont généralement plus faciles à créer que les représentations verbeuses, bien que ce ne soit pas toujours le cas.

Les systèmes basés sur des représentations verbeuses fournissent en général un sous-système d'aide à la création de représentations et nécessitent l'existence de mécanismes non triviaux permettant de vérifier la validité de ces représentations. Les représentations concises, elles, sont en général plus faciles à valider (validation essentiellement au niveau de la syntaxe).

2.4.3 L'efficacité dans le cadre d'une application

Il faut voir les représentations de solides comme étant, avant tout, des sources de données pour les algorithmes qui réalisent les fonctionnalités du système. Ainsi, dans notre système, la génération des noeuds et des éléments nécessite une représentation frontière du solide. On peut difficilement envisager cette génération à partir d'une autre représentation. Le calcul, par exemple, du volume ou du poids (à partir du poids volumique) d'un solide peut s'avérer être plus facile à réaliser à partir d'une représentation dans un schéma qu'à partir d'une représentation dans un autre schéma.

Une représentation non ambiguë constitue, comme on l'a déjà dit, une source de données suffisante pour ces algorithmes, mais elle ne contribue pas nécessairement à l'efficacité de ceux-ci. C'est ainsi qu'il est parfois utile d'ajouter des informations redondantes supplémentaires.

Dans le cadre d'une application particulière, la représentation devrait être choisie de façon à contribuer à ce

que les algorithmes qui réalisent les fonctionnalités du système soient corrects, efficaces, robustes (récupération des erreurs) et extensibles. Nous verrons au chapitre 4 comment la représentation CSG contribue à la correction et à l'extensibilité de notre système de génération.

L'expérience montre qu'il n'y a pas de schéma de représentation qui soit le meilleur dans tous les cas. On peut seulement dire que certains conviennent mieux que d'autres pour certains types d'applications. Quant aux systèmes à desseins généraux, ils contiennent généralement plusieurs représentations différentes des objets présents ainsi que des mécanismes permettant de passer d'une représentation à l'autre.

Il faut également remarquer que le langage permettant à l'utilisateur d'entrer un solide dans le système ne doit pas nécessairement être une image fidèle du schéma de représentation dont le système a besoin. Ce qui est pratique pour l'application n'est pas nécessairement pratique pour l'utilisateur. Ainsi, malgré la nécessité d'une représentation frontière pour la phase de génération, le langage d'entrée de notre système se base sur le schéma de représentation CSG (cfr ci-dessous).

2.5 Les schémas de représentation de solides

Nous développerons ici, les principaux schémas de représentation de solides les plus connus et les plus utilisés en pratique. Nous nous restreindrons aux schémas de représentation non ambigus, les seuls qui nous intéressent ici, les schémas de représentation ambigus étant inadéquats pour notre système. Des schémas de représentation analogues peuvent être conçus pour des entités qui ne sont pas des solides comme, par exemple, des surfaces. Des schémas hybrides peuvent également être construits par combinaison de différentes techniques.

Pour chacun des schémas développés, nous mettrons en évidence les propriétés qu'il possède afin de pouvoir, par la suite, orienter notre choix en fonction des critères qui nous intéressent.

2.5.1 Schéma d'instanciation pure de primitives

Ce schéma est basé sur la notion de familles d'objets, chaque objet d'une famille se distinguant des autres objets de cette même famille par quelques paramètres. Chaque famille d'objets est appelée primitive générique, et chaque objet individuel à l'intérieur d'une famille est appelé occurrence de la primitive.

Les occurrences d'une primitive sont représentées par des t-uples de valeurs.

Exemples :

- on pourrait avoir comme familles de solides : la famille des sphères, la famille des parallélépipèdes rectangles, la famille des cônes, etc ...
- les occurrences de la famille des sphères seraient représentées par des couples de la forme ('Sphère',R) où 'Sphère' est une chaîne de caractères représentant le nom de la famille et R est le rayon de la sphère;
- les occurrences de la famille des parallélépipèdes rectangles seraient représentées par des quadruplets de la forme ('ParaRect',L,P,H) où L est la longueur, P la profondeur et H la hauteur du parallélépipède rectangle;
- on pourrait également avoir des familles de surfaces, comme la famille des rectangles, la famille des cercles, la famille des trapèzes, etc ...

Propriétés de ce schéma

o Ce schéma est non ambigu et unique (correspondance biunivoque entre les solides et leur représentation), facile à valider (validation au niveau syntaxique), concis (quelques paramètres) et facile à utiliser.

- La grande lacune de ce schéma réside dans le fait que son domaine, c'est-à-dire l'ensemble des solides représentables, est extrêmement restreint. En effet, il n'y a, dans ce schéma, aucune façon de combiner les occurrences de primitives pour construire des structures qui représentent des objets plus complexes. Ce schéma n'a donc aucun intérêt en soi excepté le fait qu'il est utilisé comme sous-structure par d'autres schémas tel que le schéma CSG par exemple.

- Un autre inconvénient de ce schéma est que les algorithmes qui travaillent sur ces représentations doivent inclure une grande quantité de connaissances spécifiques à chaque famille. Chaque famille d'objets doit être traitée comme un cas particulier ne permettant aucune uniformisation au niveau des traitements.

2.5.2 Décomposition en cellules

La décomposition en cellules est une généralisation de la triangularisation. La triangularisation d'un polyèdre (rectiligne ou curviligne) consiste à décomposer le polyèdre en tétraèdres (rectilignes ou curvilignes) qui sont soit disjoints, soit connexes aux faces, côtés ou extrémités communes. Dans la décomposition en cellules, les tétraèdres sont remplacés par des cellules à trois dimensions qui possèdent un nombre arbitraire de côtés.

Un solide peut être représenté en le décomposant en cellules à trois dimensions et en représentant chacune de ces cellules.

Propriétés de ce schéma

- Ce schéma n'est pas unique.
- La validité est très coûteuse à vérifier, des algorithmes de vérifications de frontières en 3 dimensions devant être mis en oeuvre.

- Les décompositions en cellules ne sont généralement ni concises, ni faciles à créer.
- Ce schéma est particulièrement commode pour le calcul de certaines propriétés comme, par exemple, vérifier si le solide est composé d'un seul morceau, s'il possède des trous, calculer le volume ou le moment d'inertie.

2.5.3 Enumération d'occupation spatiale

Le schéma d'énumération d'occupation spatiale est un cas particulier du schéma de décomposition en cellules dans lequel toutes les cellules sont des cubes et sont situées dans une grille fixe.

Propriétés de ce schéma

- Ce schéma n'est pas unique sauf en ce qui concerne l'unicité positionnelle.
- Les représentations de ce schéma sont faciles à valider mais plutôt verbeuses.

2.5.4 Géométrie constructive du solide

L'expression 'géométrie constructive du solide', traduit de l'anglais 'constructive solid geometry' (CSG en abrégé), dénote une famille de schémas formels de représentation. Ces schémas de représentation permettent de concevoir des entités mathématiques qui sont adéquates pour modéliser certains types d'objets physiques.

Dans un tel schéma, un solide est représenté comme étant la construction ou combinaison de composants solides plus petits au moyen d'opérateurs ensemblistes régularisés.

La représentation CSG ainsi que la représentation par frontières (développée au point 2.5.6) étant les schémas de représentation actuellement les plus importants et ceux qui

ont fait l'objet de notre choix, nous les traiterons de façon un peu plus approfondie que les autres.

La représentation CSG

La représentation CSG fait usage d'arbres binaires pour représenter un objet. Il est possible de définir une représentation CSG indépendamment d'une implémentation particulière, mais l'usage d'arbres binaires permet d'avoir une vue plus imagée de ce qu'est une représentation CSG. Un arbre CSG peut être caractérisé de la façon suivante :

1° les noeuds non terminaux représentent des opérateurs qui peuvent être soit un mouvement rigide (translation, rotation), soit une union, une intersection ou une différence régularisée. Nous exclurons le complément régularisé qui peut engendrer des ensembles non bornés;

2° les noeuds terminaux ou feuilles sont soit des solides primitifs, soit des arguments d'un mouvement rigide.

La sémantique est la suivante : chaque noeud non terminal de l'arbre représente l'ensemble résultant de l'application de l'opérateur de mouvement ou de combinaison associé au noeud aux ensembles représentés par les sous-arbres (cfr l'exemple de la figure 2.4).

Les solides primitifs

Les entités de plus bas niveau sont les demi-espaces primitifs. Un demi-espace primitif peut être défini comme :

$$\{ p \in E^3 \mid f(p) \geq 0 \}$$

où $f(p)=0$ défini une surface appelée surface primitive. Un demi-espace primitif est un ensemble régulier mais n'est pas nécessairement borné.

Des demi-espaces plans et des demi-espaces cylindriques sont des exemples typiques de demi-espaces primitifs (non bornés).

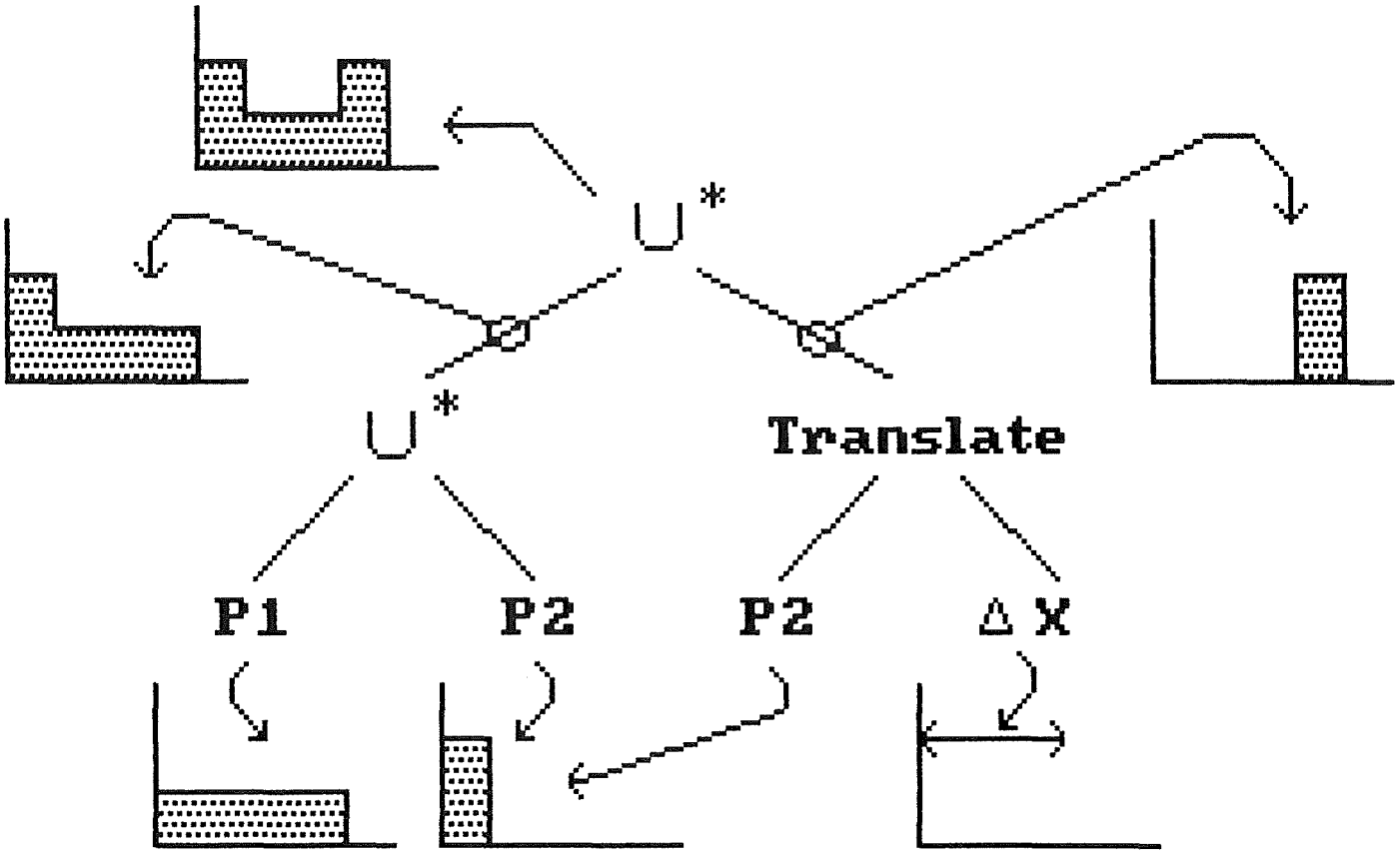


Fig 2.4 Arbre CSG d'un solide

Les demi-espaces primitifs peuvent être combinés au moyen d'opérateurs ensemblistes régularisés pour former des ensembles réguliers bornés (r-ensembles) qui pourront être vus comme des occurrences de solides primitifs.

Par exemple, un schéma CSG possédant seulement les demi-espaces primitifs plans et cylindriques offrirait typiquement deux solides primitifs : un bloc défini comme étant l'intersection régularisée de six demi-espaces plans et un cylindre (fini) défini comme étant l'intersection régularisée d'un demi-espace cylindrique et de deux demi-espaces plans.

Un schéma CSG offre généralement un petit répertoire de solides primitifs et n'admet aucun demi-espace primitif, ceci afin d'assurer que les solides représentables dans le schéma soient bornés. On parlera dans ce cas de schéma CSG basé sur des primitives bornées (puisque les solides primitifs sont bornés, ce sont en fait des r-ensembles).

Un schéma CSG offrant des demi-espaces primitifs sera appelé schéma CSG basé sur des demi-espaces généraux. Un tel schéma ne garantit pas qu'un solide construit à partir de ces demi-espaces primitifs soit fini et donc que cet objet ait un sens physique.

Propriétés

- Ce schéma n'est pas unique. En effet, un solide peut être construit à partir de solides primitifs et d'opérateurs de maintes façons différentes. De plus, les sous-structures d'une représentation CSG peuvent être permutées (non unicité permutationnelle).

- Le domaine d'un schéma CSG dépend des demi-espaces qui sont sous-jacents aux solides primitifs et de la disponibilité d'opérateurs de mouvement et de combinaison.

- Dans le cas d'un schéma basé sur des primitives bornées (r-ensembles), tout arbre CSG est une représentation valide

d'un objet physique si les primitives feuilles de l'arbre sont valides (ceci ne serait pas le cas si on admettait l'opérateur de complémentarité qui peut engendrer des ensembles non bornés).

Comme la validité des primitives feuilles de l'arbre peut être vérifiée facilement, la validité d'un schéma basé sur des primitives bornées peut être vérifiée essentiellement à un niveau syntaxique.

On peut donc dire que toute représentation construite à partir de primitives instanciées correctement et au moyen d'opérateurs ensemblistes régularisés ou d'opérateurs de mouvement rigide est valide, c'est-à-dire correspond à un solide de l'espace de modélisation.

- Les arbres CSG des schémas basés sur des demi-espaces généraux (primitives non bornées) peuvent représenter des ensembles non bornés et donc être invalides. Dans ce cas, la vérification de validité est, d'un point de vue calcul, très coûteuse et requiert l'utilisation d'algorithmes non triviaux. Toutefois, les algorithmes travaillant avec des demi-espaces primitifs non bornés sont souvent plus simples à mettre en oeuvre que les algorithmes travaillant avec des solides primitifs bornés. C'est pourquoi il est commode de choisir les primitives bornées pour la représentation externe (afin de faciliter la validité) et de les convertir en primitives non bornées pour la représentation interne (afin de rendre plus simple les algorithmes travaillant sur celle-ci).

- Les schémas CSG dont les primitives sont bien choisies sont généralement concis. Les schémas basés sur des primitives bornées sont généralement beaucoup plus concis que ceux basés sur des primitives non bornées (ces derniers requièrent l'utilisation de plus de primitives).

- Les représentations CSG ne sont pas adéquates pour tout ce qui a trait au graphisme. Toutefois, elles apparaissent comme étant d'une importance capitale pour la fabrication automatisée de pièces.

2.5.5 Représentation par balayage

Ce schéma est, d'un point de vue théorique, l'un des moins compris (par les chercheurs dans ce domaine) de tous les schémas discutés dans ce chapitre. Il permet de construire un volume en faisant bouger un objet dans l'espace. Le volume est représenté par l'objet que l'on fait bouger plus la trajectoire que suit celui-ci.

De façon générale, un objet à 3 dimensions peut être obtenu soit par rotation, soit par translation d'un objet à 2 dimensions selon une trajectoire donnée.

Les propriétés de ces schémas sont largement déterminées par les schémas spécifiques utilisés pour représenter les objets à 2 dimensions.

Propriétés

- Ces schémas ne sont pas uniques.
- Les domaines de ces schémas sont limités aux objets qui possèdent des symétries obtenues par rotation ou par translation (domaines assez restreints).
- Ce schéma de représentation est important dans le domaine de la fabrication automatisée de pièces.

2.5.6 Représentation frontière

Dans ce schéma de représentation, un solide est représenté en segmentant sa frontière en un nombre fini d'ensembles bornés appelés faces, et en représentant chaque face par ses côtés frontières et ses sommets.

On suppose que chaque schéma de représentation frontière possède un nombre fini de surfaces primitives génériques (cas

à 3 dimensions) ou d'arêtes primitives génériques (cas à 2 dimensions).

Une liste de conditions, que nous ne citerons pas ici, doit être vérifiée afin d'assurer la validité d'une représentation frontière.

Propriétés

- Domaine : les schémas de représentation frontière couvrent en principe des domaines aussi riches que ceux des schémas CSG. En effet, on peut toujours construire un schéma de représentation frontière qui possède le même domaine qu'un schéma CSG en prenant comme surfaces primitives les frontières des demi-espaces primitifs.

- Les schémas de représentation frontière ne sont pas uniques.

- La validité des représentations frontières peut en général être testée en vérifiant certaines propriétés relatives aux faces, côtés et sommets qui, si elles sont vérifiées, permettent d'assurer que le solide a du sens. Toutefois, la validité est beaucoup plus coûteuse que dans le schéma CSG puisqu'elle ne peut être vérifiée à un niveau syntaxique.

- Les représentations frontières sont assez verbeuses.

- Etant donné qu'une représentation des faces et côtés est disponible, ce schéma de représentation est particulièrement bien adapté pour supporter l'affichage graphique ainsi que l'interaction graphique.

2.5.7 Résumé des caractéristiques des différents schémas

Dans cette section, nous présenterons un tableau récapitulatif reprenant, pour chaque schéma de représentation, une évaluation des différentes propriétés vues dans la section 2.4.

	Dom	Valid	Complét	Unicité	Concis°	Facilité de créat°
Instanciation de primitives		B	B	B	B	B
Enumération spatiale		B	B	B		
Décomposition en cellules	B		B			
CSG demi-espaces	B		B			
CSG primitives bornées	B	B	B		B	B
Balayage	B		B	?	B	B
Frontière	B		B	?		

Légende : B = Bon ? = Pas clair
Blanc = pauvre ou peu prometteur

2.5.8 Schémas hybrides

Les schémas dits hybrides ou non-homogènes sont des schémas qui sont construits en combinant différents schémas vus dans les sections précédentes. Parmi les plus connus, on peut citer :

1° Le schéma hybride CSG / frontière : les représentations sont des arbres CSG hybrides dont les feuilles sont soit des solides primitifs, soit des représentations frontières de solides non primitifs

2° Le schéma hybride CSG / balayage : de la même façon, les représentations sont de arbres hybrides dont les feuilles peuvent être des représentations par balayage de solides non primitifs

Ces schémas posent de sérieux problèmes en pratique car il faut concevoir des algorithmes qui s'accommodent aux différentes représentations.

2.5.9 Conversion entre représentations

En pratique, on voit qu'il n'existe pas de schéma qui soit, de façon générale, meilleur qu'un autre, c'est pourquoi il est parfois utile d'avoir à sa disposition plusieurs représentations d'un même solide afin de cumuler les avantages des différentes représentations. Bien entendu, cela ne peut se faire que si l'on dispose de mécanismes permettant de passer d'une représentation dans un schéma à une représentation correspondante dans un autre schéma.

On distingue deux types de conversion :

1° les conversions cohérentes ou exactes qui produisent une représentation du solide original;
2° les conversions approximées qui produisent une représentation d'un solide qui approxime le solide original.

Dans l'état actuel des connaissances, une conversion exacte entre tout schéma de représentation vers le schéma de représentation frontière est toujours possible. Toutefois, une conversion bidirectionnelle n'est pas encore possible pour deux raisons :

1° les domaines des différents schémas ne sont pas équivalents;
2° les algorithmes effectuant certaines conversions sont encore inconnus actuellement.

2.6 Les schémas de représentation dans notre système

Nous exposerons, dans cette section, les schémas de représentation sur lesquels se base notre système et verrons en quoi ceux-ci sont particulièrement bien adaptés aux exigences de notre système.

Il est clair que notre système doit contenir une représentation frontière des pièces puisque celle-ci est indispensable à la génération automatique des données (noeuds, éléments, ...). En effet, cette génération n'est pas directement réalisable à partir des autres représentations.

Comme il a déjà été mentionné auparavant, le logiciel BEM de calcul par éléments frontières ne supporte pas la troisième dimension, c'est pourquoi notre schéma de représentation frontière se réduit à deux dimensions. De plus, d'un commun accord avec l'utilisateur, nous prendrons comme côtés de base les segments linéaires et circulaires.

Toutefois, comme nous l'avons déjà dit, un schéma de représentation adéquat pour les algorithmes réalisant les fonctionnalités du système (dans notre cas, la génération) n'est pas forcément adéquat pour l'utilisateur. Ainsi, un langage d'entrée basé sur un schéma de représentation frontière aurait pour l'utilisateur les inconvénients majeurs suivants :

- toute description de pièce serait assez verbeuse;
- une description de pièce correcte d'un point de vue syntaxique pourrait ne pas être valide.

C'est pourquoi nous avons décidé avec l'utilisateur de baser notre langage d'entrée sur un autre schéma de représentation, à savoir le schéma CSG, celui-ci permettant de pallier les inconvénients cités ci-dessus. En effet, ce schéma est relativement concis et, pour autant que les solides primitifs soient bornés, toute représentation syntaxiquement correcte est valide.

De plus, il semble que ce schéma soit particulièrement bien adapté pour servir de base à un sous-système d'entrée. La majorité des systèmes de modélisation actuels basent d'ailleurs leur sous-système d'entrée sur un tel schéma. Il semble également, d'après l'expérience acquise grâce à ces systèmes, que l'homme peut avec beaucoup de facilité créer des représentations CSG de certaines classes d'objets.

Nous adopterons donc le schéma CSG comme second schéma de représentation avec comme solides primitifs les rectangles, les triangles et les cercles (ce qui est suffisant pour que le domaine soit au moins aussi grand que celui du schéma de représentation frontière). Les objets seront construits à partir de ces solides de base au moyen des opérateurs de mouvement de translation, de rotation et de symétrie et des opérateurs ensemblistes régularisés d'union, d'intersection et de différence. L'utilisation du schéma CSG nous garanti que toute représentation construite de cette façon à partir de primitives instanciées correctement correspond effectivement à un solide de l'espace de modélisation (la correction de l'instanciation pouvant être validée de façon triviale).

Notre système de génération de données possédant deux schémas de représentation distincts, un sous-système de conversion sera nécessaire pour pouvoir générer la représentation frontière d'une pièce à partir de sa représentation CSG. Un tel passage (conversion exacte) nécessite le recours à des algorithmes non triviaux. Les algorithmes d'évaluation de la frontière seront basés sur la notion de classification d'appartenance à un ensemble (Set membership classification). Cette technique sera étudiée de façon plus approfondie dans le chapitre 4.

Il est à noter que, ultérieurement, les solides primitifs apparaissant dans une représentation CSG seront remplacés par les demi-espaces sous-jacents (demi-espaces plans et portions de disques). En effet, les solides primitifs bornés sont encore d'un niveau de complexité trop élevé pour pouvoir être traités par les algorithmes opérant le changement de représentation.

Chapitre 3
Le compilateur

3.0 Introduction

Nous avons, jusqu'à présent, discuté du choix de représentation de solides. Il s'est avéré qu'une représentation (représentation CSG) semblait parfaitement convenir à l'utilisateur, de par sa puissance d'expression et par sa facilité d'utilisation, et qu'une autre (représentation frontière) convenait parfaitement aux besoins du module de génération des données pour les programmes d'éléments frontières. D'où la nécessité d'un transformateur pour effectuer la conversion entre elles.

Notre système devrait permettre d'éviter un inconvénient majeur, à savoir une longue et fastidieuse introduction de données à la main dans un fichier destiné à servir d'input aux programmes d'éléments frontières. Un langage est un moyen simple pour solutionner ce problème. Il offre à l'utilisateur la possibilité de définir ce qu'il veut dans un formalisme simple et précis. Les techniques de compilation permettent de transformer ce formalisme en un autre, plus proche de la solution finale.

Un compilateur est donc un programme qui lit un texte écrit dans un langage - le langage source - et le traduit en un programme équivalent dans un autre langage - le langage objet. Cette tâche est suffisamment complexe pour considérer le processus de compilation comme une interconnection de plus petits processus. On identifie essentiellement, pour un compilateur simple, trois parties différentes :

- un analyseur lexical chargé de lire le fichier de caractères en entrée et de regrouper ces caractères en symboles de grammaire.
- un analyseur syntaxique décide si la suite de symboles qu'il reçoit de l'analyseur lexical constitue une phrase du

langage. Le concept de phrase sera développé plus loin dans ce chapitre;
- un générateur de code qui génère la traduction de cette phrase dans le langage objet.

Notons que ces trois phases ne sont pas clairement séparées à l'exécution. En effet, l'analyseur lexical ne lira pas tout le fichier source en construisant un ruban de symboles qui servira par la suite d'input à l'analyseur syntaxique. Il en va de même entre l'analyseur syntaxique et le générateur. Ces trois phases interagissent entre elles de manière permanente : l'analyseur lexical détermine quel est le symbole suivant du fichier d'entrée et le donne directement à l'analyseur syntaxique. Celui-ci pourra, éventuellement passer la main au générateur, puis, lorsque le traitement du symbole sera terminé, on en reviendra à l'analyseur lexical pour obtenir le symbole suivant et ainsi de suite pour tout le fichier source.

Une section de ce chapitre est consacrée à la présentation détaillée du langage source mis au point pour satisfaire les besoins du système de génération automatique des données. Rappelons qu'il a été conçu en collaboration étroite avec les personnes du service M.S.M. de l'Université de Liège responsables des programmes d'éléments frontières auxquels ce langage devrait fournir une aide importante.

Le langage objet quant à lui servira d'input aux autres modules du système. Ce langage objet sera constitué :
- d'une représentation CSG du solide ainsi qu'une série de tables associées à cette représentation qui serviront au transformateur;
- d'une série de caractéristiques générales du solide, ainsi qu'une description des forces et déplacements imposés en certains endroits de la frontière du solide. Celles-ci seront fournies au module de génération des données.

Dans ce chapitre, nous développerons les concepts nécessaires à la construction d'un compilateur. Nous étudierons plus en détail les compilateurs weak precedence ascendants. Nous expliquerons et justifierons le choix de ce type de compilateur par rapport à d'autres alternatives

possibles. Le langage admis par le compilateur sera expliqué; ses possibilités et restrictions par rapport à notre problème seront mises en évidence. Enfin, une section sera consacrée à l'implémentation du compilateur.

3.1 Les compilateurs weak precedence ascendants

Dans cette section, nous développerons brièvement des concepts qui pourraient servir de base à la conception d'un compilateur. Nous particulariserons ces concepts dans le cadre des compilateurs weak precedence ascendants.

3.1.1 Généralités

a. Grammaire

- Un alphabet est un ensemble fini de symboles.
- Une chaîne sur un alphabet est une suite finie de symboles appartenant à l'alphabet.
- Une grammaire définit un langage par une règle de génération de l'ensemble de ses chaînes.

On va se définir deux alphabets :

- un alphabet terminal : un ensemble fini de symboles terminaux;
- un alphabet non terminal : un ensemble fini de classes syntaxiques (ou variables syntaxiques).

Parmi les variables syntaxiques, on en distingue une particulière : la variable distinguée. Pour chaque classe syntaxique, on définit un ensemble de règles de production. Dorénavant, pour décrire une grammaire ou une règle de grammaire, nous utiliserons la syntaxe B.N.F. (Backus Naur Form) bien connue.

b. Dérivations

1. Formes de phrase

Une forme de phrase est une suite finie de symboles construite au moyen des règles suivantes :

- la variable distinguée est une forme de phrase;
- Soit $W_0\ W_1\ W_2 \ldots W_i \ldots W_n$ une forme de phrase où W_i est, pour tout i, soit
  - un symbole terminal
  - une classe syntaxique

  Si dans la grammaire, il existe une règle de production $W_i ::= Z_0\ Z_1 \ldots Z_n$.
  alors $W_0\ W_1\ W_2 \ldots Z_0\ Z_1 \ldots Z_n \ldots W_n$ est une forme de phrase.

Passer d'une forme de phrase à une autre de cette façon revient à effectuer une étape de dérivation. Une série d'étapes de dérivations est appelée une dérivation.

2. Phrase

Une phrase est une forme de phrase qui ne contient que des symboles terminaux.

Pour un compilateur, un ruban de symboles est syntaxiquement correct s'il est une phrase.

c. Arbre d'analyse

Etant donnée une dérivation, il est possible de lui associer un arbre d'analyse. Cette construction est réalisée en interprétant les substitutions réalisées à chaque étape de dérivation comme étant des opérations de construction d'arbre en suivant certaines règles :

- les sommets (noeuds) non terminaux de l'arbre sont des symboles non terminaux;
- les feuilles de cet arbre sont des symboles terminaux;
- la racine de l'arbre est étiquetée avec la variable distinguée;
- soit une forme de phrase

  $W_0\ W_1\ W_2\ \ldots\ W_i\ \ldots\ W_n$

  On cherche dans la grammaire une règle dont W_i est partie gauche. Soit

  $W_i\ ::=\ Z_0\ Z_1\ Z_2\ \ldots\ Z_i\ \ldots\ Z_n$

  cette règle de grammaire. On crée alors, au noeud W_i, un sous-arbre pour chaque symbole de la partie droite de cette règle de grammaire.

  Donc, effectuer une étape de dérivation revient à construire de nouveaux sous-arbres. Finalement, la phrase est représentée par un arbre.

A un arbre d'analyse peut correspondre plusieurs dérivations, mais à une dérivation ne correspond qu'un seul arbre d'analyse.

L'arbre d'analyse montre donc quelles règles de production ont été appliquées pendant la dérivation, et à quelles occurences de symbole non terminal elles ont été appliquées. Cependant, cet arbre d'analyse ne fournit aucune information au sujet de l'ordre dans lequel ces règles de production ont été appliquées

Définissons la notion de dérivation la plus à droite. Soit une étape de dérivation

$W_0\ W_1\ W_2\ \ldots\ W_i\ \ldots W_n\ \rightarrow\ W_0\ W_1\ W_2\ \ldots\ Z_0\ Z_1\ \ldots\ Z_n\ \ldots W_n$

où l'on substitue $Z_0\ Z_1\ \ldots\ Z_n$ à W_i. Cette dérivation est appelée dérivation la plus à droite si et seulement si W_i était le symbole non terminal le plus à droite dans la forme de phrase $W_0\ W_1\ W_2\ \ldots\ W_i\ \ldots W_n$. La dérivation la plus à

droite consiste donc, à chaque étape de dérivation, à substituer toujours sur le symbole non terminal le plus à droite de la forme de phrase courante.

Ce type de dérivation correspond à une analyse syntaxique suivant une méthode ascendante (bottom-up) où l'on essaye de reconstruire la dérivation la plus à droite. Cela permet également d'examiner les symboles dans le ruban d'entrée de gauche à droite. Nous verrons dans la section 4.2 qu'il est possible de travailler différemment.

Les analyseurs bottom-up ont deux structures de données :

- Un stack qui contient une séquence de symboles qui peuvent être :
  - . des classes syntaxiques;
  - . des symboles terminaux;
  - . un symbole spécial;
- Un ruban de symboles (input) qui contient le texte à analyser.

On utilisera également une table : celle-ci permet de déterminer une action à effectuer en fonction d'éléments sur le stack et sur l'input. Ces actions sont :

- . un shift : prendre un certain nombre de symboles sur l'input et les placer sur le stack
- . un reduce : prendre des symboles en tête de stack et les remplacer par une classe syntaxique

Chaque couple (symbole stack , symbole input) possible fournit une entrée dans cette table. La particularité des analyseurs weak precedence est qu'ils prennent la décision d'une action à effectuer en fonction du symbole au sommet du stack et du symbole au sommet de l'input.

3.1.2 Algorithme général

```
begin
   error := false;
   "initialiser le stack avec "$""
   "initialiser l'input avec le ruban de symbole concaténé à
   "$""
   while non error and non configuration finale do
      t := SR [stack-top , input-first]
      case t of
         shift  : passage de "first input" sur "top stack"
         reduce : "opération de réduction"
         erreur : error := true
      endcase
   endwhile
end
```

Commentaires

1° Le symbole "$" est un symbole spécial : il n'appartient pas à la grammaire.

2° la "configuration finale" est une configuration telle que :
- sur le stack se trouve deux symboles : le symbole spécial "$" suivi de la classe syntaxique qui correspond à la variable distinguée;
- sur l'input, on trouve seulement le symbole spécial "$".

3° SR est le nom de la table de l'analyseur. SR [stack-top , input-first] est donc l'élément de la table qui correspond aux entrées du symbole du sommet du stack et de celui du sommet de l'input. La structure et la méthode de construction de cette table seront davantage discutées au point 3.1.3.

4° "L'opération de réduction" correspond à l'exécution d'une routine sémantique, et doit englober deux opérations :
- une opération qui consiste à trouver sur le stack une séquence de symboles qui forme une partie droite de règle de production de la grammaire. Dans cette recherche, soit
  . on n'en trouve pas : c'est une erreur.
  . on en trouve : on prend la plus longue d'entre elles.

Il faut alors remplacer cette séquence de symboles sur le sommet du stack par le symbole qui constitue la partie gauche de la règle choisie.

- simultanément à cette réduction, on peut générer une partie du code objet. Nous savons qu'une partie du code objet est constituée d'un arbre qui est la représentation CSG d'un objet. La génération de cet arbre constitue donc une partie importante de la génération du code. Si, par exemple, la réduction a été effectuée sur base d'une règle de grammaire permettant de spécifier une primitive de l'arbre CSG, alors on introduira dans cet arbre un noeud feuille supplémentaire. Ce noeud sera étiqueté avec le nom de la primitive, et on y associera une entrée dans une table donnant les caractéristiques d'instanciation de cette primitive.

3.1.3 Le générateur de table

On constate donc que le compilateur utilise constamment la table pour prendre les décisions d'actions à effectuer. Nous donnerons tout d'abord les règles qui permettent de déterminer quels sont les couples de symboles qui sont dans une situation de shift ou de reduce. Nous verrons ensuite l'algorithme qui permet de construire la table à partir de ces règles, et comment cet algorithme a été implémenté dans notre système.

a. Les shifts

On notera a < b le fait qu'il existe un shift pour le couple de symboles (a , b)

1. $ < VD (3.1)

   Remarque : VD = variable distinguée.

2. Si dans une partie droite de règle de production on a

   ... X Y ...

   alors X < Y (3.2)

3. Si X < Y et Y ::= L_0 ... L_n

   alors X < L_0 (3.3)

Ces trois règles seront à appliquer pour trouver tous les couples qui sont dans la situation du shift.

b. Les reduces

On notera a > b le fait qu'il existe un reduce pour le couple de symboles (a , b).

1. VD > $ (3.4)
2. si X < Y et X ::= L_0 ... L_n
   alors L_n > Y (3.5)
3. si X > Y et X ::= L_0 ... L_n
   alors L_n > Y (3.6)
4. si X > Y et Y ::= L_0 ... L_n
   alors X > L_0 (3.7)

c. Construction de la table

Les entrées dans la table sont
- l'ensemble des symboles terminaux (en ligne);
- l'ensemble des classes syntaxiques et des symboles terminaux de la grammaire, plus "$" (en colonne).

Pour a < b ou a > b, on entre en colonne avec a, en ligne avec b, et on inscrit shift ou reduce dans la case. Après remplissage de la table, les cases encore vides correspondent à "erreur".

Il faut s'assurer, lors de la construction, que pour un même couple de symboles, on ne trouve pas en même temps un shift et un reduce. Si cela se produit, c'est que la grammaire ne convient pas pour un analyseur weak precedence; il faut alors la modifier. Ce problème se produisant assez fréquemment, il faut parfois "bricoler" la grammaire pour la rendre analysable par un analyseur weak precedence.

Il faut également vérifier que l'on n'a pas, dans la grammaire, deux règles ayant deux parties droites identiques pour deux parties gauches différentes.

Si ce dernier problème peut être assez facilement résolu par une lecture attentive de la grammaire, le premier ne peut être résolu manuellement. En effet, on

pourrait très bien imaginer générer la table manuellement en appliquant les règles définies ci-dessus. Cependant, les conflits résultants de deux valeurs différentes pour une même case de la table nous obligent à utiliser la méthode d'essais et erreurs pour générer la table. On imagine une grammaire, on calcule tous les shifts et les reduces. S'il y a conflit, on modifie la grammaire et on répète l'expérience jusqu'à la disparition de tout conflit. Cette méthode nous interdit, bien sûr, tout procédé manuel de génération de cette table. Il faut donc construire un programme qui, à partir des règles de la grammaire, construit la table de façon automatique.

Voyons comment construire ce programme. En ce qui concerne les règles de production de la grammaire, on remarque par la règle 3.2 qu'il doit être possible de les parcourir séquentiellement. En outre, les règles 3.3, 3.5, 3.6 et 3.7 nécessitent un accès direct à une règle de production dont on connaît la partie gauche. Nous aurons donc, outre un tableau contenant la représentation des règles de production ordonnées selon leur partie gauche, un tableau permettant d'accéder aux règles dont on connaît la partie gauche. On réalise ceci en mémorisant pour chaque classe syntaxique les valeurs d'indices qui constituent la borne inférieure et supérieure dans le tableau des règles de production où cette classe sera partie gauche.

Il faut également trouver une structure de données qui contiendra les couples qui sont dans une situation de shift ou de reduce, en tenant compte du fait que :

1. il faut pouvoir éliminer les couples redondants;
2. il faut pouvoir trouver les couples dont le premier élément est une variable syntaxique;
3. il faut pouvoir trouver les couples dont le second élément est une variable syntaxique.

On rangera donc les couples dans trois listes :

1. une liste de couples dont les deux éléments sont des symboles terminaux;

2. une liste de couples dont le premier élément est une classe syntaxique;
3. une liste de couples dont le second élément est une classe syntaxique.

On peut remarquer deux choses :

1. il existera des couples qui pourront appartenir à deux listes en même temps. Il s'agira des couples dont les deux éléments sont des variables syntaxiques;
2. l'insertion d'un couple dans la liste se fera en queue de liste, s'il ne s'y trouve pas déjà.

Voyons l'effet de l'application des règles sur les trois listes des shifts :

- la règle 3.1 a pour effet d'insérer un couple dans la liste n°3;
- la règle 3.2 a pour effet d'ajouter des couples dans les trois listes;
- La règle 3.3 a pour effet de parcourir la liste n°3 et de remplir les trois listes.

Les couples shifts qui seront pris en compte pour la construction de la table seront ceux qui appartiennent à la liste n°1 et à la liste n°2 dont le second élément est un symbole terminal.

On utilisera la même structure de donnée pour les reduces, et

- la règle 3.4 a pour effet d'insérer un couple dans la liste n°2;
- la règle 3.5 a pour effet de parcourir la seconde liste des shifts et d'insérer des couples dans les trois listes;
- la règle 3.6 a pour effet de parcourir la liste n°2 et d'insérer des couples dans les trois listes;
- la règle 3.7 a pour effet de parcourir la liste n°3 et d'insérer des couples dans les trois listes.

Les éléments pris en compte pour construire la table seront ceux de la liste n°1 et de la liste 2 sauf

- le couple dont le premier élément est la variable distinguée;
- les couples dont le second élément est une variable syntaxique.

3.2 Alternatives envisagées

La section précédente développe certains concepts de base pour la conception d'un compilateur. Ces concepts ont été plus particulièrement approfondis dans le cadre des compilateurs weak precedence ascendants. Il existe évidemment d'autres alternatives possibles. Nous allons brièvement et en ne prétendant nullement être complets voir quelles autres possibilités s'offraient à nous.

Nous avons vu que les compilateurs weak precedence prennent les décisions d'actions qu'ils vont effectuer en fonction du symbole au sommet du stack et du symbole en tête de l'input. Il existe des compilateurs qui relâchent cette contrainte et prennent des décisions, par exemple, en fonction de plusieurs symboles sur l'input.

Nous avons également appliqué la méthode de dérivation la plus à droite, ce qui nous a conduit à une méthode ascendante (bottom-up). Il existe d'autres alternatives. Il est possible, par exemple, d'imaginer la dérivation la plus moyenne, ou la plus à gauche. Avec la dérivation la plus à gauche, on obtiendra une méthode d'analyse syntaxique descendante (top-down). L'analyseur ascendant cherche à trouver sur le stack une partie droite de règle de production de la grammaire. S'il la trouve, il la remplace par la partie gauche de la même règle. L'analyseur descendant remplace sur le stack une partie gauche d'une règle de grammaire par sa partie droite correspondante. Cette différence d'approche conduit à des avantages et inconvénients de part et d'autre. Comme le but de cette section n'est pas de savoir quel analyseur est le meilleur, nous n'entrerons pas dans les détails. Nous donnerons simplement deux exemples de difficultés rencontrées lors de la conception de la grammaire

pour notre compilateur. Nous justifierons le choix d'un compilateur weak precedence ascendant.

Voyons d'abord quelles sont les difficultés majeures que nous avons rencontrées.

1. Nous aurions voulu exprimer le fait que certains éléments d'une liste pouvaient être facultatifs. les règles de grammaire

   <vide> ::=

   <élément de la liste> ::= ... | <vide>

   auraient constitué une solution acceptable, mais elles étaient inconcevables dans le cadre d'un analyseur ascendant. En effet, comment pourrait-on voir si <vide> se trouve sur le sommet du stack ? Cela nous a obligé à écrire autant de règles de production que de combinaisons possibles des éléments de la liste c'est-à-dire

   $2^{\text{(nombre d'éléments facultatifs de la liste)}}$.

   Nous avons donc dû restreindre fortement les cas où un élément est facultatif, sous peine d'avoir un nombre très élevé de règles de grammaire.

   Remarquons qu'avec un analyseur descendant, nous aurions pu utiliser les règles de grammaire définies plus haut. En effet, en trouvant la classe syntaxique <vide> sur le stack, nous l'aurions remplacé par le vide, c'est-à-dire rien du tout.

2. Dans certains cas, nous aurions voulu laisser le choix à l'utilisateur de l'ordre dans lequel il devait spécifier certaines instructions. De nouveau, cela n'aurait été possible qu'en prenant en compte tous les cas envisageables dans les règles de grammaire, ce qui était inacceptable.

Cependant, l'analyseur que nous avons implémenté est un analyseur weak precedence ascendant, ceci pour la raison suivante. Un cours pratique nous a été donné sur la façon d'implémenter un tel compilateur. L'implémenter différemment nous obligeait à repartir à zéro et à reconstruire tout à partir de nos propres idées, ce qui doublait le risque d'erreurs.

Etant donné que le compilateur n'était qu'une étape de notre travail et que celui-ci devait être terminé et opérationnel avant de pouvoir poursuivre la conception des autres sous-systèmes, nous avons choisi la voie la plus rapide.

3.3 Le langage

Cette section sera consacrée à la discussion du langage mis au point pour satisfaire les besoins du système de génération automatique des données.

3.3.1 Explication du langage

Le langage a été conçu pour répondre aux besoins des utilisateurs. Avant tout, passons en revue ces besoins.

Rappelons que l'utilisateur désire définir des objets dans le plan pour pouvoir distribuer des noeuds sur la frontière de ces objets. Un objet, qu'il appelle domaine, est composé de un ou plusieurs sous-domaines. Il désire définir des caractéristiques propres à un domaine. Par exemple, il voudra définir le nombre de noeuds qui seront répartis sur la frontière du domaine. Les objets représentant des pièces réelles, il voudra définir une série de caractéristiques propres à cet objet. Pour chaque sous-domaine, l'utilisateur voudra se définir un poids qui influencera le nombre de noeuds qui seront répartis sur sa frontière. Pour pouvoir étudier la déformation de l'objet qu'il veut représenter, l'utilisateur voudra définir en certains points ou intervalles de la frontière, des forces ou déplacements qu'il impose en ces endroits particuliers du domaine.

A présent, nous allons voir comment le langage a été conçu pour répondre au mieux aux besoins de l'utilisateur (et pour lui donner la plus grande facilité d'utilisation).

L'utilisateur définit tout d'abord un domaine en donnant un nom (unique) au domaine qu'il veut créer :

Domain NomDeDomain =

Il peut ensuite déclarer un ensemble de sous-domaines qui composent le domaine :

Subdomain NomDeSousDomaine =

On assignera à chaque sous-domaine un objet de la manière suivante : NomDeSousDomaine = <Objet>

On assignera également un poids à chaque sous-domaine de la maniière suivante : ponderation = <réel>. Ceci influencera la répartition des noeuds sur la frontière de ce sous-domaine.

Notons que ces notions de domaine et sous-domaines ont été introduites dans notre langage exclusivement à la demande de l'utilisateur. Elles n'ont donc de signification que pour les programmes d'éléments frontières pour lesquels les données sont produites, et il n'est pas nécessaire d'en connaître la signification dans notre système.

En ce qui concerne la représentation des objets dans le plan, on peut rappeler que le compilateur produira une représentation CSG de l'objet et que le langage est orienté de manière à faciliter la production de ce résultat. Rappelons également que la représentation CSG est une représentation où les objets sont des arbres dont les noeuds sont des opérateurs binaires ou unaires, et dont les feuilles sont des primitives.

Les déclarations d'objets se font de la façon suivante :

&NomObjet = <Objet>

Les noms d'objets sont donc précédés du signe & (et commercial). La raison est la suivante : pour plus de concision des textes sources, nous ne voulions pas obliger l'utilisateur à se livrer à toute une série de déclarations d'identificateurs. Comme notre langage ne contenait que deux types d'identificateurs différents (objets et points), et pour éviter une série de déclarations, nous avons pris la

convention de faire précéder d'un & (et commercial) les identificateurs d'objets, ce qui les différencie des identificateurs de points, qui eux sont précédés de % (pour-cent).

Un <Objet> peut être :

- un identificateur d'objet déclaré précédemment

- un objet primitif. Nous avons choisi de doter le langage de trois types d'objets primitifs :

  . des rectangles : un rectangle est déclaré en donnant sa longueur et sa largeur, ainsi que, facultativement, son coin inférieur gauche. Par exemple,

  ```
  Rect (5;10) At (1,1)
  Rect (1;11)
  ```

  Notons que si le coin inférieur gauche n'est pas spécifié, alors ce coin est situé par défaut à l'origine du système d'axe.

  . des cercles : un cercle se définit en donnant son rayon et, facultativement, son centre. Par exemple,

  ```
  Circle (8) At (1,1)
  Circle (8)
  ```

  Le centre du cercle est par défaut situé à l'origine du système d'axe

  . des triangles : on définit un triangle par les trois points qui constituent ses sommets. Par exemple

  ```
  Triangle ((0,0);(1,0);(0,1))
  ```

- un objet auquel on fait subir un mouvement. Notre langage est doté de trois opérations de mouvement :

  . des translations : une translation est définie en donnant
  1° un objet à translater;
  2° la quantité dont il faut translater l'objet selon l'axe X et Y.

Par exemple,

```
Trl &NomObjet By (2;-2)
Trl Rect (1;1) At (0,0) By (-1;-1)
```

. des rotations : une rotation se spécifie en donnant
1° un objet auquel il faut appliquer la rotation;
2° un point définissant le centre de rotation, et l'angle de rotation.

Par exemple,

```
Rot &NomObjet By (2;-2)
Rot Rect (1;1) At (0,0) By (-1;-1)
```

. des symétries : une symétrie se définit en donnant
1° un objet auquel il faut appliquer une symétrie;
2° deux points distincts définissant l'axe de la symétrie.

Par exemple,

```
Sym &NomObjet By ((0,0);(1,1))
Sym Circle (8) At (2,1) By ((0,0);(1,1))
```

- une opération ensembliste de deux objets. Notre langage fournit trois opérations ensemblistes :

. une union de deux objets. Par exemple,

```
Union &NomObjet1 With &NomObjet2
Union Rect (3;3) With Circle (8) At (10,10)
Union Trl Rect (3;3)
      By (2,1)
With &NomObjet
```

. une intersection de deux objets. Par exemple,

```
Inter &NomObjet1 With &NomObjet2
Inter Rect (3;3) With Circle (8) At (10,10)
Inter Circle (8)
With Rot &NomObjet
     By ((0,0);-90)
```

. une différence de deux objets. Par exemple,

```
Diff Inter Rect (3;3)
     With Sym Circle (8) At (2,1)
          By ((0,0);(1,1))
With Union Trl &NomObjet
           By (2,1)
     With &NomObjet
```

Comme le montre ce dernier exemple, on peut combiner à volonté ces différentes manières de définir des objets. Notons que l'indentation rend le texte assez clair et compréhensible.

Pour encore plus de facilité lors de la création de ces objets, l'utilisateur pourra se définir des points intermédiaires dans le plan qu'il réutilisera plus tard partout où un point sera permis. Pour définir un point, on peut utiliser soit les coordonnées cartésiennes, soit les coordonnées polaires, en faisant précéder celles-ci du symbole P. Pour des raisons évoquées ci-dessus, les identificateurs de points seront précédés de % (pour-cent). On peut donner comme exemple les déclarations suivantes :

```
%a = (1,1)
%b = P (1,45)
```

Notre langage offre également la possibilité de modifier, à tout moment, l'origine du sytème d'axe. Par exemple,

```
Origin = (5,10) ou Origin = P (5,90)
```

finalement, l'utilisateur a encore la possibilité de définir des constantes réelles réutilisables par la suite partout où un réel est permis. Par exemple,

```
Pi = 3.1415927
```

En ce qui concerne les forces ou les déplacements en un point ou dans un intervalle de la frontière de l'objet, l'utilisateur se définit des ensembles de sollicitations qu'il peut nommer :

```
SET OF SOLLICITATIONS NomEnsembleSollicitations
```

Dans un ensemble de sollicitations, on peut définir une suite de déclarations de sollicitations. Par déclaration de sollicitation, on entend :

- une déclaration de sollicitation ponctuelle définie par exemple de la façon suivante :

```
On (1,1) : Forces = (1;/)
           Displacements = (/;2)
On %Point : Forces = (/;/)
            Displacements = N (1,2)
```

On spécifie l'endroit de la frontière de l'objet où la sollicitation ponctuelle est imposée au moyen d'un point. On spécifie ensuite les forces et/ou déplacements imposés selon les deux axes X et Y. Deux et exactement deux valeurs doivent être imposées. Les deux valeurs inconnues sont représentées par un / (slash). Notons que les forces et/ou déplacements au point peuvent être spécifiés dans le système d'axe Normal/Tangent, en utilisant la lettre N.

- une déclaration de sollicitation répartie définie par exemple de la façon suivante :

```
In ((0,0);(1,1)) : Forces = (0:1;/)
                   Displacements = (/;2:4)
In (%point1;(0,0)) : Forces = (/;/)
                     Displacements = N (1:3;1:2)
```

On spécifie l'endroit où la sollicitation répartie est imposée au moyen d'un intervalle défini par deux points distincts. Les mêmes possibilités sont offertes que dans le cas ponctuel. Dans le premier exemple, on impose au premier point (0,0) du segment une force de 0 selon l'axe X et un déplacement de 2 selon l'axe Y, et au deuxième point (1,1) de l'intervalle, on impose une force de 1 selon l'axe X, et un déplacement de 4 selon l'axe Y.

- un mouvement appliqué à un ensemble de sollicitations comme par exemple :

. la translation d'un ensemble de sollicitations :

```
Trl NomEnsembleSollicitations By (2;2)
```

. la rotation d'un ensemble de sollicitations :

```
Rot NomEnsembleSollicitations By (%point;-120)
```

. la symétrie d'un ensemble de sollicitations :

```
Sym NomEnsembleSollicitations By (%point;(0,0))
```

Finalement, notre langage permet de définir des caractéristiques propres à la pièce, ainsi que des paramètres de génération. Comme ces caractéristiques résultent uniquement d'une demande de l'utilisateur, et ne sont, pour la plupart, utiles que pour les programmes d'éléments frontières, on se contentera de les citer. On trouvera donc :

- la déclaration du nombre total de noeuds à distribuer sur l'ensemble des sous-domaines;
- une déclaration du nombre de points d'intégration;
- une déclaration d'intensité et de direction de la gravité;
- une déclaration du poids volumique de l'objet;
- une déclararation du module d'élasticité;
- une déclaration du coefficient de poisson;

On trouvera en annexe la syntaxe complète du langage exprimée en B.N.F.. On y trouvera également quelques exemples de textes sources acceptés par le compilateur accompagnés des représentations dans le plan des objets que ces textes définissent.

3.3.2 Restrictions

- La restriction la plus évidente est que le langage permet de définir des objets dans le plan alors qu'une pièce est toujours tridimensionnelle. Il est à noter que cette restriction est volontaire et uniquement due aux programmes d'éléments frontières qui, pour l'instant, ne supportent pas la troisième dimension. Il aurait été aisé de construire un langage semblable à celui défini, qui supporte la troisième dimension.

- Nous aurions pu imaginer un langage qui offre des opérateurs plus évolués, c'est-à-dire des opérateurs avec un nombre quelconque d'arguments. Une nouvelle fois, cela n'est pas une limitation due au langage lui-même, mais au mode de

représentation choisi pour les objets. La représentation CSG utilise des arbres binaires. Nous aurions facilement pu imaginer un langage dont les opérateurs peuvent s'appliquer à un nombre quelconque d'arguments.

- Concernant les opérateurs binaires, nous avons dû nous limiter dans leur nombre. Peut-être certains utilisateurs regretteront-ils parfois l'absence de l'un ou l'autre ?

- Il en va de même avec les opérateurs unaires. En effet, nous aurions pu offrir, par exemple, les mécanismes de symétrie centrale, d'homothétie,...

- Une autre restriction concerne peut-être les primitives. En effet, nous aurions pu offrir plus de primitives de base à l'utilisateur. Par exemple, un parallélogramme, un losange, un pentagone, un hexagone,... auraient été bien utiles. Cependant, il est à noter que ces nouvelles primitives n'auraient pas pour autant augmenté le domaine, c'est-à-dire l'ensemble des solides représentables dans le schéma CSG.

- Une restriction concerne le choix des demi-espaces primitifs. Nous avons choisi des demi-espaces plans et circulaires. Des demi-espaces ellipsoïdaux auraient pu être envisagés. En tout cas, ils auraient augmenté le domaine.

- Le langage ne permet pas de spécifier des expressions qui auraient peut-être été utiles lors de la définition des points, constantes, origines, objets, forces ou déplacements.

- Le langage ne permet pas la déclaration de procédures (ou de macros). Une procédure aurait permis de construire des super-primitives, à partir des primitives de base. Un appel paramétré de ces procédures nous aurait fournit une instanciation de ces super-primitives. Par exemple, on aurait pu définir une procédure comme suit :

```
procedure quadrilatère (p1,p2,p3,p4)
begin
   quadrilatère = Union Tri (p1,p2,p3)
                  With Tri (p1,p3,p4)
end
```

Un appel de procédure quadrilatère ((0,0),(0,1),(1,0),(2,2)) nous fournirait une instanciation d'un quadrilatère de sommet (0,0),(0,1),(1,0) et (2,2).

C'est suite aux nombreuses discussions avec l'utilisateur que nous avons, de commun accord, décidé de nous limiter.

3.4 Implémentation

On ne détaillera ici ni l'algorithme ni les structures de données que nous avons mises en oeuvre pour implémenter le compilateur. On rappelle simplement que, comme tout compilateur, le nôtre lit un texte source exprimé dans le langage décrit à la section 3.3 et produit la traduction de ce texte. Cette traduction se compose

- d'une représentation CSG de l'objet décrit dans le fichier source;
- d'une série de tables où on trouvera des informations associées
    - aux primitives;
    - aux opérateurs unaires;
    - aux forces et déplacements imposés en certains endroits de la frontière de l'objet;
    - aux caractéristiques propres à l'objet.

qui seront sauvées sur un fichier résultat de la compilation.

On peut encore formuler deux remarques concernant notre implémentation. La première concerne la génération des messages d'erreurs. Nous avons, dans la mesure du possible, essayé de produire des messages d'erreurs qui étaient significatifs pour l'utilisateur. Il y a cependant des cas où ces messages ne sont pas très significatifs. Par exemple, lorsque, en entrant dans la table, l'analyseur lexical trouve une erreur, le message produit est alors "SYNTAX ERROR". Ceci

vient du fait qu'il était très difficile et coûteux (en temps et lignes de code) de gérer de telles erreurs.

La seconde remarque est que, au niveau du compilateur, aucune vérification de validité de l'objet que l'utilisateur définit n'est faite. En effet, la plupart des vérifications ne peuvent être faites que si l'on dispose de la représentation frontière de l'objet. De telles vérifications seront donc laissées à un module ultérieur du système de génération automatique de données. Les seules vérifications effectuées à ce niveau sont des vérifications de primitives; On vérifie que les primitives sont instanciées correctement (par exemple, que tous les cercles sont définis avec un rayon positif).

Nous avons vu, dans le chapitre trois que le schéma de représentation CSG n'était pas unique, ce qui signifie qu'à un objet peut correspondre plusieurs représentations possibles. Un objet peut donc être exprimé dans le langage de plusieurs façons différentes. Il n'existe aucune méthodologie particulière dictant la manière de procéder pour décrire un objet de façon constructive. Tout au plus peut-on, pour terminer, donner quelques conseils :

- essayer de trouver les symétries de l'objet de façon à pouvoir générer simplement certaines parties de l'objet au moyen des opérateurs de translation, symétrie et rotation;
- se définir des objets intermédiaires de manière à ne pas entrer dans des niveaux d'emboîtement des opérateurs trop profonds. De plus, on pourra réutiliser ces objets intermédiaires plus tard comme s'ils étaient des primitives;
- utiliser fréquemment les possibilités de changement de l'origine, de définitions de points (en coordonnée polaire éventuellement) et de définitions de constantes rendra également la tâche de l'utilisateur plus simple.

Chapitre 4
Le transformateur

4.0 Introduction

Ce chapitre est consacré à l'explication du sous-système, appelé transformateur, qui s'occupe du passage de la représentation CSG générée par le compilateur à la représentation frontière nécessaire au générateur automatique de données. Dans un premier temps, nous détaillerons la méthode de classification sur laquelle est basé le transformateur. Nous verrons ensuite comment cette méthode sera utilisée pour générer une représentation frontière d'un solide à partir d'une représentation CSG de ce même solide. Nous expliquerons brièvement les différentes étapes de notre sous-système de conversion ainsi que ce qui a du être implémenté. Nous terminerons par quelques considérations relatives aux possibilités d'optimisation de la méthode et de notre système.

4.1 La méthode de classification

4.1.1 Introduction

La méthode dite de classification d'appartenance à un ensemble, que nous appellerons simplement méthode de classification, est la démarche la plus fréquemment utilisée pour répondre à des problèmes de type géométrique. Cette méthode possède le grand avantage de reposer sur des fondements mathématiques solides, permettant par là une vérification mathématique des algorithmes élaborés.

Parmi les problèmes géométriques les plus fréquemment rencontrés, on citera :

- l'inclusion de points : étant donné un point p et un solide S, déterminer si ce point p se trouve à l'intérieur, sur la frontière ou à l'extérieur du solide S;
- le fractionnement d'un segment par rapport à un polygone : étant donné un segment L (linéaire ou curviligne) et un

polygone P, fractionner le segment L en 3 parties disjointes. Une partie qui se trouve à l'intérieur de P, une partie qui se trouve sur la frontière de P et une partie qui se trouve à l'extérieur de P;

- l'intersection de polygones : étant donné deux polygones P et Q, trouver le polygone R qui est l'intersection des deux polygones P et Q ($R = P \cap Q$);
- l'interférence de solides : étant donné deux solides R et S, déterminer si ces deux solides interfèrent (c'est-à-dire se coupent), et si oui, trouver à quel endroit;
- la conversion entre représentations : étant donné la représentation CSG d'un solide, générer sa représentation frontière.

Tout au long de ce chapitre, nous ferons référence au modèle mathématique des solides expliqué dans la section 2.1. Ainsi, nous modéliserons les solides par des sous-ensembles réguliers, fermés et bornés de l'espace euclidien E^3 (r-ensembles). Les notions d'intérieur, de fermeture, de frontière et de régularité ont également été définies dans la section 2.1.

4.1.2 Approche intuitive de la méthode

La méthode de classification permet de répondre aux problèmes présentés ci-dessus d'une façon relativement uniforme, au moyen d'une fonction de classification M[X,S] à deux arguments : un ensemble S dit ensemble de référence et un ensemble X dit ensemble candidat. Cette fonction a pour but de partitionner ou segmenter l'ensemble candidat X en 3 sous-ensembles disjoints XinS, XonS et XoutS. Ces trois sous-ensembles correspondent aux parties de l'ensemble candidat X qui se trouvent, respectivement, à l'intérieur, sur la frontière et à l'extérieur de l'ensemble de référence S.

4.1.3 Approche mathématique de la méthode

Les ensembles utilisés seront des r-ensembles et les opérateurs ensemblistes seront les opérateurs régularisés d'union, d'intersection et de différence ($\cup^*, \cap^*, \setminus^*$).

La fonction de classification est une fonction des deux ensembles de points suivants :

1° S : ensemble de référence régulier dans un sous-espace W de E^n.

2° X : ensemble candidat régulier dans un sous-espace W' de W.

Aux différents choix de W et W' correspondront différents cas particuliers de la méthode de classification.

Par exemple :

- pour le problème d'intersection de deux polygones P et Q, on prendra $W = E^2$ et $W' = E^2$;
- pour le problème de partition d'un segment L par rapport à un polygone P, on prendra $W = E^2$ et $W' = E^1$.

On notera que les différentes notions d'intérieur, de fermeture, de frontière et de régularité, ainsi que les opérateurs ensemblistes sont différents selon l'espace dans lequel on travaille. Généralement, on utilise des symboles primés pour dénoter des opérations dans W' et des symboles non primés pour dénoter des opérations dans W. Toutefois, afin d'alléger les notations, nous ne ferons pas la distinction.

Avant de donner une définition de la fonction de classification, il est peut-être utile de rappeler quelques définitions établies dans la section 2.1 :

Soit X un sous-ensemble de E^n.

- Un point p est un élément de l'intérieur de X, iX, ssi il existe un voisinage de p qui est contenu dans X.
- Un point p est un élément de la fermeture de X, kX, ssi tout voisinage de p contient un point de X.

- Un point p est un élément de la frontière de X, bX, ssi p est à la fois un élément de kX et de k(cX) où cX dénote le complémentaire de X.
- L'ensemble X est dit régulier ssi X = kiX c'est-à-dire ssi il est égal à la fermeture de son intérieur. La régularisation de X, rX, est définie comme rX = kiX.
- L'union régularisée de deux ensembles A et B est définie par : A U* B = r(A U B)
- L'intersection régularisée de deux ensembles A et B est définie par : A ∩* B = r(A ∩ B)
- Le complément régularisé d'un ensemble A est défini par : c*A = rcA

La fonction de classification M[X,S] peut être définie mathématiquement de la façon suivante :

M[X,S] = (XinS,XonS,XoutS)
où XinS = X ∩*iS = ki(X ∩ iS)
XonS = X ∩*bS = ki(X ∩ bS)
XoutS = X ∩*cS = ki(X ∩ cS)

On peut, à partir de cette définition, faire les deux remarques suivantes :
- presque tous les points (pas tous) de XinS sont dans iS, presque tous les points de XonS sont dans bS et presque tous les points de XoutS sont dans cS;
- cette fonction produit une quasi-partition de l'ensemble candidat X, c'est-à-dire :

X = XinS U XonS U XoutS
XinS ∩ XonS, XinS ∩ XouS et XonS ∩ XoutS ne comportent qu'un nombre fini de points

Les trois ensembles XinS, XonS et XoutS ne sont pas disjoints dans le sens traditionnel car il se peut, par exemple, que XinS et XonS partagent un même point. Ceci découle de la première remarque.

4.1.4 Classification et schéma CSG

Dans cette section, nous montrerons comment la méthode de classification aborde le problème de façon récursive dans le cas d'un ensemble de référence représenté selon le schéma CSG.

Le schéma de représentation CSG a été détaillé dans la section 2.5.4. Nous apporterons cependant une petite restriction quant aux opérateurs utilisés dans les arbres CSG. Bien que l'opérateur régularisé de complémentarité conserve la régularité des ensembles, cet opérateur ensembliste sera écarté car il ne conserve pas le caractère borné. Nous exclurons également des arbres CSG les opérateurs de mouvement rigide, ceux-ci pouvant être absorbés dans les primitives (on peut supprimer un noeud de translation ou de rotation en appliquant cette translation ou cette rotation à toutes les primitives qui sont descendantes de ce noeud).

La récursivité est une approche naturelle pour calculer la valeur d'une fonction f d'un ensemble régulier S représenté de façon constructive. Lorsque S n'est pas primitif, c'est-à-dire lorsque S est de la forme A <op> B, le problème du calcul de la valeur f(S) peut être décomposé en deux sous-problèmes qui sont le calcul de f(A) et le calcul de f(B). Ces deux valeurs peuvent alors être combinées, d'une façon qui dépend de l'opérateur <op>, pour donner lieu à la valeur f(S). Lorsque S est primitif, aucune décomposition du problème n'est possible. Il faut donc avoir à notre disposition un moyen pour calculer f(S) lorsque S est primitif.

Il faut cependant remarquer que, de façon générale, il n'est pas possible de calculer n'importe quelle fonction selon cette approche. Bien souvent, il est nécessaire d'englober la fonction que l'on veut calculer dans une fonction qui généralise la première, c'est-à-dire qui fait usage de plus d'informations.

Cette approche peut être appliquée au calcul de la fonction de classification M[X,S] lorsque l'ensemble S est

représenté de façon constructive. Le calcul de M[X,S] peut se faire selon l'algorithme suivant :

```
if (S est un solide primitif)
then Prim-M[X,S]
else ( S est un solide de la forme A <op> B )
     combine ( M[X,A] , M[X,B] , <op> )
```

Selon le problème que la méthode de classification est sensé résoudre, il reste à définir les procédures primitives de classification prim-M[X,S] (pour chacun des objets primitifs) ainsi que les procédures de combinaison "combine" (pour chaque opérateur <op> possible).

Exemple : problème d'inclusion d'un point dans un polygone :

- X est un point;
- S est un polygone représenté de façon constructive à partir des demi-espaces plans;
- Prim-M[X,S] est le problème primitif de classification qui consiste à déterminer si le point est à l'intérieur, sur la frontière ou à l'extérieur d'un demi-espace plan, ce qui est un problème quasiment trivial;
- la procédure de combinaison est cependant plus subtile et sera expliquée dans la section 4.1.5.

4.1.5 Formules récursives de classification

Nous verrons dans cette section que certaines ambiguïtés peuvent apparaître dans les procédures de combinaison. Nous expliquerons ensuite comment ce problème peut être résolu. Nous donnerons finalement des formules récursives permettant d'exprimer, pour tout opérateur ensembliste régularisé <op>, M[X,S] en fonction de M[X,A] et M[X,B] lorsque S = A <op> B.

Le problème d'ambiguïté

Le problème d'ambiguïté réside dans le fait qu'il n'est pas possible d'obtenir le résultat M[X,S] à partir des seules

informations M[X,A] ,M[X,B] et <op>. En effet, prenons l'exemple de la classification d'un point par rapport à un solide non primitif. La figure 4.1 nous montre que le point candidat p est sur la frontière de A et sur la frontière de B. Si l'on prend par exemple S = A ∩*B, dans l'exemple de gauche, le point p se trouve sur la frontière de S, alors que dans l'exemple de droite, le point p se trouve à l'extérieur de S puisque S est vide. Le résultat varie donc selon la situation. Cette ambiguïté, appelée communément ambiguïté on/on, apparaît également pour les opérateurs régularisés d'union et de différence.

Ce problème d'ambiguïté ne peut se résoudre sans avoir recours à l'utilisation d'informations supplémentaires. Nous enrichirons donc la fonction M[.] pour finalement aboutir à la fonction définitive M*[.].

Pour résoudre l'ambiguïté on/on, il faut utiliser une information nous disant de quel côté se trouve les solides A et B par rapport au point p. Nous formaliserons cela par la notion de voisinage régularisé de la façon suivante :

si S est un ensemble régulier dans W et p est un point de W, alors nous définirons le voisinage régularisé de rayon R > 0 de p vis-à-vis de S comme étant

$$N(p,S;R) = B(p;R) \cap^* S$$

où B(p;R) est la boule ouverte de rayon R et de centre p.

A partir de cette définition, on peut dire que :

- le point p est à l'intérieur de S ssi il existe un rayon R > 0 tel que le voisinage régularisé de rayon R est la boule toute entière (on dira si son voisinage régularisé est plein) c'est-à-dire :
  $p \in iS \Leftrightarrow$ il existe R > 0 tel que $N(p,S;R) = kB(p;R)$
- le point p est à l'extérieur de S ssi il existe un rayon R > 0 tel que le voisinage régularisé de rayon R est vide c'est-à-dire
  $p \in ckS \Leftrightarrow$ il existe R > 0 tel que $N(p,S;R) = \emptyset$

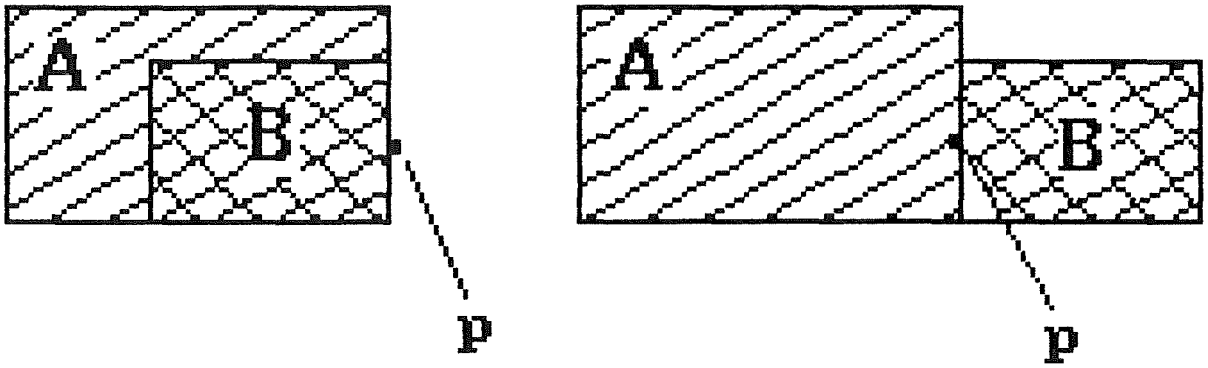


Fig 4.1 ambiguite "on/on" pour la classification d'un point.

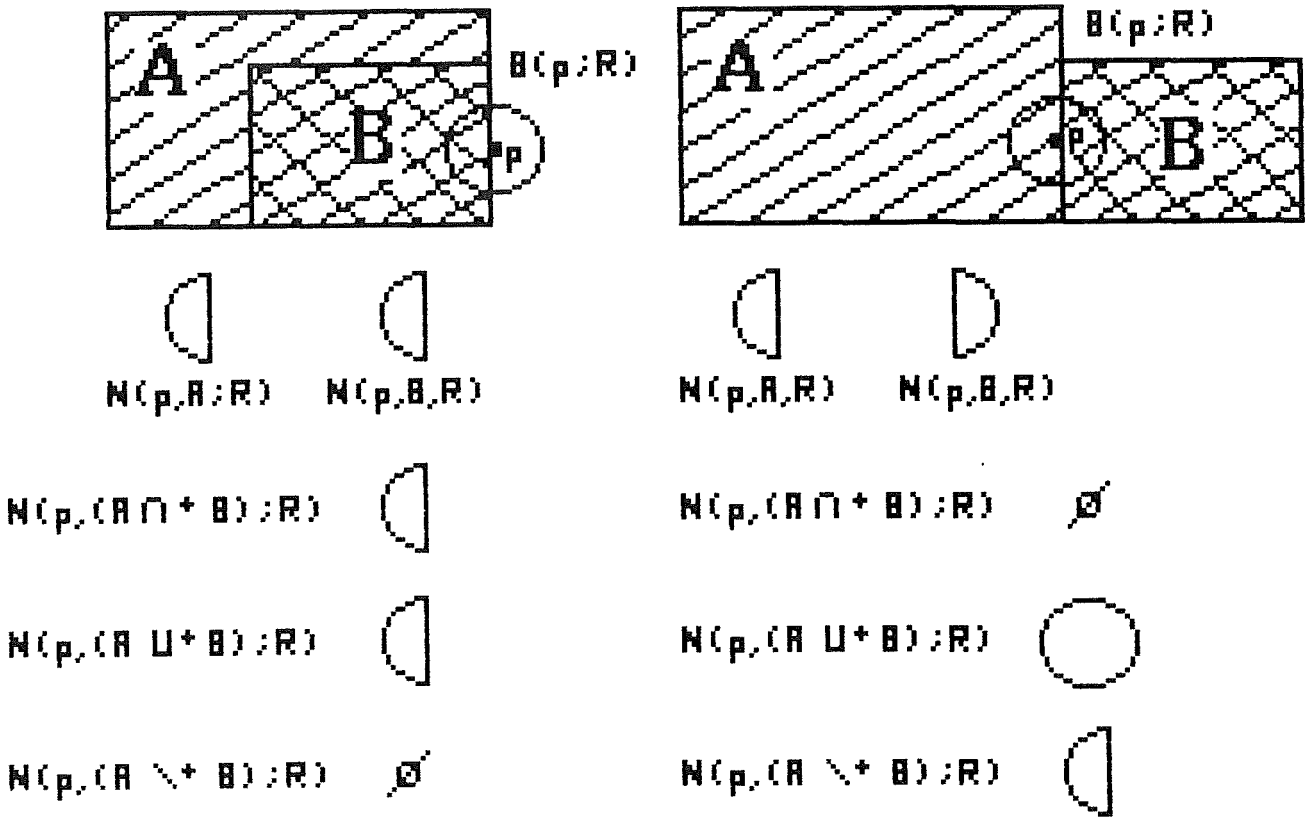


Fig 4.2 Voisinage regularise

- le point p est sur la frontière de S ssi pour tout rayon R > 0 le voisinage régularisé de rayon R n'est ni plein ni vide c'est-à-dire :

  p ∈ bS <=> pour tout R > 0 : N(p,S;R) <> ∅ et
  N(p,S;R) <> kB(p;R)

Le voisinage régularisé de p pour S peut être obtenu à partir des voisinages régularisés de p pour A et B de la façon suivante :

$$N(p,S;R) = N(p,A;R) \langle op \rangle N(p,B;R) \text{ si } S = A \langle op \rangle B$$

où <op> est U*, ∩* ou *

La figure 4.2 permet d'illustrer comment la notion de voisinage régularisé peut être utilisée pour résoudre le problème d'ambiguïté on/on. Dans l'exemple de gauche, le voisinage régularisé N(p,(A∩*B);R) de p pour S=A ∩*B, obtenu par intersection régularisée des voisinages régularisés N(p,A;R) de p pour A et N(p,B;R) de p pour B, n'est ni vide ni plein, ce qui signifie que le point p appartient bien à la frontière du solide S=A ∩*B. Par contre, dans l'exemple de droite, le voisinage de p pour S obtenu de la même façon est vide, ce qui signifie que le point p n'appartient pas à la frontière du solide S. Le cas où S=A U* B est similaire. Dans le cas où S=A * B, les conclusions sont inversées (le point p appartient à la frontière de S dans l'exemple de droite et pas dans l'exemple de gauche).

La notion de voisinage a été définie dans le cas particulier d'un point. Elle peut être généralisée pour tout sous-ensemble x de X régulier dans W'. Nous définirons une fonction de voisinage, notée N*[x,S](p;R) ou de façon abrégée N*[x,S], qui, à chaque point p de x et pour tout rayon R > 0, associe un voisinage N(p,S;R) tel que défini précédemment.

La fonction de voisinage est une source suffisante d'informations pour résoudre les problèmes d'ambiguïtés on/on rencontrés dans les procédures de combinaison.

On utilise la fonction de voisinage pour définir de façon mathématique où sont situés les points d'un ensemble par rapport à un solide. En pratique, on peut résoudre le problème d'ambiguïté plus facilement en utilisant des conventions. Il suffira, par exemple, d'orienter les côtés d'un polygone de façon à ce que l'intérieur du polygone soit à la gauche des côtés. Pour les surfaces, on peut prendre, par exemple, la direction de la normale à la surface pour distinguer ce qui est à l'intérieur de ce qui est à l'extérieur.

La fonction enrichie M*[X,S] sera définie de la façon suivante :

M*[X,S] = (XinS,(XonS,N*[XonS,S]),XoutS)

Ceci nous montre que, pour calculer la partie de X qui est sur S, on fera appel à la fonction de voisinage N*[XonS,S] de XonS par rapport à S afin de résoudre les problèmes d'ambiguïté on/on.

Formules récursives

Des formules récursives peuvent être obtenues pour exprimer M*[X,A <op> B] en termes de M*[X,A] et M*[X,B]. Ces formules font appel à la fonction de voisinage pour résoudre les ambiguïtés on/on.

Formule 1 :

si S = A ∩*B, alors

1° XinS = XinA ∩* XinB

2° XonS = t_1 U* t_2 U* t_3 où

t_1 = XonA ∩* XinB

t_2 = XinA ∩* XonB

t_3 = r{ p ∈ XonA ∩* XonB tq N*[XonA ∩* XonB,S](p;R) n'est ni vide ni plein pour tout R > 0}

3° XoutS = X * (XinS U* XonS)

L'ensemble t_3 représente l'ensemble des points parmi ceux de XonA ∩* XonB qui appartiennent effectivement à la frontière

du solide S = A ∩* B. Intuitivement, dans le cas où X est un segment par exemple, on obtient t_3=XonA ∩* XonB lorsque les solides A et B se trouvent du même côté par rapport au segment XonA ∩* XonB, le vide sinon.

Formule 2 :

si S = A U*B, alors

1° XoutS = XoutA ∩* XoutB

2° XonS = t_1 U* t_2 U* t_3 où

t_1 = XonA ∩* XoutB

t_2 = XoutA ∩* XonB

t_3 = r(p ∈ XonA ∩* XonB tq N*[XonA U* XonB,S](p;R) n'est ni vide ni plein pour tout R > 0)

3° XinS = X * (XoutS U* XonS)

Intuitivement, dans le cas où X est un segment par exemple, on obtient t_3=XonA ∩* XonB lorsque les solides A et B se trouvent du même côté par rapport au segment XonA ∩* XonB, le vide sinon.

Formule 3 :

si S = A *B, alors

1° XinS = XinA ∩* XoutB

2° XonS = t_1 U* t_2 U* t_3 où

t_1 = XonA ∩* XoutB

t_2 = XinA ∩* XonB

t_3 = r(p ∈ XonA ∩* XonB tq N*[XonA * XonB,S](p;R) n'est ni vide ni plein pour tout R > 0)

3° XoutS = X * (XinS U* XonS)

Intuitivement, dans le cas où X est un segment par exemple, on obtient t_3=XonA ∩* XonB lorsque les solides A et B ne se trouvent pas du même côté par rapport au segment XonA ∩* XonB, le vide sinon.

4.2 Evaluation frontière

La section 4.1 a été consacrée à l'explication de la méthode de classification. Nous verrons dans cette section

comment, grâce à cette méthode, nous allons pouvoir générer la représentation frontière d'un solide à partir de sa représentation CSG. Il est à noter qu'il s'agit d'une conversion exacte entre les deux représentations et non d'une conversion approchée. Nous nous restreindrons dans cette section aux solides à deux dimensions puisque seuls ceux-ci nous intéressent dans notre système. Des résultats similaires existent pour les solides en trois dimensions et peuvent être trouvés dans [Requicha et Tilove 1984].

Le schéma de représentation CSG que nous avons choisi dans notre système a déjà été décrit dans la section 2.6. Pour rappel, nous avons choisi comme primitives de base les rectangles, les triangles et les cercles et, comme opérateurs ensemblistes, les opérateurs régularisés d'union, d'intersection et de différence.

La représentation frontière d'un solide consiste, quant à elle, en la simple énumération (ordonnée) des côtés de ce solide (qui sont des segments linéaires ou des segments circulaires). La représentation frontière d'un objet sera implémentée sous forme de liste chaînée.

La première étape de cette démarche est de générer un ensemble suffisant de côtés d'essai. En d'autre mots, il faut se constituer un ensemble de côtés dont on est sûr qu'il contienne au moins tous les côtés de la frontière du solide.

De façon plus formelle, si δS dénote la frontière du solide S, on dira qu'un ensemble C de côtés d'essai C_i est suffisant ssi δS est inclus dans l'union des C_i.

L'algorithme opérant la conversion nécessite donc au départ un ensemble de côtés qui couvre au moins la frontière du solide et écarte au fur et à mesure les portions de côtés qui ne sont pas sur la frontière.

Cet ensemble de côtés peut être généré en utilisant le fait que $\delta(A \langle op \rangle B)$ est inclus dans $(\delta A \cup \delta B)$, ce qui peut d'ailleurs être démontré de façon mathématique. Cela signifie

que si S est un solide non primitif, c'est-à-dire S = A <op> B, alors, l'union des frontières de A et de B contient au moins la frontière de S.

Dans le cas d'un solide S représenté selon le schéma CSG, on peut appliquer cette formule de façon itérée jusqu'à ce qu'on arrive aux primitives feuilles de l'arbre CSG. Ainsi, on peut donner comme exemples d'ensembles suffisants de côtés, les ensembles suivants :

- l'ensemble des côtés des primitives feuilles de l'arbre CSG correspondant au solide;
- l'ensemble des segments correspondant aux demi-espaces primitifs sous-jacents aux primitives de l'arbre CSG.

Une fois cet ensemble C de côtés généré, la deuxième étape de la démarche consiste à écarter les parties des côtés qui n'appartiennent pas à la frontière de solide. Pour ce faire, on utilise la méthode de classification vue précédemment. On classifie chacun des côtés C_i de C par rapport au solide S et on ne garde de ces côtés que les parties qui sont "on" (C_i onS).

Comme nous l'avons vu, la méthode de classification opère par décomposition du problème en sous-problèmes et par combinaison des résultats obtenus. Il reste à définir quelles sont les procédures primitives à mettre en oeuvre et comment effectuer la combinaison des résultats.

La combinaison des résultats se fera au moyen des formules récursives de classification vues dans la section 4.1.5. La procédure "combine" aura principalement besoin de procédures mettant en oeuvre les opérations régularisées d'union, d'intersection et de différence de segments linéaires ou circulaires.

Les procédures primitives consistent à classifier les différents types de côtés possibles par rapport aux différents demi-espaces primitifs sous-jacents aux solides primitifs. Les procédures primitives à concevoir dans notre système sont donc les procédures de classification d'un segment linéaire ou

d'un segment circulaire par rapport à un demi-espace plan ou à une portion de disque.

Bien que la méthode de classification ait réduit de façon conséquente la complexité du problème d'évaluation de la frontière d'un solide, on ne peut pour autant qualifier ces procédures primitives de triviales. Elles nécessitent la manipulation d'équations de droites et de cercles et se basent sur des algorithmes dont il est difficile de montrer la correction. La principale cause de cette complexité réside dans le choix que nous avons fait d'adopter les portions de disques comme demi-espaces primitifs.

4.3 Le sous-système de conversion

Nous expliquerons dans cette section, sans entrer dans les détails, les différentes étapes de notre sous-système de conversion ainsi que la façon dont l'implémentation a été réalisée.

4.3.1 Les différentes étapes

La première étape consiste à transformer l'arbre CSG résultant de la compilation (dont les feuilles sont, pour rappel, des solides primitifs). En effet, la classification d'un segment par rapport à un solide primitif est une opération d'un niveau trop élevé et donc trop complexe. La transformation consiste donc à remplacer, dans l'arbre CSG, les solides primitifs (feuilles) par les demi-espaces qui les engendrent. Par exemple, un rectangle sera remplacé par l'intersection régularisée de quatre demi-epaces plans. Ainsi, les procédures primitives de classification consistent, entre autres, à classifier un segment par rapport à un demi-espace plan, ce qui est déjà un problème plus facile.

La deuxième étape du sous-système de conversion consiste à faire absorber les opérateurs de mouvement rigide par les demi-espaces primitifs feuilles de l'arbre CSG résultant de la

première étape. Pour cela, il suffit d'enlever chaque noeud correspondant à un opérateur de mouvement en appliquant cet opérateur à tous les demi-espaces primitifs qui sont descendants de ce noeud.

La troisième étape consiste à collecter un ensemble suffisant de côtés candidats. Comme on l'a vu, l'ensemble des côtés des solides primitifs est un ensemble suffisant. On obtient donc ces côtés par un simple parcours des feuilles de l'arbre CSG obtenu à l'étape 2.

La quatrième étape est l'étape de classification proprement dite. Chacun des côtés collectés est classifié par rapport au solide représenté par l'arbre. On ne garde de chaque côté que la partie qui se trouve sur le solide (XonS).

L'ensemble des côtés obtenus constitue la frontière du solide. Toutefois, ces côtés peuvent ne pas être disjoints dans le sens où ils peuvent se recouvrir. Une cinquième étape devra donc être envisagée pour réduire certains segments en vue d'éliminer les parties qui se recouvrent.

Finalement, la générateur de données a besoin d'avoir les côtés de la frontière dans un certain ordre (ordre de parcours de la frontière). Une sixième étape est donc nécessaire pour remettre en ordre les côtés résultant de l'étape cinq.

On a déjà dit qu'un solide (aussi appelé domaine) est composé de une ou plusieurs sous-régions (aussi appelées sous-domaines). En fait, il y a autant d'arbres CSG résultant de la compilation qu'il y a de sous-domaines. Les six étapes décrites ci-dessus seront répétées pour chacun des sous-domaines.

On a vu au chapitre 2 qu'un arbre CSG correspond à une représentation valide d'un solide si les solides primitifs feuilles sont instanciés correctement. Cette vérification se faisant à la compilation, l'arbre CSG qui en résulte est donc valide. Toutefois, on a admis que les solides valides pouvaient être composés de plusieurs morceaux non connexes.

Or, le programme de calcul par éléments frontières nécessite que le domaine soit composé d'un seul morceau c'est-à-dire que les sous-domaines qui le composent soient disjoints deux à deux et connexes.

Les sous-domaines résultant de la compilation peuvent ne pas vérifier ces exigences. Des vérifications restent donc à faire. Toutefois, nous laisserons le soin à l'utilisateur de vérifier visuellement ces conditions puisque, de toute façon, il doit vérifier que le solide qu'il a décrit correspond bien au solide qu'il voulait créer.

4.3.2 Implémentation

Un segment linéaire sera représenté par ses deux extrémités. Ces segments seront, par convention, orientés de telle sorte que l'intérieur du solide dont ils font partie se trouve à leur gauche.

Un segment circulaire sera représenté par ses deux extrémités et par le centre du cercle générateur, avec comme convention que la première extrémité se trouve, dans le sens contraire des aiguilles d'une montre, avant la deuxième. Une information supplémentaire spécifiera de quel côté d'un segment circulaire se trouve l'intérieur du solide.

Les conventions adoptées dans la représentation des segments linéaires et circulaires sont telles qu'aucune information supplémentaire relative au voisinage n'est nécessaire pour résoudre les problèmes d'ambiguïté on/on.

Le reste de cette section est consacré à une énumération des différentes procédures mises en oeuvre dans notre système ainsi qu'à une brève description de chacune d'entre elles. Le lecteur non intéressé peut passer directement à la section suivante.

Etape 1

L'étape 1 nécessite des procédures qui remplacent dans les arbres CSG venant de la compilation les solides primitifs par les demi-epaces sous-jacents. Ainsi, on aura :

- une procédure qui remplace un rectangle par l'intersection régularisée de 4 demi-espaces plans;
- une procédure qui remplace un triangle par l'intersection régularisée de 3 demi-espaces plans;
- une procédure qui remplace un cercle par une portion de disque (qui est en fait un disque complet).

Etape2

L'étape 2 nécessite :

- une procédure qui parcourt l'arbre afin d'éliminer les noeuds correspondant aux opérateurs de mouvement rigide en les appliquant directement aux demi-espaces qui descendent de ces noeuds;
- une procédure effectuant la translation d'un demi-espace plan et d'une portion de disque;
- une procédure effectuant la rotation (d'un angle donné par rapport à un point donné) d'un demi-espace plan et d'une portion de disque;
- une procédure effectuant la symétrie (par rapport à un axe donné) d'un demi-espace plan et d'une portion de disque.

Etape 3

L'étape 3 nécessite :

- une procédure collectant tous les segments linéaires ou circulaires feuilles d'un arbre CSG.

Etape 4

L'étape de classification est l'étape qui nécessite le plus de procédures et les plus difficiles à mettre en oeuvre. Ainsi, on aura :

- des procédures effectuant l'union, l'intersection et la différence régularisée de 2 segments linéaires. Ces procédures font appel aux équations paramétriques des segments et utilisent un algorithme de tri;
- des procédures effectuant l'union, l'intersection et la différence régularisée de 2 segments circulaires. Ces

procédures se sont révélées plus difficiles à réaliser que dans le cas linéaire;

- des procédures effectuant l'union, l'intersection et la différence régularisée de deux listes de segments linéaires et circulaires. Celles-ci utilisent les procédures précédentes;
- une procédure de classification d'un point par rapport à un demi-espace plan. Elle consiste principalement à remplacer les coordonnées du point à classifier dans l'équation du demi-espace plan. Selon le signe de la valeur obtenue, on peut déterminer si le point se trouve in, on ou out;
- une procédure de classification d'un segment linéaire par rapport à un demi-espace plan. On calcule d'abord l'équation du segment linéaire et celle du demi-espace plan. A partir de ces deux équations, on détermine si les deux droites sont confondues, parallèles ou sécantes. Dans le cas où elles sont confondues, le résultat est immédiat (on). Dans le cas où elles sont parallèles, on classifie une des deux extrémités du segment de façon à voir de quel côté il se situe par rapport au demi-espace plan. Dans le cas sécant, on calcule le point d'intersection et on détermine quel est le morceau qui est in et quel est le morceau qui est out;
- une procédure de classification d'un segment circulaire par rapport à un demi-espace plan. On calcule d'abord l'équation du segment circulaire et celle du demi-espace plan. Par résolution du système composé de ces deux équations, on calcule le nombre de points d'intersection réels ainsi que leurs coordonnées. Ensuite, on détermine, par une série de tests, les parties in, on et out;
- une procédure de classification d'un point par rapport à une portion de disque;
- une procédure de classification d'un segment linéaire par rapport à une portion de disque;
- une procédure de classification d'un segment circulaire par rapport à une portion de disque.

Ces six dernières procédures sont en fait les procédures primitives de classification prim-M*[X,S], tandis que les premières interviennent lors de la combinaison des résultats. Pour la construction des procédures primitives de

classification, nous avons davantage procédé par tâtonnements que selon une méthodologie bien particulière. Le procédé a consisté à relever les différentes cas de figures possibles et à voir quelles conditions devaient être vérifiées pour qu'on se trouve dans chacun de ces cas. Un tel procédé rend difficile la démonstration formelle des algorithmes implémentés.

Etape 5

Cette étape nécessite les procédures suivantes :

- une procédure déterminant si deux segments possèdent une partie commune c'est-à-dire déterminant s'ils sont disjoints ou non.
- une procédure qui élimine les parties redondantes des segments de la frontière du solide.

Etape 6

On aura pour cette étape les procédures suivantes :

- une procédure de recherche, parmi une liste de segments, d'un segment dont la première extrémité correspond à un point donné.
- une procédure mettant en ordre les segments de la frontière du solide.

4.4 Optimisations possibles

Cette section vise à expliquer comment il est possible d'améliorer l'efficacité de l'algorithme récursif vu dans la section 4.1.4 et donc les performances générales ou moyennes d'un système basé sur un tel algorithme. L'optimisation peut se faire en ayant recours à des informations supplémentaires qui peuvent être de deux types : intrinsèques et extrinsèques.

Les informations intrinsèques sont des informations spécifiques au domaine d'application. Ces informations sont donc connues lors de la conception du système. L'exploitation de ces informations n'engendre aucun coût supplémentaire d'un point de vue calcul au sacrifice d'une perte de généralité des algorithmes. En tenant compte, par exemple, de certaines

propriétés des solides primitifs (notamment la convexité), on peut être amené à simplifier les formules récursives vues dans la section 4.1.5.

Les informations extrinsèques sont, quant à elles, des informations indépendantes du domaine d'application. Ces informations sont acquises dynamiquement via des tests de haut niveau effectués lors de la classification. L'exploitation de ces informations n'engendre aucune perte de généralité mais peut provoquer, lorsque les cas particuliers que l'on teste se présentent rarement, une augmentation sensible du coût calcul. De telles informations peuvent, par exemple, servir à éviter certaines combinaisons inutiles dans l'algorithme récursif de la section 4.1.4.

Cette technique d'exploitation dynamique d'informations vise à améliorer les performances moyennes des algorithmes mais peut, dans le pire des cas, dégrader leurs performances. Il est à noter que la façon dont les performances seront améliorées dépend étroitement du type de problème traité dans l'application.

Donnons, comme exemple de technique d'utilisation d'informations acquises à l'exécution, une technique utilisée dans le système PADL (présenté dans [Vloecker 1978]) qui s'est par ailleurs avérée très efficace :

La technique consiste à court-circuiter la procédure "combine" de l'algorithme récursif de la section 4.1.4 en utilisant certaines informations. Lorsqu'on calcule M*[X,S] avec S = A <op> B, les valeurs de M*[X,A] et de M*[X,B] sont déjà connues. Supposons que X est complètement hors de A c'est-à-dire M*[X,A]=(XinA=∅,XonA=∅,XoutA=X). Alors, dans ce cas, il est possible, dans les formules de classification, de tirer la valeur de M*[X,S] sans avoir recours aux procédures "combine" souvent coûteuses en calcul. Ainsi, les simplifications suivantes peuvent êtres obtenues :

Lorsque M*[X,A] = (XinA=∅,XonA=∅,XoutA=X) alors
si S = A ∩* B alors M*[X,S] = (XinS=∅,XonS=∅,XoutS=X)

si S = A U* B alors M*[X,S] = M*[X,B]

si S = A * B alors M*[X,S] = (XinS=∅,XonS=∅,XoutS=X)

Ces simplifications découlent directement des formules de la section 4.1.5 lorsqu'on prend XinA=∅ et XonA=∅. Des simplification similaires peuvent être obtenues en testant M*[X,B] au lieu de M*[X,A].

De façon générale, des formules de ce genre peuvent être définies lorsque l'ensemble candidat X reste, après classification, non segmenté vis-à-vis du sous-arbre gauche ou du sous-arbre droit c'est-à-dire lorsque deux des trois valeurs de M*[X,A] ou de M*[X,B] sont nulles.

Il est à remarquer que, dans notre système, aucune optimisation de ce genre n'a été mise en oeuvre principalement parce que nous n'en avons pas eu le temps (et nous le regrettons). Une optimisation basée sur l'exploitation dynamique d'informations du genre de celles présentées ci-dessus aurait tout à fait été possible et aurait eu comme conséquence une nette amélioration des performances (par exemple un gain considérable en temps calcul au niveau du changement de représentation) étant donné que des cas particuliers se présentent fréquemment (ensemble candidat X non segmenté vis-à-vis de A ou de B). Une optimisation intrinsèque c'est-à-dire dépendante du domaine est, en première analyse, moins évidente à trouver dans le cadre de notre système ...

4.5 Remarques et conclusion

On peut rappeler qu'un des grands atouts de la démarche suivie tout au long de ce chapitre est qu'elle repose sur des fondements mathématiques bien établis. Comme on l'a vu au chapitre 2, les représentations de solides ont été conçues après que les entités mathématiques abstraites et leurs propriétés aient été bien définies. De la même façon, l'algorithme récursif est basé sur des formules récursives qui

peuvent être entièrement démontrées de façon mathématique (en se basant sur les propriétés de régularité et de voisinage).

On peut également dire que la fonction de classification M* et l'algorithme récursif de la section 4.1.4 se sont avérés être des outils conceptuels très puissants garantissant la correction, permettant de traiter les cas dits pathologiques et débouchant sur un code relativement modulaire.

De plus, l'algorithme confère à notre système la caractéristique hautement souhaitable d'extensibilité. En effet, tout ce qui est dépendant du domaine d'application a été localisé dans les procédures de bas niveau (prim-M*[X,S]). Ainsi, l'ajout d'un nouveau demi-espace, par exemple, se traduirait par l'ajout à notre système de nouvelles procédures primitives de classification. On peut également dire que toutes les procédures de bas niveau peuvent servir de base à une éventuelle extension à la troisième dimension.

Chapitre 5
Le générateur de données

5.0 Introduction

Ce chapitre est consacré à l'étude du sous-système de génération de données. Une fois le texte source compilé et le changement de représentation effectué, nous avons à notre disposition toutes les informations nécessaires à la génération des données destinées au logiciel BEM de calcul par éléments frontières, à savoir, la représentation frontière de la pièce ainsi qu'une série de tables relatives aux sollicitations que doit subir celle-ci.

Le sous-système de génération de données ne met en oeuvre aucune technique bien particulière. Il est composé de deux parties distinctes, à savoir, une partie effectuant un certain nombre de vérifications, et une partie effectuant la génération des données proprement dite. C'est à l'étude de chacune de ces deux parties que sera consacré ce chapitre.

5.1 Les vérifications

Nous passerons en revue, dans cette section, les différentes conditions qui doivent être vérifiées avant que la génération des données ne puisse avoir lieu. Il est à remarquer que ces vérifications ne peuvent être faites à partir du langage puisqu'elles nécessitent une représentation frontière de la pièce.

Rappelons qu'une pièce peut être décrite en la partitionnant en plusieurs morceaux (connexes et disjoints). La pièce est appelée domaine alors que les morceaux qui la composent sont appelés sous-domaines. L'usage de sous-domaines permet, entre autres, de générer des noeuds à l'intérieur de la structure afin d'y calculer certaines grandeurs inconnues.

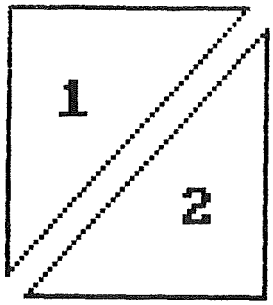
On vérifie en premier lieu que les sous-domaines sont composés d'un seul morceau. Etant donné que les sous-domaines ont été initialement décrits de façon constructive, cette vérification peut se faire par un simple parcours des segments de leur frontière.

On vérifie ensuite que les sous-domaines sont connexes entre eux et ce afin que la pièce (le domaine) soit constitué d'un seul morceau (exigence imposée par le programme BEM de calcul par éléments frontières). En d'autres mots, il faut vérifier que chaque sous-domaine contient au moins un segment (linéaire ou circulaire), parmi la liste des segments qui constituent sa frontière, dont une partie est commune avec un segment quelconque d'un des autres sous-domaines. Vérifier qu'un segment possède une partie commune avec un autre segment peut se faire en vérifiant que les deux segments possèdent une intersection régularisée non nulle. (cfr fig 5.1)

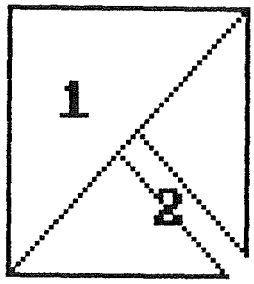
La vérification qui consiste à voir si les sous-domaines sont disjoints étant trop coûteuse à mettre en oeuvre par programme (il faudrait voir s'ils possèdent une intersection régularisée nulle en utilisant la méthode de classification), on laissera le soin à l'utilisateur de faire cette vérification visuellement.

Viennent ensuite des vérifications relatives aux sollicitations que l'utilisateur veut faire subir au solide, sollicitations décrites dans le texte source et dont les informations se retrouvent dans une série de tables résultant de la compilation. Les vérifications à faire sont les suivantes :

- toute sollicitation ponctuelle (c'est-à-dire localisée en un endroit ponctuel du solide) doit être imposée sur la frontière du solide. Il faut donc vérifier que toute sollicitation ponctuelle se trouve sur au moins un segment de la frontière d'un sous-domaine et que ce segment n'ait pas une partie commune avec un autre sous-domaine (auquel cas la sollicitation serait imposée à l'intérieur du solide, ce qui n'est pas possible);

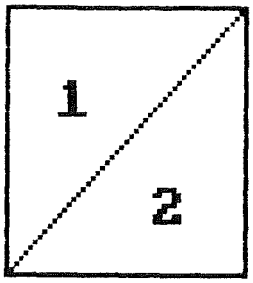


Les deux sous-domaines ne sont pas connexes



Le sous-domaine 2 n'est pas constitué d'un seul morceau

Cas correct

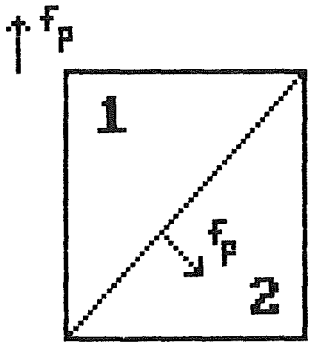


Les deux sous-domaines sont connexes et constitués d'un seul morceau

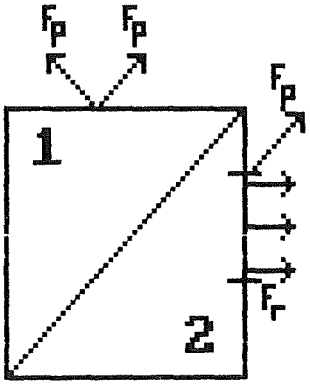
Fig 5.1 Verifications relatives aux sous-domaines

F_p = force ponctuelle

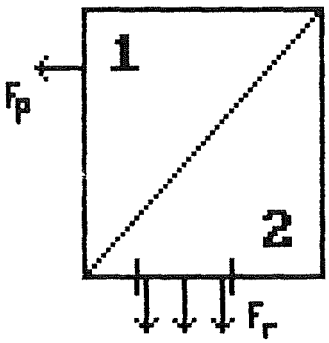
F_r = force repartie



Forces imposées en dehors de la frontière du domaine



Forces imposées au même endroit ou bien forces qui se recouvrent



Forces correctement imposées

Fig 5.2 Verifications relatives aux sollicitations

- toute sollicitation répartie (c'est-à-dire distribuée sur un segment de la frontière du solide) doit être imposée sur la frontière du solide. Des vérifications du même genre que dans le cas ponctuel doivent donc être mises en oeuvre;

- deux sollicitations différentes ne peuvent être imposées au même endroit ou bien se recouvrir partiellement. En d'autres mots, une sollicitation ponctuelle ne peut être imposée là où une sollicitation ponctuelle ou répartie a déjà été imposée. Deux sollicitations réparties ne peuvent se recouvrir, même partiellement. (cfr fig 5.2)

5.2 La génération

Une fois ces vérifications effectuées, la génération du fichier de données destiné au programme de calcul par éléments frontières peut avoir lieu. La difficulté principale de la génération réside dans la génération des noeuds et des éléments sur la frontière ainsi que la génération des sollicitations. Le reste de la phase de génération consiste en un simple recopiage d'informations délivrées par le compilateur.

Génération des noeuds et des éléments

Les noeuds et les éléments sont générés sur la frontière de chacun des sous-domaines. L'usage des sous-domaines permet donc de générer des noeuds à l'intérieur de la structure, une partie de la frontière de chaque sous-domaine étant située à l'intérieur de la structure (lorsqu'il y a deux sous-domaines au moins). En procédant de cette façon, il est possible de calculer des grandeurs inconnues (forces, déplacements) à l'intérieur de la structure.

La spécification de la génération des noeuds et des éléments sur la frontière des sous-domaines est la suivante :

- il faut générer, pour l'ensemble des sous-domaines, un nombre de noeuds aussi proche que possible du nombre qui a été spécifié dans le fichier source (Number Of Nodes);
- afin que le programme BEM de calcul par éléments frontières obtienne les meilleurs résultats, les noeuds et les éléments doivent être générés de façon uniforme sur la frontière de chaque sous-domaine;
- dans le fichier source, un poids est associé à chacun des sous-domaines afin de permettre à l'utilisateur de mettre plus de noeuds et d'éléments sur certains sous-domaines que sur d'autres. La façon dont les poids entrent en ligne de compte pour générer les noeuds et les éléments n'est pas spécifiée;
- etant donné que toute sollicitation doit être associée à un noeud, un noeud doit être généré partout où une sollicitation ponctuelle est imposée. Afin de ne pas dégrader les résultats donnés par le programme BEM de calcul par éléments frontières, aucun noeud ne doit être généré dans un voisinage proche de celui-ci;
- de la même façon, deux noeuds doivent être générés aux extrémités d'une sollicitation répartie;
- un noeud double (c'est-à-dire deux noeuds distincts situés au même endroit (ayant mêmes coordonnées)) doit être généré lorsqu'il y a un changement de direction de la normale à la frontière du solide (par exemple au point d'intersection de deux segments);
- un noeud double doit également être généré lorsqu'un changement dans les sollicitations se produit. Par exemple, à chaque extrémité d'une sollicitation répartie, un noeud double est généré (cfr fig 5.3);
- pour un segment commun à deux sous-domaines, des noeuds doubles sont générés sur tout le segment.

Les noeuds doivent être numérotés dans un certain ordre (celui du parcours de la frontière).

Un élément linéaire est un segment linéaire dont les deux extrémités sont des noeuds. Un élément circulaire est un segment circulaire défini, dans le programme BEM, par trois noeuds dont deux aux extrémités et un au milieu du segment.

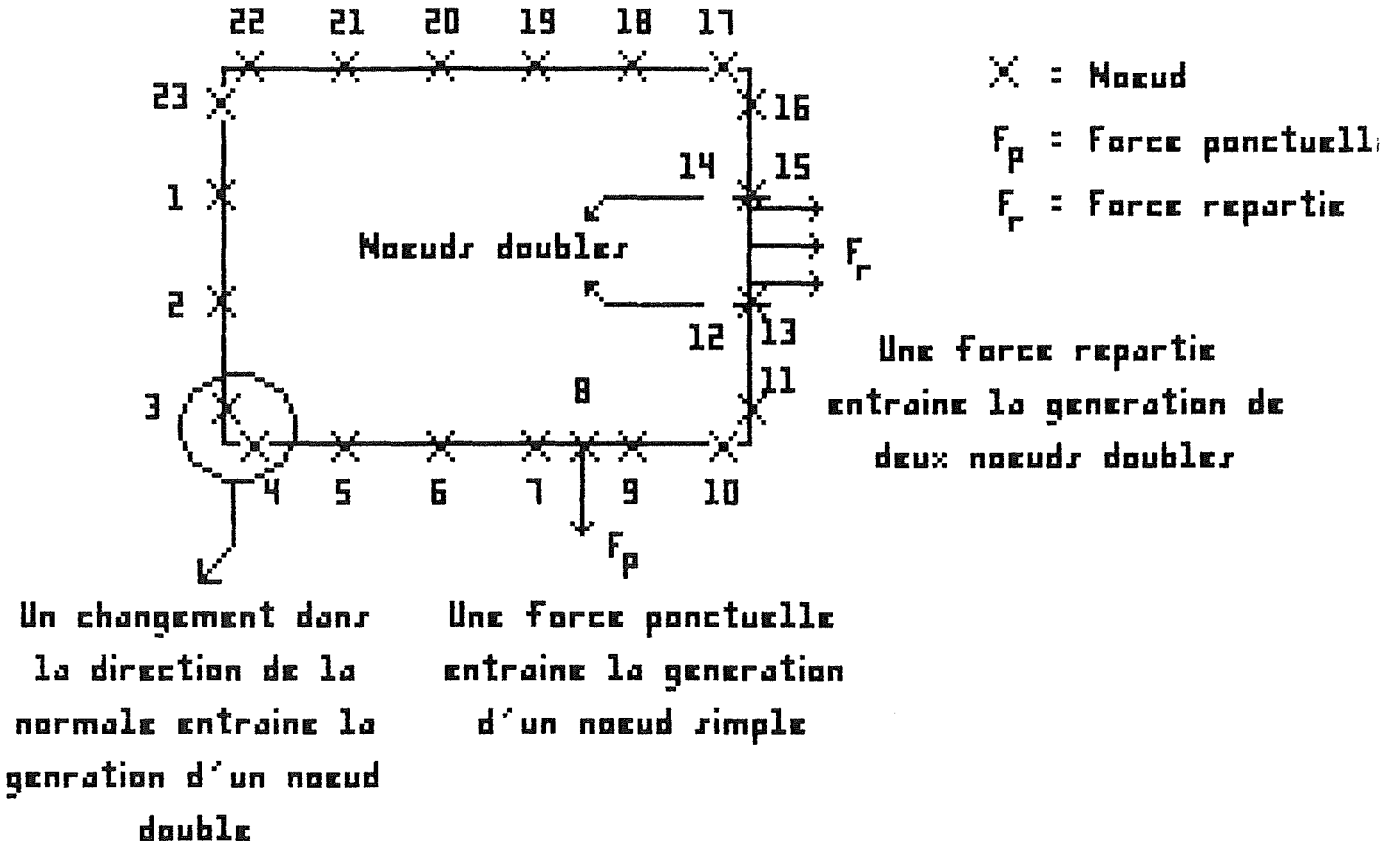


Fig 5.3 Generation des noeuds

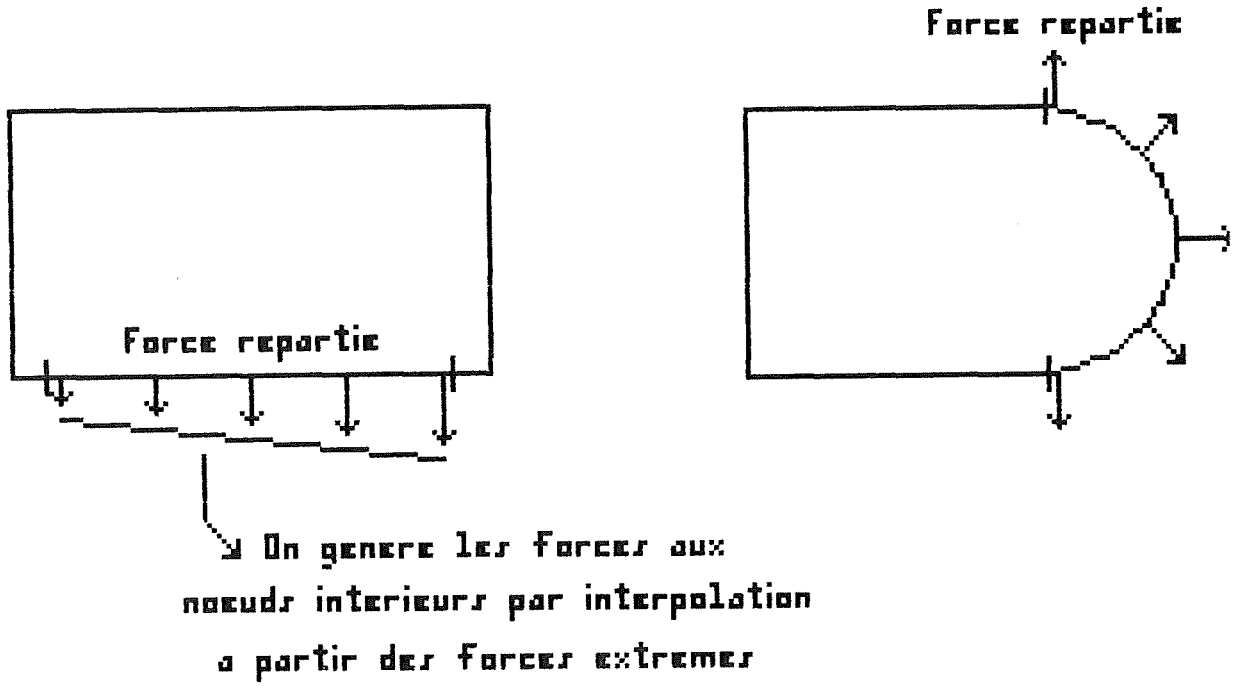


Fig 5.4 Generation des sollicitations par interpolation

Un élément circulaire ne peut être un cercle complet. Les éléments sont numérotés dans l'ordre de parcours de la frontière.

On peut voir que ces règles de génération ne sont pas d'une précision exemplaire et laissent certaines ambiguïtés difficiles à résoudre dans certains cas de figure. Par exemple, lorsqu'une sollicitation ponctuelle est imposée à l'intersection de deux segments où un noeud double (c'est-à-dire deux noeuds de mêmes coordonnées) est généré, on ne peut déterminer au quel de ces deux noeuds doit être associée la sollicitation.

Le nombre total de noeuds à générer pour tous les sous-domaines a été spécifié au moyen de notre langage dans le fichier source. Toutefois, il se peut que le nombre de noeuds réellement générés ne soit pas exactement le même. En effet, tout segment linéaire entraîne la génération minimum de deux noeuds aux extrémités du segment et tout segment circulaire la génération minimum de trois noeuds. Ainsi, il se peut que le nombre de noeuds spécifié soit trop petit (auquel cas on générera le plus petit nombre possible de noeuds) ou soit impossible à générer exactement (auquel cas on générera un nombre de noeuds avoisinant celui spécifié). Dans les deux cas, un message d'avertissement est produit afin de prévenir l'utilisateur que le nombre de noeuds générés diffère du nombre de noeuds spécifié.

La génération des noeuds est essentiellement effectuée de la façon suivante :

- le nombre total de noeuds à générer est réparti entre les différents sous-domaines de façon proportionnelle à la longueur de chacun de ceux-ci. On a donc la formule suivante :

  Nb noeuds ssdom = (Nb tot noeuds * long frontière ssdom)
  / Σ longueur frontière ssdom

- le nombre de noeuds pour chaque sous-domaine est ensuite multiplié par le poids qui lui est associé afin de forcer le générateur à mettre plus de noeuds sur certains sous-domaines que sur d'autres;

- le nombre de noeuds à générer sur un segment se fait proportionnellement à la longueur du segment par rapport à la longueur du sous-domaine. On a donc :

  Nb noeuds seg = (nb noeuds ssdom * lg seg) / lg ssdom

  Si ce nombre n'est pas suffisant (dans le cas par exemple où le segment est très petit), on le fixe au minimum requis (2 dans le cas linéaire, 3 dans le cas circulaire);
- sur un segment commun à deux sous-domaines, on génère le plus grand nombre de noeuds des deux nombres calculés vis-à-vis de chaque sous-domaine. Les noeuds générés sur ces segments sont des noeuds doubles;
- lorsqu'un noeud associé à une sollicitation ponctuelle se trouve trop près de l'extrémité d'un segment, celui-ci est automatiquement rejeté vers l'intérieur du segment (afin de respecter le fait qu'aucun noeud ne doit se trouver dans un proche voisinage);
- il faut également tenir compte de toutes les règles de génération énoncées ci-dessus.

Génération des sollicitations

En chacun des noeuds générés, il faut spécifier quel type de sollicitation lui est associé ainsi que les valeurs imposées.

Les différents types de sollicitations sont les suivants :
- force imposée selon X et Y;
- déplacement imposé selon X et Y;
- force imposée selon X et déplacement imposé selon Y;
- force imposée selon Y et déplacement imposé selon X.

Les deux valeurs non fixées sur les quatres représentent à chaque fois les inconnues du problème. Ainsi, lorsqu'on impose en un point, par exemple, une force selon les deux axes X et Y, les inconnues du problème en ce point sont les déplacements de ce point selon X et Y.

Par défaut, un noeud où aucune sollicitation n'a été spécifiée dans le texte source subira une sollicitation du

premier type avec zéro comme valeurs imposées (force nulle imposée selon les deux axes).

Une sollicitation peut être exprimée soit dans le système d'axe XY, soit dans un système d'axe Normal/Tangent centré à l'endroit où est imposée la sollicitation. Le système d'axe Normal/Tangent n'étant disponible qu'au moment où on dispose d'une représentation frontière de la pièce, la conversion d'un vecteur exprimé dans le système d'axe NT en son vecteur équivalent dans le système d'axe XY n'a pu être effectuée à la compilation et doit donc être réalisée dans ce module.

Pour une sollicitation répartie sur un morceau de la frontière, seules les valeurs imposées aux extrémités de ce morceau ont été spécifiées. Il faut donc générer par interpolation linéaire les valeurs de sollicitation pour les noeuds générés à l'intérieur de ce morceau de frontière (cfr fig 5.4). Lorsqu'une sollicitation répartie se trouve sur un segment linéaire, l'interpolation linéaire se fera dans le système d'axe XY. Dans le cas circulaire, elle se fera dans le système d'axe NT.

Génération des caractéristiques de la pièce

Les caractéristiques propres à la pièce sont également générées dans le fichier de données. Ces informations peuvent être trouvées dans les tables fournies lors de la compilation. Il s'agit donc d'un simple recopiage d'informations.

Le lecteur intéressé pourra trouver en annexe un exemple de fichier tel qu'il est généré par notre système.

Chapitre 6
Le module d'affichage

6.0 Introduction

Le module d'affichage permet la visualisation des pièces définies dans le langage dont l'affichage a été explicitement demandé via le statement "display". On peut afficher

- soit la totalité de la pièce (domaine) via l'ordre d'affichage :

  display (nom-de-domaine)

- soit certaines sous-régions de celle-ci (sous-domaines) via une suite d'ordres d'affichage de la forme :

  display (nom-de-ssdomaine)

Ce chapitre explique la raison d'être de ce module ainsi que quelques détails d'implémentation.

6.1 Justification de l'existence de ce module

L'existence du module d'affichage peut se justifier par les raisons suivantes :

- la pièce décrite par l'utilisateur via notre langage peut très bien être valide mais ne pas être correcte (cfr chapitre 2). L'affichage de la pièce permet donc à l'utilisateur de vérifier visuellement si elle correspond à ce qu'il voulait réellement créer;

- nous avons vu au chapitre 2 que, lorsqu'on utilise le schéma CSG comme schéma de représentation, toute représentation syntaxiquement correcte est valide. En d'autres termes, toute représentation résultant de la compilation est valide. Cependant, nous avons vu dans la section 4.3 que, même lorsque la représentation est valide, certaines conditions (sous-domaines disjoints et connexes) peuvent ne pas être vérifiées. Comme nous l'avons dit, nous avons décidé de

donner à l'utilisateur la responsabilité de vérifier ces conditions visuellement.

6.2 Implémentation

L'affichage des pièces à l'écran ne peut se faire que si l'on dispose d'une représentation frontière de celles-ci. Le module d'affichage ne peut donc être activé avant qu'on ne soit passé de la représentation CSG à la représentation frontière via le module de conversion (transformateur).

Puisque la frontière d'une pièce ne peut être composée que de segments linéaires et circulaires, les seules grandes procédures à mettre en oeuvre sont les suivantes :

- affichage d'un segment de droite dont on connait les deux extrémités.
- affichage d'un arc de cercle dont on connait le centre et les deux extrémités.

L'affichage d'une pièce consiste simplement à afficher à tour de rôle tous les segments qui constituent sa frontière au moyen des deux procédures triviales citées ci-dessus.

Pour des raisons de facilité, aucun paramètre d'affichage n'a été prévu dans le langage avec l'ordre d'affichage d'une pièce. Une pièce pouvant avoir à priori des dimensions quelconques, il se pourrait que celle-ci soit affichée en dehors de l'écran. On a donc prévu un mécanisme permettant de transformer de façon automatique les coordonnées de la pièce de telle manière que la pièce puisse tenir entièrement sur l'écran.

Le module d'affichage a été conçu et implémenté de façon à être portable. Notre système étant amené à être utilisé sur des terminaux graphiques Tektronix, notre sous-système d'affichage a été élaboré en suivant la norme Tektronix. Ainsi, il se compose de deux couches distinctes :

- une première couche offrant des procédures permettant de réaliser des fonctionnalités de bas niveau comme, par exemple, l'effacement de l'écran, la définition du mode de résolution, l'affichage d'un point, le déplacement du curseur en un point, etc ... Les dépendances vis-à-vis de la norme Tektronix sont localisées et cachées dans ces procédures de bas niveau;

- une deuxième couche (au dessus de la première) offrant des procédures d'un niveau plus élevé comme, par exemple, l'affichage d'une droite, d'un arc de cercle, d'un cadre, du système d'axe, etc ... Ces procédures sont réalisées en utilisant les procédures de la couche inférieure.

Grâce à cette décomposition en deux couches, notre système s'avère être relativement portable. En effet, si un changement de norme doit être effectué suite au passage sur des terminaux d'un autre type, ce changement sera presque entièrement répercuté sur la couche inférieure. Il suffit, par exemple, de remplacer cette couche par une autre offrant les même primitives mais respectant une autre norme, sans toucher à la couche supérieure.

Des exemples d'affichages de pièces peuvent être trouvés en annexes.

Conclusion

Grâce au système de génération de données que nous avons conçu et réalisé, on peut s'attendre à ce que l'ingénieur soit grandement facilité dans la tâche fastidieuse que constitue le maillage de la structure dont il veut étudier le comportement. Grâce à ce maillage automatique, on peut espérer, non seulement un gain de temps considérable, mais également que des structures d'une plus grande complexité puissent être étudiées.

Le fait d'avoir basé le langage du sous-système d'entrée sur un schéma de représentation CSG apporte quelques avantages. Parmi ceux-ci, on peut citer la concision des descriptions de pièces et la facilité de créer des représentations valides. Il semble également que la description de la géométrie d'une pièce de façon constructive est une approche assez naturelle à laquelle l'utilisateur humain s'accomode avec une certaine facilité.

Toutefois, le fait d'avoir basé notre langage sur un tel schéma de représentation a accrû considérablement la complexité de mise en oeuvre de notre système. La principale difficulté émerge du module de conversion opérant le passage d'une représentation CSG d'un solide à une représentation frontière de ce même solide. Bien que la méthode de classification utilisée dans ce module soit relativement bien établie dans la littérature, nous avons dû faire preuve de beaucoup de créativité quant à la conception des primitives (non triviales) de bas niveau.

Lors de la réalisation de ce projet, nous avons également remarqué combien il est parfois difficile de communiquer avec des scientifiques appartenant à une autre discipline que la nôtre, ceux-ci s'exprimant parfois en des termes techniques difficilement compréhensibles. De la même façon, nous avons éprouvé certaines difficultés à rentrer dans des considérations relatives à une discipline qui nous est étrangère. C'est ainsi que, par exemple, la compréhension de la méthode des éléments finis nous a posé certains problèmes.

Outre cela, nous pouvons nous réjouir d'avoir traité des problèmes intéressants de par leur nature et les techniques mises en oeuvre pour les résoudre. Nous pouvons également nous enorgueillir d'avoir conçu et mis en oeuvre un sytème d'une relative complexité dont le fonctionnement s'avérera être, nous l'espérons, relativement satisfaisant.

Bibliographie

Méthode des éléments finis

[Massonet 1972] Ch. Massonnet, G. Deprez, R. Maquoi, R. Muller et G. Fonder, "Calcul des structures sur ordinateur, tome1 : analyse matricielle des structures", Eyrolles éditeur, Masson et Cie éditeurs, 1972.

[Dhatt et Touzot 1981] Dhatt G. et Touzot G., "Une présentation de la méthode des éléments finis", Malouine S.A., Paris 1981.

[de Saxcé 1987] de Saxcé Géry, "Le projet CHARLY, un logiciel de calcul par éléments finis et éléments frontières de seconde génération, spécifications fonctionnelles et architecture interne", Séminaire de Génie Logiciel, Aide-mémoire et document de réflexion, M.S.M. Université de Liège, 13 juillet 1987.

Langages et techniques de compilation

[Aho, Sethi et Ullman] Aho, Sethi et Ullman, "Compilers principles techniques and tools", Addison Wesley.

[Hunter] R. Hunter, "The design and construction of compilers", Wiley.

[Lewis, Rosen Krantz et Stearns] P.M. Lewis II, D.J. Rosen Krantz et R.E. Stearns, "Compiler design theory", The systems programming series.

[Rayward et Smith] Rayward et Smith, "A first course in formal language theory", Blockwell Scientific Publications.

[Badler et Bajcsy 1978] N. Badler et R. Bajcsy 1978, "Three-dimensional representations for computer graphics and computer vision", ACM Computer Graphics, Vol. 12, n°3, pp. 153-160, August 1978.

[Barton et Buchana 1980] E.E. Barton et I. Buchana 1980, "The polygon package", Computer-Aided Design, Vol. 12, n°1, pp. 3-11, January 1980.

[Braid 1975] I.C. Braid 1975, "The Synthesis of Solids Bounded by Many Faces", Communications of the ACM, Vol. 18, n°4, pp. 209-216, April 1975.

[Eastman et Henrion 1977] C. Eastman et M. Henrion 1977, "GLIDE : a language for design information systems", ACM Computer Graphics, Vol. 11, n°2, pp. 24-33, July 1977.

[Goldstein et Nagel 1971] R.A. Goldstein et R. Nagel, "3-D Visual simulation", Simulation, Vol. 16, n°1, pp. 25-31, 1971.

[Meier 1986] A. Meier 1986, "Applying relational database techniques to solid modelling", Computer Aided Design, vol. 18, n°6, pp. 319-326, July / August 1986.

[Parent 1977] R.E. Parent 1977, "A system for sculting 3-D data", ACM Computer Graphics, Vol. 11, n°2, pp. 138-147, July 1977.

[Requicha 1977] A.A.G. Requicha, "Mathematical models of rigid solid objects", Technical Memorandum N°28, Production Automation Project, University of Rochester, November 1977.

[Requicha 1978] A.A.G. Requicha, "Representation of rigid solid objects", Technical Memorandum N°29, Production Automation Project, University of Rochester, 1978.

[Requicha 1980] A.A.G. Requicha, "Representations for rigid solids : Theory, Methods, and Systems", ACM Computing Surveys, vol. 12, n°4, pp. 437-464, December 1980.

[Requicha et Tilove 1978] A.A.G. Requicha et R.B. Tilove, "Mathematical foundations of constructive solid geometry : General topology of closed regular sets", Technical Memorandum N°27, Production Automation Project, University of Rochester, June 1978.

[Requicha et Tilove 1984] A.A.G. Requicha et R.B. Tilove, "Boolean operations in solid modelling : boundary evaluation and merging algorithms", Technical Memorandum N°26, Production Automation Project, January 1984.

[Requicha et Voelcker 1977] A.A.G. Requicha et H.B. Voelcker, "Constructive Solid Geomtry", Technical Memorandum N°25, Production Automation Project, University of Rochester, November 1977.

[Sutherland et Hodgman 1974] I.E. Sutherland et G.W. Hodgman 1974, "Reentrant Polygon Clipping", Communications of the ACM, Vol. 17, n°1, pp. 32-42, January 1974.

[Tilove 1977] R.B. Tilove 1977, "A study of set membership classification", Technical Memorandum N°30, Production Automation Project, University of Rochester, November 1977.

[Tilove 1980] R.B. Tilove 1980, "Set membership classification : A Unified Approach to Geometric Intersection Problems", IEEE transactions on computers, vol. C-29, n°10, pp. 874-883, October 1980.

[Tilove et Requicha 1980] R.B. Tilove et A.A.G. Requicha, "Closure of Boolean operations on geometric entities", Computer-Aided Design, vol. 12, n°5, pp. 219-220, September 1980.

[Vloecker 1978] H.B. Vloecker 1978, "The PADL-1.0/2 system for defining and displaying solid objects", ACM Computer Graphics, vol. 12, n°3, pp. 257-263, August 1978.

[Weiler et Atherton] K. Weiler et P. Atherton, "Hidden surface removal using polygon area sorting", ACM Computer Graphics, pp. 214-222.

Annexes

Grammaire du langage sous forme BNF

<Domaine> ::= **DOMAIN** <Identificateur de domaine> <Suite de déclarations de ss-domaines> <Suite de déclarations d'ensemble de sollicitations> <déclaration du nombre de noeuds> <Déclaration du nombre de points d'intégration> <Déclaration de gravité> <déclaration de poids volumique> <Déclaration du module d'élasticité> <Déclaration du coefficient de poisson>

<Domaine> ::= **DOMAIN** <Identificateur de domaine> <Suite de déclarations de ss-domaines> <Suite de déclarations d'ensemble de sollicitations> <déclaration du nombre de noeuds> <Déclaration du nombre de points d'intégration> <Déclaration de gravité> <déclaration de poids volumique> <Déclaration du module d'élasticité> <Déclaration du coefficient de poisson> <Suite de display>

<Identificateur de domaine> ::= <Identificateur>

<Suite de déclarations de ss-domaines> ::= <Déclaration de ss-domaine> | <suite de déclarations de ss-domaines> <Déclaration de ss-domaine>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclaration de points> <Affectation au ss-domaine>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclaration de points> <Suite de déclarations d'objets, de constantes ou d'origines> <Affectation au ss-domaine>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclaration de points> <Affectation au ss-domaine> <Déclaration de poids>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclaration de points> <Suite de déclarations d'objets, de constantes ou d'origines> <Affectation au ss-domaine> <Déclaration de poids>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Affectation au ss-domaine>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclarations d'objets, de constantes ou d'origines> <Affectation au ss-domaine>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Affectation au ss-domaine> <Déclaration de poids>

<Déclaration de ss-domaine> ::= **SUBDOMAIN** <Identificateur de ss-domaine> = <Suite de déclarations d'objets, de constantes ou d'origines> <Affectation au ss-domaine> <Déclaration de poids>

<Identificateur de ss-domaine> ::= <Identificateur>

<Suite de déclaration de points> ::= <Déclaration de point> | <Suite de déclaration de points> <Déclaration de point>

<Déclaration de point> ::= <Identificateur de point> = <Coordonnée>

<Identificateur de point> ::= **%** <Identificateur>

<Suite de déclarations d'objets, de constantes ou d'origines> ::= <Déclaration d'objet, de constante ou d'origine> | <Suite de déclarations d'objets, de constantes ou d'origines> <Déclaration d'objet, de constante ou d'origine>

<Déclaration d'objet, de constante ou d'origine> ::= <Déclaration d'objet> | <Déclaration de constante> | <Déclaration d'origine>

<Déclaration d'objet> ::= <Identificateur d'objet> = <Objet>

<Identificateur d'objet> ::= & <Identificateur>

<Objet> ::= <Identificateur d'objet> | <Objet primitif> |
<Objet déplacé> | <Objet combiné>

<Objet primitif> ::= <Cercle> | <Rectangle> | <Triangle>

<Cercle> ::= **CIRCLE** (<Rayon>) | **CIRCLE** (<Rayon>) **AT**
<Point>

<Rayon> ::= <Réel>

<Point> ::= <Identificateur de point> | <Coordonnée>

<Coordonnée> ::= <Coordonnée cartésienne> | <Coordonnée
polaire>

<Coordonnée cartésienne> ::= (<Réel> , <Réel>)

<Coordonnée polaire> ::= **P** (<Rayon> , <Angle>)

<Rectangle> ::= **RECT** (<Longueur> ; <Largeur>) | **RECT** (
<Longueur> ; <Largeur>) **AT** <Point>

<Triangle> ::= **TRIANGLE** (<Point> ; <Point> ; <Point>)

<Objet déplacé> ::= <Objet-rot> | <Objet-trans> | <Objet-sym>

<Objet-rot> ::= **ROT** <Objet> **BY** (<Point> ; <Angle>)

<Objet-trans> ::= **TRANSL** <Objet> **BY** (<Delta-X> ; <Delta-Y>)

<Objet-sym> ::= **SYM** <Objet> **BY** (<Point> ; <Point>)

<Angle> ::= <Réel>

<Delta-X> ::= <Réel>

<Delta-Y> ::= <Réel>

<Longueur> ::= <Réel>

<Largeur> ::= <Réel>

<Objet-combiné> ::= <Union d'objets> | <Intersection d'objets>
| <Différence d'objets>

<Union d'objets> ::= **UNION** <Objet> **WITH** <Objet>

<Intersection d'objets> ::= **INTER** <Objet> **WITH** <Objet>

<Différence d'objets> ::= **DIFF** <Objet> **WITH** <Objet>

<Déclaration de constante> ::= <Identificateur de constante> =
<Constante>

<Déclaration d'origine> ::= **ORIGIN** = <Coordonnée>

<Identificateur de constante> ::= <Identificateur>

<Constante> ::= <Réel>

<Affectation au ss-domaine> ::= <Identificateur de ss domaine>
= <Objet>

<Déclaration de poids> ::= **PONDERATION** = <Réel>

<Suite de déclarations d'ensemble de sollicitations> ::=
<Déclaration d'ensemble de sollicitations> | <Suite de
déclarations d'ensemble de sollicitations> <Déclaration
d'ensemble de sollicitations>

<Déclaration d'ensemble de sollicitations> ::= **SET OF SOLLICITATIONS** <Identificateur d'ensemble de sollicitations> = <Suite de déclarations de points> <Suite de déclarations de sollicitations> | **SET OF SOLLICITATIONS** <Identificateur d'ensemble de sollicitations> = <Suite de déclarations de sollicitations>

<Identificateur d'ensemble de sollicitations> ::= <Identificateur>

<Suite de déclarations de sollicitations> ::= <Déclaration de sollicitation> | <Suite de déclarations de sollicitations> <Déclaration de sollicitation>

<Déclaration de sollicitation> ::= <Déclaration de sollicitation ponctuelle> | <Déclaration de sollicitation répartie> | <Ensemble de sollicitations déplacé>

<Déclaration de sollicitation ponctuelle> ::= **ON** <Point> **:** **FORCE** = <Valeur ponctuelle> **DISPLACEMENT** = <Valeur ponctuelle>

<Déclaration de sollicitation répartie> ::= **IN** <Intervalle> **:** **FORCE** = <valeur répartie> **DISPLACEMENT** = <Valeur répartie>

<Intervalle> ::= **(** <Point> **;** <Point> **)**

<Valeur ponctuelle> ::= **(** **/** **;** **/** **)** | **(** **/** **;** <Réel> **)** | **(** <Réel> **;** **/** **)** | **(** <Réel> **;** <Réel> **)** | **N** **(** **/** **;** <Réel> **)** | **N** **(** <Réel> **;** **/** **)** | **N** **(** <Réel> **;** <Réel> **)**

<Valeur répartie> ::= **(** **/** **;** **/** **)** | **(** **/** **;** <Réel> **:** <Réel> **)** | **(** <Réel> **:** <Réel> **;** **/** **)** | **(** <Réel> **:** <Réel> **;** <Réel> **:** <Réel> **)** | **N** **(** **/** **;** <Réel> **:** <Réel> **)** | **N** **(** <Réel> **:** <Réel> **;** **/** **)** | **N** **(** <Réel> **:** <Réel> **;** <Réel> **:** <Réel> **)**

<Ensemble de sollicitations déplacé> ::= <Translation d'un ensemble de sollicitations> | <Rotation d'un ensemble de sollicitations> | <Symétrie d'un ensemble de sollicitations>

<Translation d'un ensemble de sollicitations> ::= **TRL** <Identificateur d'ensemble de sollicitations> **BY** (<Delta-X> ; <Delta-Y>)

<Rotation d'un ensemble de sollicitations> ::= **ROT** <Identificateur d'ensemble de sollicitations> **BY** (<Point> ; <Angle>)

<Symétrie d'un ensemble de sollicitations> ::= **SYM** <Identificateur d'ensemble de sollicitations> **BY** (<Point> ; <Point>)

<Déclaration du nombre de noeuds> ::= **NUMBER OF NODES** = <Réel>

<Déclaration du nbre de points d'intégration> ::= **INTEGRATION POINTS** = <Réel>

<Déclaration de gravité> ::= **GRAVITY** = <Angle> | **GRAVITY** = (<Angle> ; <Coéfficient multiplicateur>)

<Coéfficient multiplicateur> ::= <Réel>

<Déclaration de poids volumique> ::= **VOLUME WEIGHT** = <Réel>

<Déclaration du module d'élasticité> ::= **ELASTICITY MODULE** = <Réel>

<Déclaration du coefficient de poisson> ::= **POISSON COEFFICIENT** = <Réel>

<Suite de display> ::= <Ordre de display> | <Suite de display> <Ordre de display>

<Ordre de display> ::= **DISPLAY** <Identificateur de domaine> | **DISPLAY** <Identificateur de ss-domaine>

<Identificateur> ::= <Lettre> | <Lettre> <Suite de lettres, de chiffres ou d'underscores>

<Suite de lettres, de chiffres ou d'underscore> ::= <Lettre, chiffre ou underscore> | <Suite de lettres, de chiffres ou d'underscore> <Lettre, chiffre ou underscore>

<Lettre, chiffre ou underscore> ::= <Lettre> | <Chiffre> | <Underscore>

<Lettre> ::= **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**

<Chiffre> ::= **0** | **1** | ... | **9**

<Underscore> ::= _

<Réel> ::= <Partie entière> <Partie décimale> <Partie exposant> | <Partie décimale> <Partie exposant> | <Partie entière> <Partie exposant> | <Partie entière> <Partie décimale> | <Partie entière> | <Partie décimale> | <Partie exposant>

<Partie entière> ::= <Nombre entier> | <Signe> <Nombre entier>

<Partie décimale> ::= . <Nombre entier>

<Partie exposant> ::= **E** <Nombre entier> | **E** <Signe> <Nombre entier>

<Signe> ::= **+** | **-**

<Nombre entier> ::= <Chiffre> | <Nombre entier> <Chiffre>

Contraintes sémantiques

- Pour une déclaration de sollicitation, il faut imposer deux et exactement deux valeurs (les deux autres valeurs sont donc inconnues : /).
- Si on veut afficher le domaine, on ne peut afficher l'un quelconque des sous-domaines.
  On ne peut afficher deux fois le même sous-domaine.

- Les identificateurs de domaines, de sous-domaines, d'objets et d'ensembles de sollicitations doivent être uniques.
- Les cercles doivent avoir un rayon strictement positif.
- Les rectangles doivent avoir des longueurs et largeurs positives.
- Les triangles doivent être définis par 3 points distincts et non situés tous trois sur une même droite.
- Les deux points définissant l'axe de symétrie pour une opération de symétrie doivent être distincts.
- Les angles peuvent être négatifs. Toutefois ils doivent être inférieurs en valeur absolue à 360 degrés.

Exemples de fichiers sources corrects

Exemple 1

```
DOMAIN EXEMPLE1 =

   SUBDOMAIN PIECE =
      %POINT = P (90,-120)
      &RECT1 = RECT (199;160) AT (-45,-160)
      &RECT2 = RECT (64;71) AT (90,-71)
      &RECT3 = RECT (35;25) AT (154,-160)
      &RECT4 = ROT RECT (80;180) AT %POINT
               BY (%POINT;-120)
      &CERCLE1 = CIRCLE (64) AT (154,-71)
      &TIERS1 = DIFF DIFF DIFF UNION &RECT1
                                  WITH  &RECT3
                            WITH &CERCLE1
                       WITH &RECT2
                 WITH &RECT4
      &TIERS2 = ROT &TIERS1
                BY ((0,0);120)
      &TIERS3 = ROT &TIERS2
                BY ((0,0);120)
      &HELICE = UNION UNION &TIERS1
                       WITH &TIERS2
                WITH &TIERS3
      &QUART = UNION UNION CIRCLE (8) AT (0,53)
                     WITH CIRCLE (6.1) AT P (53,45)
               WITH CIRCLE (6.1) AT (53,0)
      &DEMI = UNION &QUART
              WITH SYM &QUART BY ((0,0);(0,53))
      &UNITE = UNION &DEMI
               WITH SYM &DEMI BY ((0,0);(53,0))
      &TROUS = UNION &UNITE
               WITH CIRCLE (35)
      PIECE = DIFF &HELICE WITH &TROUS

   SET OF SOLLICITATIONS Set_1 =
      ON (35,0) : FORCES = (1;2)
```

```
                DISPLACEMENTS = (/;/)

   NUMBER OF NODES = 50
   INTEGRATION POINTS = 6
   GRAVITY = 270
   VOLUME WEIGHT = 2.32
   ELASTICITY MODULE = 210000
   POISSON COEFFICIENT = 0.8

   DISPLAY EXEMPLE
```

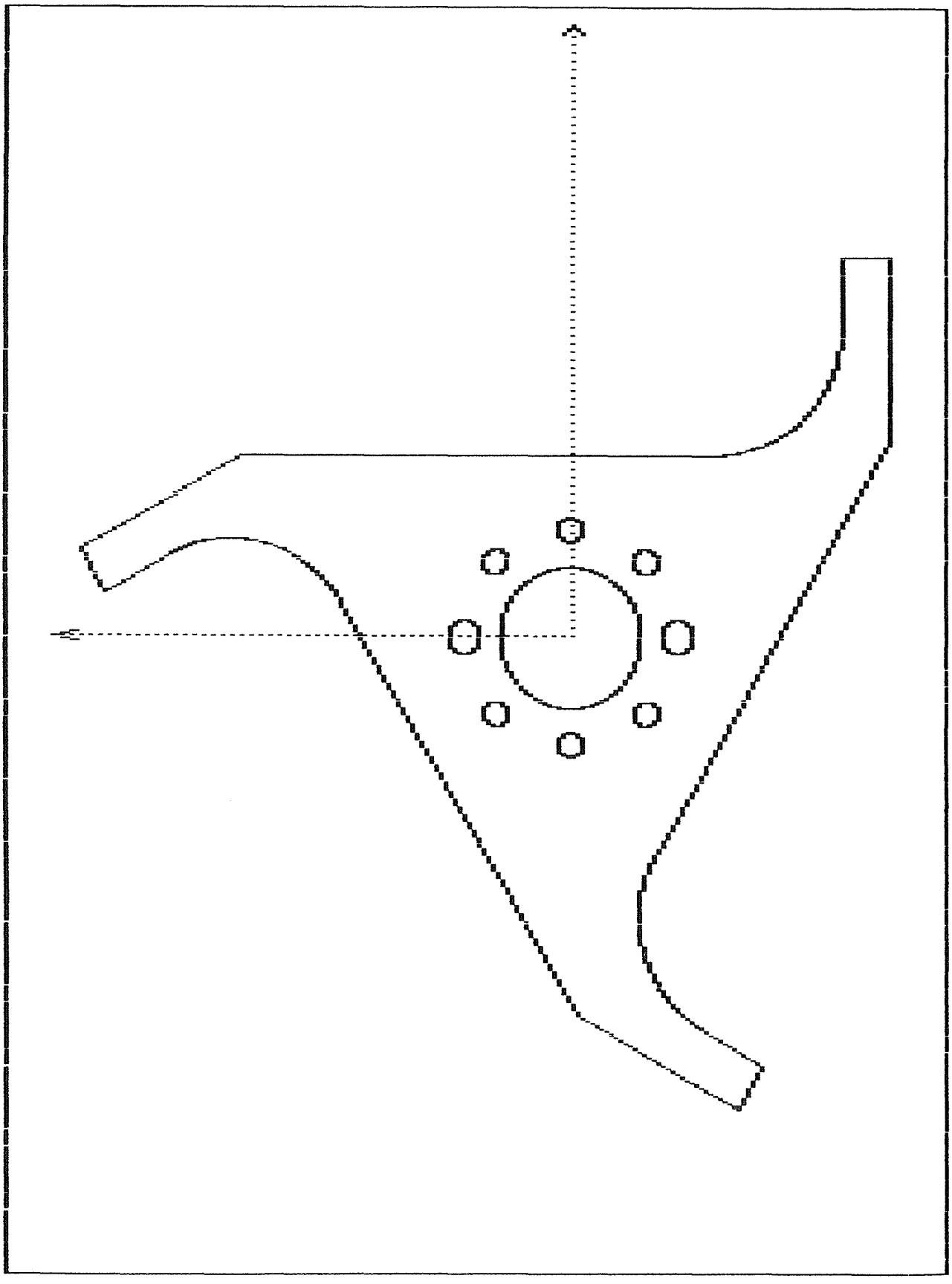
Exemple 2

```
DOMAIN Example2 =

   SUBDOMAIN PIECE2 =
      %ORIGINE = (0,0)
      &A = CIRCLE (4)
      &B = CIRCLE (1) AT (1.5,1.5)
      &R1 = RECT (1;0.5) AT (3.5,-0.25)
      &R2 = ROT &R1 BY (%ORIGINE;30)
      &R3 = ROT &R1 BY (%ORIGINE;60)
      &QUART = UNION UNION &R1
                       WITH &R2
                WITH &R3
      &DEMI = UNION &QUART
              WITH ROT &QUART BY (%ORIGINE;90)
      &ENTITE = UNION &DEMI
                 WITH ROT &DEMI BY (%ORIGINE;180)
      &TROUS = UNION UNION UNION &B
                             WITH ROT &B BY (%ORIGINE;90)
                       WITH ROT &B BY (%ORIGINE;180)
                WITH ROT &B BY (%ORIGINE;270)
      PIECE2 = DIFF UNION &A
                       WITH &ENTITE
                WITH &TROUS

   SET OF SOLLICITATIONS Set_1 =
      ON P(4,45) : FORCES = (0:1;0:2)
```

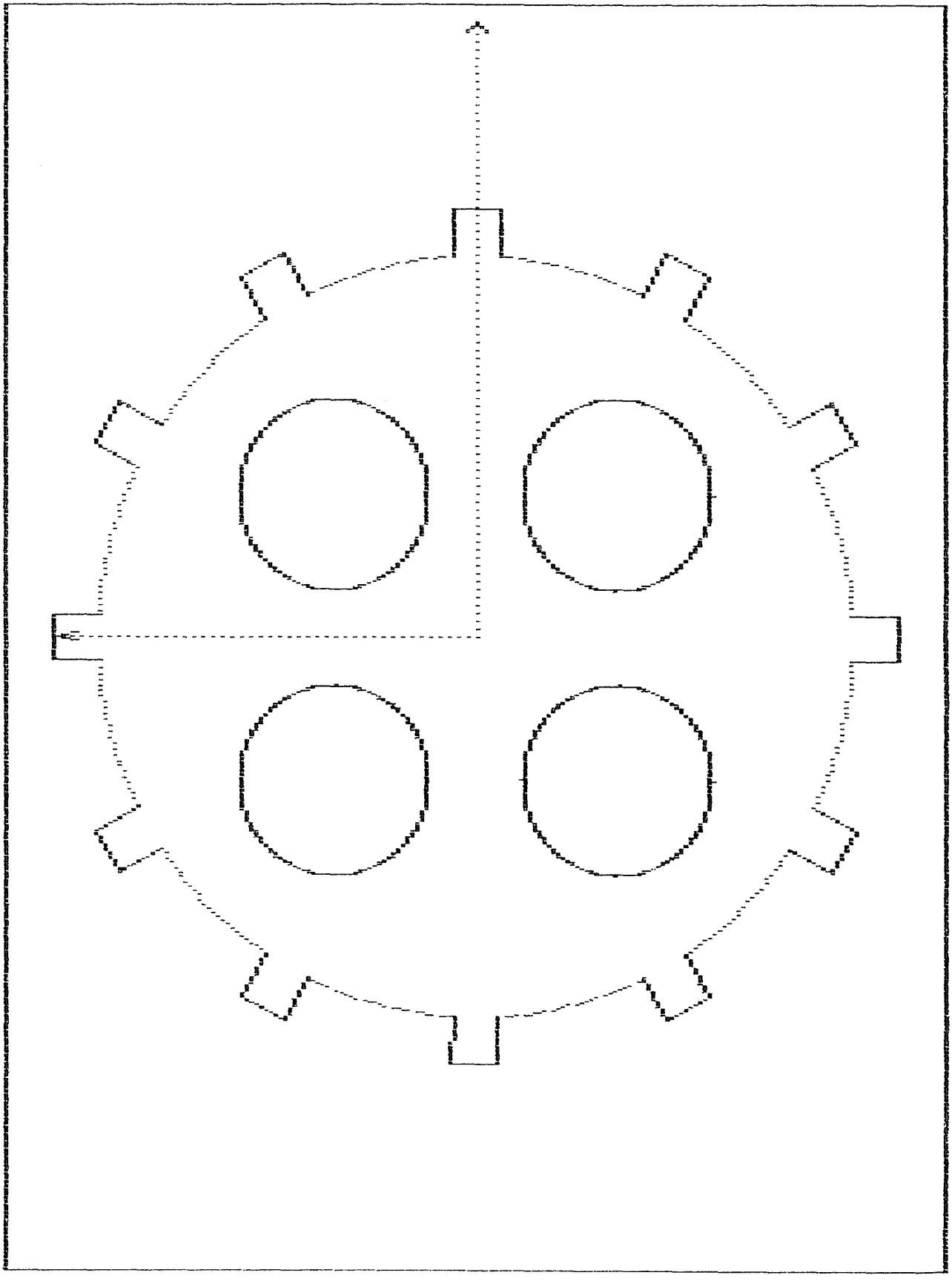
```
DISPLACEMENTS = (/;/)

NUMBER OF NODES = 50
INTEGRATION POINTS = 6
GRAVITY = 270
VOLUME WEIGHT = 2.32
ELASTICITY MODULE = 210000
POISSON COEFFICIENT = 0.8

DISPLAY PIECE2
```

Exemples d'affichages de pièces

Exemple 1



Exemple 2



Exemple de fichier généré

Exemple 1

```
 0
88          1          0          0
 1
 1    3.4999107142857E+01    2.4999840560722E-01
 2   -2.4999993546684E-01    3.4999107131929E+01
 3   -3.4999107121001E+01   -2.5000146532646E-01
 4    2.5000299518608E-01   -3.4999107110074E+01
 5    3.4999107099145E+01    2.5000452504570E-01
 6    3.5000000000000E+01    0.0000000000000E+00
 7   -5.9100000000000E+01   -5.3327895802272E-07
 8   -5.2999999200082E+01   -6.0999999999999E+00
 9   -4.6900000000000E+01    1.0665579160454E-06
10   -5.3000001333197E+01    6.0999999999999E+00
11   -5.9100000000000E+01   -1.5998368740681E-06
12   -4.3576658583809E+01   -3.7476660755244E+01
13   -3.7476657783890E+01   -4.3576660221965E+01
14   -3.1376658583809E+01   -3.7476659155408E+01
15   -3.7476659917006E+01   -3.1376660221966E+01
16   -4.3576658583808E+01   -3.7476661821802E+01
17   -8.0000000000000E+00   -5.3000000699382E+01
18    1.0490733600447E-06   -6.1000000000000E+01
19    7.9999999999999E+00   -5.2999998601236E+01
20   -1.7484556000745E-06   -4.5000000000000E+01
21   -7.9999999999997E+00   -5.3000002098147E+01
22    5.9100000000000E+01    0.0000000000000E+00
23    5.2999999733361E+01    6.1000000000000E+00
24    4.6900000000000E+01   -5.3327895802272E-07
25    5.3000000799918E+01   -6.0999999999999E+00
26    5.9100000000000E+01    1.0665579160454E-06
27    4.3576658583809E+01   -3.7476660221965E+01
28    3.7476658317169E+01   -3.1376660221965E+01
29    3.1376658583809E+01   -3.7476660755244E+01
30    3.7476659383727E+01   -4.3576660221965E+01
31    4.3576658583808E+01   -3.7476659155408E+01
32   -4.3576658583809E+01    3.7476659688686E+01
33   -3.7476657783890E+01    3.1376660221966E+01
34   -3.1376658583809E+01    3.7476661288523E+01
35   -3.7476659917006E+01    4.3576660221965E+01
36   -4.3576658583808E+01    3.7476658622129E+01
37   -8.0000000000000E+00    5.2999999300618E+01
38    1.0490733600447E-06    4.5000000000000E+01
39    7.9999999999999E+00    5.3000001398764E+01
40   -1.7484556000745E-06    6.1000000000000E+01
41   -7.9999999999997E+00    5.2999997901853E+01
42    4.3576658583809E+01    3.7476660221965E+01
43    3.7476658317169E+01    4.3576660221965E+01
44    3.1376658583809E+01    3.7476659688686E+01
45    3.7476659383727E+01    3.1376660221966E+01
46    4.3576658583808E+01    3.7476661288523E+01
47    1.8900000000000E+02   -1.6000000000000E+02
48    1.8900000000000E+02   -1.3500000000000E+02
49    1.8900000000000E+02   -1.3500000000000E+02
50    1.5400000000000E+02   -1.3500000000000E+02
51    9.0000000000000E+01   -7.1000005595058E+01
```

52	1.0874516825204E+02	-1.1625483624392E+02
53	1.5400000076319E+02	-1.3500000000000E+02
54	9.0000000000000E+01	-7.1000000000000E+01
55	8.9999991032858E+01	-1.2221139522519E+01
56	8.9999982065716E+01	4.6557720954961E+01
57	8.9999973098574E+01	1.0533658143244E+02
58	8.9999964131432E+01	1.6411544190992E+02
59	8.9999964131432E+01	1.6411544190992E+02
60	4.4064007797300E+01	2.4367881158779E+02
61	4.4064007797300E+01	2.4367881158779E+02
62	2.2413375616783E+01	2.3117880654044E+02
63	2.2413375616783E+01	2.3117880654044E+02
64	3.9913382683081E+01	2.0086792148771E+02
65	1.6487786913111E+01	1.1344229577942E+02
66	4.6307023941286E+01	1.5230351490051E+02
67	3.9913377088021E+01	2.0086793117864E+02
68	1.6487777222188E+01	1.1344229018436E+02
69	-3.4416198001649E+01	8.4052848492360E+01
70	-8.5320173225486E+01	5.4663406800364E+01
71	-1.3622414844932E+02	2.5273965108369E+01
72	-1.8712812367316E+02	-4.1154765836270E+00
73	-1.8712812367316E+02	-4.1154765836270E+00
74	-2.3306404022072E+02	-8.3678869232284E+01
75	-2.3306404022072E+02	-8.3678869232284E+01
76	-2.1141340148350E+02	-9.6178862923087E+01
77	-2.1141340148350E+02	-9.6178862923087E+01
78	-1.9391341031637E+02	-6.5867968690971E+01
79	-1.0648779440412E+02	-4.2442322747567E+01
80	-1.5505220970107E+02	-3.6048718254005E+01
81	-1.9391340145450E+02	-6.5867980267985E+01
82	-1.0648779130062E+02	-4.2442317372130E+01
83	-5.5583812368325E+01	-7.1831738029098E+01
84	-4.6798334360350E+00	-1.0122115868607E+02
85	4.6224145496255E+01	-1.3061057934303E+02
86	9.7128124428545E+01	-1.6000000000000E+02
87	9.7128124428545E+01	-1.6000000000000E+02
88	1.8900000000000E+02	-1.6000000000000E+02

88

1	0	0.0000000000000E+00	0.0000000000000E+00
2	0	0.0000000000000E+00	0.0000000000000E+00
3	0	0.0000000000000E+00	0.0000000000000E+00
4	0	0.0000000000000E+00	0.0000000000000E+00
5	0	0.0000000000000E+00	0.0000000000000E+00
6	0	1.0000000000000E+00	2.0000000000000E+00
7	0	0.0000000000000E+00	0.0000000000000E+00
8	0	0.0000000000000E+00	0.0000000000000E+00
9	0	0.0000000000000E+00	0.0000000000000E+00
10	0	0.0000000000000E+00	0.0000000000000E+00
11	0	0.0000000000000E+00	0.0000000000000E+00

12	0	0.00000000000000E+00	0.00000000000000E+00
13	0	0.00000000000000E+00	0.00000000000000E+00
14	0	0.00000000000000E+00	0.00000000000000E+00
15	0	0.00000000000000E+00	0.00000000000000E+00
16	0	0.00000000000000E+00	0.00000000000000E+00
17	0	0.00000000000000E+00	0.00000000000000E+00
18	0	0.00000000000000E+00	0.00000000000000E+00
19	0	0.00000000000000E+00	0.00000000000000E+00
20	0	0.00000000000000E+00	0.00000000000000E+00
21	0	0.00000000000000E+00	0.00000000000000E+00
22	0	0.00000000000000E+00	0.00000000000000E+00
23	0	0.00000000000000E+00	0.00000000000000E+00
24	0	0.00000000000000E+00	0.00000000000000E+00
25	0	0.00000000000000E+00	0.00000000000000E+00
26	0	0.00000000000000E+00	0.00000000000000E+00
27	0	0.00000000000000E+00	0.00000000000000E+00
28	0	0.00000000000000E+00	0.00000000000000E+00
29	0	0.00000000000000E+00	0.00000000000000E+00
30	0	0.00000000000000E+00	0.00000000000000E+00
31	0	0.00000000000000E+00	0.00000000000000E+00
32	0	0.00000000000000E+00	0.00000000000000E+00
33	0	0.00000000000000E+00	0.00000000000000E+00
34	0	0.00000000000000E+00	0.00000000000000E+00
35	0	0.00000000000000E+00	0.00000000000000E+00
36	0	0.00000000000000E+00	0.00000000000000E+00
37	0	0.00000000000000E+00	0.00000000000000E+00
38	0	0.00000000000000E+00	0.00000000000000E+00
39	0	0.00000000000000E+00	0.00000000000000E+00
40	0	0.00000000000000E+00	0.00000000000000E+00
41	0	0.00000000000000E+00	0.00000000000000E+00

42	0	0.00000000000000E+00	0.00000000000000E+00
43	0	0.00000000000000E+00	0.00000000000000E+00
44	0	0.00000000000000E+00	0.00000000000000E+00
45	0	0.00000000000000E+00	0.00000000000000E+00
46	0	0.00000000000000E+00	0.00000000000000E+00
47	0	0.00000000000000E+00	0.00000000000000E+00
48	0	0.00000000000000E+00	0.00000000000000E+00
49	0	0.00000000000000E+00	0.00000000000000E+00
50	0	0.00000000000000E+00	0.00000000000000E+00
51	0	0.00000000000000E+00	0.00000000000000E+00
52	0	0.00000000000000E+00	0.00000000000000E+00
53	0	0.00000000000000E+00	0.00000000000000E+00
54	0	0.00000000000000E+00	0.00000000000000E+00
55	0	0.00000000000000E+00	0.00000000000000E+00
56	0	0.00000000000000E+00	0.00000000000000E+00
57	0	0.00000000000000E+00	0.00000000000000E+00
58	0	0.00000000000000E+00	0.00000000000000E+00
59	0	0.00000000000000E+00	0.00000000000000E+00
60	0	0.00000000000000E+00	0.00000000000000E+00
61	0	0.00000000000000E+00	0.00000000000000E+00
62	0	0.00000000000000E+00	0.00000000000000E+00
63	0	0.00000000000000E+00	0.00000000000000E+00
64	0	0.00000000000000E+00	0.00000000000000E+00
65	0	0.00000000000000E+00	0.00000000000000E+00
66	0	0.00000000000000E+00	0.00000000000000E+00
67	0	0.00000000000000E+00	0.00000000000000E+00
68	0	0.00000000000000E+00	0.00000000000000E+00
69	0	0.00000000000000E+00	0.00000000000000E+00
70	0	0.00000000000000E+00	0.00000000000000E+00
71	0	0.00000000000000E+00	0.00000000000000E+00

```
        72                  0       0.00000000000000E+00
0.00000000000000E+00
        73                  0       0.00000000000000E+00
0.00000000000000E+00
        74                  0       0.00000000000000E+00
0.00000000000000E+00
        75                  0       0.00000000000000E+00
0.00000000000000E+00
        76                  0       0.00000000000000E+00
0.00000000000000E+00
        77                  0       0.00000000000000E+00
0.00000000000000E+00
        78                  0       0.00000000000000E+00
0.00000000000000E+00
        79                  0       0.00000000000000E+00
0.00000000000000E+00
        80                  0       0.00000000000000E+00
0.00000000000000E+00
        81                  0       0.00000000000000E+00
0.00000000000000E+00
        82                  0       0.00000000000000E+00
0.00000000000000E+00
        83                  0       0.00000000000000E+00
0.00000000000000E+00
        84                  0       0.00000000000000E+00
0.00000000000000E+00
        85                  0       0.00000000000000E+00
0.00000000000000E+00
        86                  0       0.00000000000000E+00
0.00000000000000E+00
        87                  0       0.00000000000000E+00
0.00000000000000E+00
        88                  0       0.00000000000000E+00
0.00000000000000E+00
         0
        88                 43              0 2.10000000000000E+05
8.00000000000000E-01
         1                  3               6               1
2                3
         2                  3               6               3
4                5
         3                  3               6               5
6                7
         4                  3               6               7
8                9
         5                  3               6               9
10               11
         6                  3               6              12
13               14
         7                  3               6              14
15               16
         8                  3               6              17
18               19
         9                  3               6              19
20               21
        10                  3               6              22
23               24
        11                  3               6              24
25               26
```

28	12	29	3	6	27
30	13	31	3	6	29
33	14	34	3	6	32
35	15	36	3	6	34
38	16	39	3	6	37
40	17	41	3	6	39
43	18	44	3	6	42
45	19	46	3	6	44
48	20		2	6	47
50	21		2	6	49
52	22	53	3	6	51
55	23		2	6	54
56	24		2	6	55
57	25		2	6	56
58	26		2	6	57
60	27		2	6	59
62	28		2	6	61
64	29		2	6	63
66	30	67	3	6	65
69	31		2	6	68
70	32		2	6	69
71	33		2	6	70
72	34		2	6	71
74	35		2	6	73
76	36		2	6	75
78	37		2	6	77
80	38	81	3	6	79
83	39		2	6	82
84	40		2	6	83
85	41		2	6	84

```
        42                  2                   6                   85
86
        43                  2                   6                   87
88
 2.3200000000000E+00      4.7123891115189E+00
```

Partie 2

Etude de l'influence du mécanisme de mémoire virtuelle sur les performances de programmes

Introduction

Depuis l'avènement du premier ordinateur de Von Neuman en 1946, l'informatique a fait des progrès considérables. Les premiers ordinateurs étaient très imposants de par leur taille. Paradoxalement, leurs possibilités et souplesse étaient très limitées. Sur une période de quarante ans, grâce essentiellement au progrès de la miniaturisation, nous sommes passés de ces appareils gigantesques à des ordinateurs qui sont de petites merveilles. Leur taille s'est considérablement amenuisée, leur souplesse s'est améliorée et leurs possibilités se sont incroyablement accrues (on est passé, par exemple, de quelques mots mémoire pour les premiers ordinateurs à des capacités de 640 KBytes pour de simples micros). Cette évolution est allée de pair avec une diminution du coût du matériel informatique.

Toutefois, malgré cette baisse, les prix pratiqués sur le marché de l'informatique restent relativement élevés. Concernant cette dernière affirmation, il est nécessaire de détailler davantage. Si l'on prend l'exemple de la mémoire des ordinateurs, on s'aperçoit qu'il existe une hiérarchie des mémoires. Approximativement, on distingue deux types de mémoires : la mémoire centrale et la mémoire secondaire. La mémoire centrale se caractérise par des temps d'accès à une information très réduits (environ deux microsecondes pour un mini-ordinateur). Elle se caractérise également par un coût non négligeable. C'est pour cela qu'on la trouve en quantité relativement limitée sur les machines. La mémoire secondaire (typiquement un disque), par contre, a une vitesse d'accès à l'information beaucoup moindre que la mémoire centrale (30-40 millisecondes en moyenne pour un mini). Son coût est également beaucoup moindre. Quant à sa capacité de stockage, elle est très élevée.

Les ordinateurs actuels représentent donc un compromis entre la mémoire centrale, rapide mais coûteuse, et la mémoire secondaire plus lente et moins chère. Par exemple, sur un

VAX/VMS 8600, on trouve une mémoire centrale de 8 MegaBytes, et un grand nombre de disques possédant des capacités de l'ordre de 400 MegaBytes.

Parallèlement à ce développement du hardware, l'informatique s'est développée dans divers secteurs, au point qu'actuellement, on la rencontre dans tous les domaines de la vie quotidienne. Le nombre d'utilisateurs de l'informatique et leurs besoins se sont donc accrûs considérablement.

Avec l'apparition des mini-ordinateurs qui offrent la possibilité de servir plusieurs utilisateurs simultanément, on voit que le problème de la mémoire est encore plus crucial puisqu'ils doivent se la partager. Le problème est donc double :

- il y a de multiples utilisateurs pour une mémoire centrale limitée;
- les besoins des utilisateurs en mémoire pour résoudre certains problèmes peuvent être supérieurs à la quantité de mémoire centrale disponible.

On s'est donc évertué à trouver des mécanismes qui permettent de pallier ces deux problèmes. D'une part, des mécanismes de gestion de la mémoire assurant un compromis entre mémoire centrale et mémoire secondaire, de manière à ce que l'utilisateur ne souffre pas trop, du point de vue de la vitesse d'exécution de ses programmes, du manque de mémoire centrale. D'autre part, des mécanismes permettant à l'utilisateur d'utiliser un volume de données supérieur à la mémoire centrale sans devoir gérer cela explicitement dans son programme.

Le mécanisme de mémoire virtuelle est un mécanisme prévu à cet effet. Il permet d'offrir à l'utilisateur un espace mémoire virtuelle très supérieur à l'espace mémoire centrale disponible sur la machine. Cela quel que soit le nombre d'utilisateurs du système, de manière transparente, et en essayant d'influencer le moins possible le temps d'exécution des programmes de l'utilisateur.

Cependant, ce mécanisme, s'il est souvent présenté comme transparent vis-à-vis de l'utilisateur, ne l'est pas tout à fait. En effet, un utilisateur averti et connaissant précisément l'implémentation de ce mécanisme sur une machine particulière, peut très bien en tirer profit et arriver à des gains de performances assez considérables.

Nous verrons, dans cette partie, quelles solutions d'implémentation nous avons envisagées pour résoudre un problème de performance de programmes particuliers tenant compte du mécanisme de mémoire virtuelle tel qu'il est implémenté sur un mini-ordinateur de la famille VAX/VMS.

Chapitre 1
Exposé du problème

Nous exposerons, dans ce chapitre, le problème posé par une étude comparée de l'influence du mécanisme de mémoire virtuelle sur les algorithmes de programmes d'éléments finis.

Les programmes d'éléments finis sont des logiciels destinés à approcher le comportement de milieux continus : solides ou fluides de forme quelconque, représentables par un système aux dérivées partielles. Le principe de la méthode consiste à découper le milieu que l'on désire étudier en éléments finis connectés entre eux en certains points appelés noeuds auxquels sont associées les inconnues discrètes du problème. Ces programmes d'éléments finis se composent de plusieurs phases distinctes. Seule une d'entre elles nous intéresse vraiment pour notre problème. Il s'agit d'une phase où on recherche la solution d'un système d'équations linéaires. Voyons quelles raisons font que cette partie de programme est particulièrement critique pour les programmes d'éléments finis.

Tout d'abord, on manipule dans cette partie une structure de données (la matrice représentant le système d'équations) qui peut atteindre des tailles considérables. Les programmes d'éléments finis manipulent couramment des systèmes de 5000 équations.

Ensuite, cette résolution du système n'aboutit pas directement à la solution; elle s'inscrit plutôt dans le cadre d'une méthode d'approximaton de la solution réelle. Cette phase de résolution est donc répétée un nombre non négligeable de fois avant d'arriver à une solution jugée suffisamment approchée de la solution réelle.

Ces deux considérations montrent que cette phase de résolution nécessite beaucoup de calculs. En fait, elle représente à elle seule une très grande partie des calculs effectués par les programmes d'éléments finis, et cela prend

toute sa signification quand on sait que ces derniers s'exécutent pendant plusieurs dizaines d'heures. Cette phase est donc critique pour ces programmes du point de vue de ses performances d'exécution.

La majorité des logiciels d'éléments finis ont été conçus à une époque où les machines à mémoire virtuelle n'existaient pas encore. La taille de la mémoire centrale des machines de l'époque étant assez limitées, et de toute façon inférieure à la taille de la matrice représentant le système d'équations, on devait systématiquement transférer des données intermédiaires en mémoire secondaire, typiquement sur disques. Le schéma d'exécution était donc simple : on chargeait un premier bloc de la matrice en mémoire centrale, on lui appliquait une méthode de résolution, on la réécrivait sur disque, et ainsi de suite jusqu'à avoir épuisé tous les blocs de la matrice. On répétait cette opération approchée de la solution réelle. On était donc exposé à des pertes considérables de temps pour effectuer les entrées/sorties disque.

Par contre, sur les nouveaux équipements, ces transferts peuvent être automatiquement pris en charge par le système de gestion de la mémoire virtuelle. Les transferts de données vers les disques ne seront plus systématiquement gérés par les programmes d'éléments finis, mais seront implicitement pris en charge par le mécanisme de gestion de la mémoire virtuelle. De façon générale, les transferts de données vers les disques réalisés par le mécanisme de gestion de la mémoire virtuelle seront plus performants que des transferts disque demandés par l'utilisateur. Ceci devrait permettre un gain de temps appréciable vis-à-vis des autres programmes qui n'utilisent pas ce mécanisme.

On sait également que, bien que le mécanisme de mémoire virtuelle soit généralement présenté comme transparent pour l'utilisateur, il est possible d'en tirer des avantages appréciables, en temps calcul et place mémoire disponible, en se basant sur des organisations matricielles particulièrement adaptées à ce genre de problème.

Nous nous proposions donc, au départ, de réaliser des procédures de résolution de systèmes d'équations en tenant compte du fait que :

- nous ne devions pas gérer les trannsferts disques, ceux-ci étant gérés par le mécanisme de mémoire virtuelle;
- une structure de données adéquate pour représenter la matrice peut influencer très favorablement la vitesse d'exécution de ces procédures.

Nous aurions ensuite effectué des tests de performances pour ces procédures travaillant sur des systèmes comparables en taille à ceux manipulés par les programmes d'éléments finis. Nous aurions ensuite comparé les performances de ces procédures avec celles des routines de résolution des programmes d'éléments finis. Si le résultat s'avérait positif, il ne resterait plus qu'à substituer notre procédure à la routine de résolution existante, cela pour obtenir des programmes d'éléments finis beaucoup plus efficaces en temps calcul.

Chapitre 2
Implémentation du mécanisme de mémoire virtuelle sur VAX/VMS

Pour atteindre les objectifs fixés au chapitre précédent, nous disposions, comme équipement, d'un VAX/VMS de Digital Equipement Corporation. Nous développerons dans ce chapitre certains concepts nécessaires à la compréhension du mécanisme de gestion de la mémoire tel qu'il est implémenté dans l'operating system du VAX/VMS.

2.1 Gestion de la mémoire

L'operating system du VAX/VMS est un operating system multi-utilisateurs. Dans un tel contexte de multi-programmation, la taille totale des espaces mémoire occupés par tous les processus est souvent supérieure à la taille réelle de la mémoire physique. Le mécanisme de gestion de la mémoire du VAX/VMS détermine la manière dont la mémoire physique est allouée, et ceci à deux niveaux :
- à l'intérieur même d'un processus;
- entre tous les processus dans le système.

Nous développerons plus en détail le mécanisme de gestion au premier niveau, puisque seul celui-là nous intéresse vis-à-vis de notre problème. Nous n'approfondirons donc pas les mécanismes opérant au second niveau.

Dans les prochaines sections de ce chapitre, nous définirons les concepts de mémoire virtuelle, de working set d'un processus, puis nous expliciterons les mécanismes de gestion de la mémoire virtuelle, avec notamment, les concepts de liste des pages libres et liste des pages modifiées.

2.2 Mémoire virtuelle

La technique de gestion de la mémoire utilisée par l'operating system du VAX/VMS est dite à mémoire virtuelle.

La mémoire virtuelle est l'ensemble des endroits de stockage, en mémoire physique et sur disques, qui sont référencés par des adresses virtuelles. La taille de la mémoire virtuelle est l'ensemble des adresses virtuelles accessibles. L'architecture hardware des VAX étant une architecture 32 bits, l'espace d'adressage virtuel est 2^{32} ou 4.3 billions de bytes.

VAX/VMS partitionne cet espace d'adressage virtuel en un ensemble de zones mémoire de taille fixe, appelées pages virtuelles.

La mémoire physique est également divisée en pages (appelées cadres); les pages en mémoire physique sont de même taille que les pages en mémoire virtuelle, c'est-à-dire 512 bytes.

A ce stade de l'étude, deux remarques peuvent être formulées :

- seulement une partie d'un programme (les pages qui sont référencées) doit être en mémoire centrale pendant l'exécution; le reste peut se trouver sur disque en mémoire secondaire (en fait, dans le paging file comme on le verra plus tard).
- les programmes peuvent dépasser, en taille, la mémoire physique disponible.

Il est également intéressant de voir comment l'operating system garde la trace d'une page virtuelle. A chaque processus, l'operating system adjoint une table des pages. Comme le montre la figure 2.1, à chaque page virtuelle du processus on crée une entrée dans la table des pages. La table des pages contient donc autant d'entrées que le processus ne possède pages virtuelles. La figure 2.2 nous montre que chaque entrée dans la table des pages fournit une adresse physique (soit en mémoire centrale, soit sur disque) pour la page virtuelle correspondant à cette entrée. Pour chaque page virtuelle appartenant au processus, on crée donc une entrée dans la table des pages du processus, et on trouve à cet endroit l'adresse physique de cette page, soit en

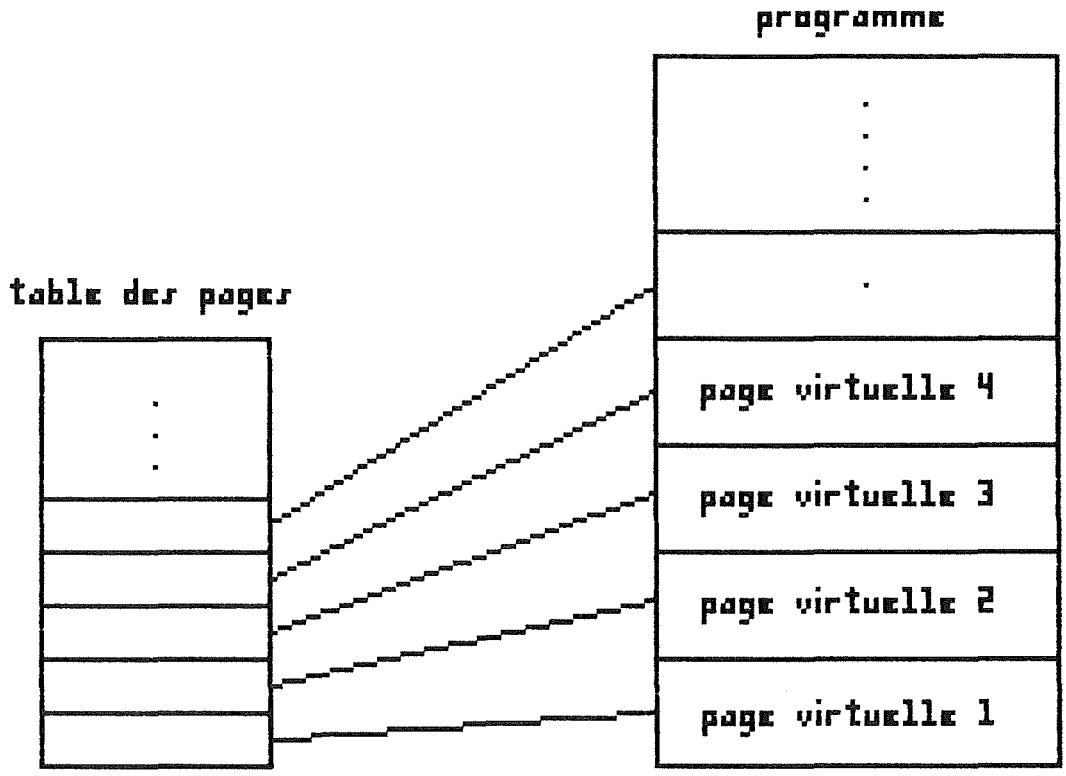


fig 2.1 pages du processus et table des pages

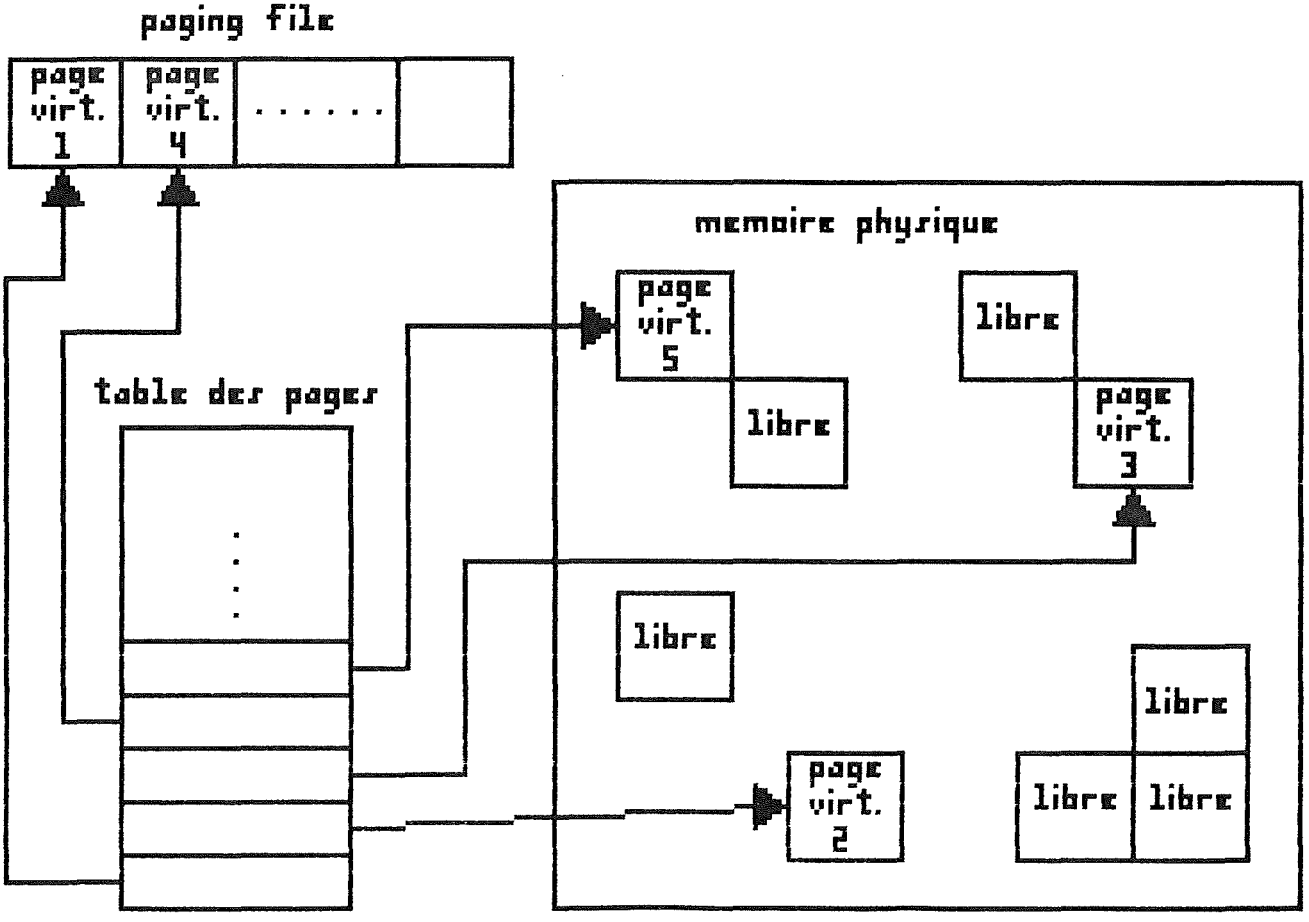


fig 2.2 Utilisation de la table des pages

mémoire centrale, soit sur disque. On voit donc comment on peut passer d'une adresse virtuelle à une adresse physique par une simple indirection et un calcul d'indice dans une table.

2.3 Working set

Comme nous l'avons déjà mentionné ci-dessus, lorsqu'un processus s'exécute, seulement un sous-ensemble de ses pages doit se trouver en mémoire centrale. Ce sous-ensemble est appelé le working set du processus. Le working set du processus est constitué de toutes les pages virtuelles du processus qui se trouvent en mémoire centrale, et auxquelles le processus peut accéder directement, sans déclencher de défaut de page. Un défaut de page est une référence à une page qui n'est pas dans le working set du processus. On développera cette notion plus en longueur par la suite.

Le working set est une caractéristique dynamique du processus. Il a une taille minimum et maximum. Sa taille minimum est fixée par le système (80 pages sous VAX/VMS), tandis que le gestionnaire du système fixe sa taille maximum, et cela pour chaque utilisateur du système.

2.4 Liste des pages libres et liste des pages modifiées

Nous verrons, dans cette section comment tous les concepts définis dans les sections précédentes s'articulent pour former le mécanisme de gestion de la mémoire virtuelle.

Définissons d'abord le concept de pagination. Paginer est l'action qui consiste à amener des pages d'un processus qui s'exécute en mémoire centrale, lorsque celles-ci sont référencées. Nous savons que lorsqu'un processus s'exécute, toutes ses pages se trouvent en mémoire virtuelle. Cependant, seulement un sous-ensemble de ses pages virtuelles doit se trouver en mémoire centrale, le reste pouvant rester sur disque jusqu'à ce que l'on en ait effectivement besoin. Une

page d'un processus est éjectée de la mémoire centrale seulement quand le processus référence une page qui n'est pas dans son working set, et que celui-ci est complètement utilisé. Quand un processus référence une page qui ne se trouve pas dans son working set, un défaut de page se produit. Un mécanisme (pager) doit alors rendre la page qui fait défaut accessible en mémoire centrale pour que le processus puisse continuer son exécution. Puisque cette page virtuelle n'est pas en mémoire centrale, elle se trouve sur disque dans le paging file du processus; c'est donc dans le paging file du processus que le pager va rechercher la page qui fait défaut pour l'amener en mémoire centrale. Le paging file contient donc toutes les pages virtuelles du processus, sauf celles qui sont en mémoire centrale.

La philosophie de pagination adoptée par l'operating system du VAX/VMS est dite "pagination contre le processus". Cela signifie que lorsqu'un défaut de page se produit et que le working set est entièrement utilisé, une page du working set du processus doit être éjectée pour laisser place à la nouvelle page qui sera lue sur le paging file. Voyons plus en détail la manière dont cela est organisé.

Liste des pages libres

La liste des pages libres est une liste de pages se trouvant en mémoire centrale et qui sont disponibles. Quand une page est éjectée du working set d'un processus, elle est insérée dans la liste des pages libres, si elle n'a pas été modifiée. Les pages sont insérées à la fin de cette liste, et retirées du début de cette liste. La politique de gestion de cette liste est donc FIFO (First In First Out). Notons que cette politique est propre à VAX/VMS et qu'on pourrait imaginer d'autres politiques : LRU (Last Recently Used) ou LFU (Last Frequently Used).

Remarquons la double fonction de la liste des pages libres : d'une part, elle est une source de pages disponibles en mémoire centrale pour le processus; d'autre part, elle sert

de cache pour les pages les plus récemment éjectées. En effet, lorsqu'un défaut de page se produit, plutôt que d'aller systématiquement chercher la page manquante sur disque dans le paging file, on va d'abord voir dans la liste des pages libres si la page manquante ne s'y trouve pas. Si elle s'y trouve, elle est disponible sans provoquer d'accès disque.

Liste des pages modifiées

La liste des pages modifiées est une liste dont le contenu doit être recopié sur disque (dans le paging file), avant que ces pages ne soient insérées dans la liste des pages libres. Quand une page est ejectée du working set, elle est placée dans la liste des pages modifiées si elle a été modifiée. Lorsque la liste des pages libres atteint un certain niveau (jugé insuffisant) de remplissage, la liste des pages modifiées est recopiée sur disque. Ainsi, les pages de la liste de pages modifiées s'ajoutent aux pages de la liste des pages libres pour en rehausser le nombre.

Similairement au mécanisme qui faisait que la liste des pages libres servait de cache pour les pages les plus récemment éjectées, une page de la liste des pages modifiées peut être récupérée en cas de défaut de page pour cette page, de manière à ne pas produire d'accès disque pour la récupérer dans le paging file.

On peut donc remarquer que chaque page d'un processus se trouve dans un des états suivants :

1° en dehors du working set du processus et disponible. Une telle page n'a jamais encore été référencée et donc amenée en mémoire centrale. Lorsqu'elle le sera, le mécanisme de pagination fournira alors une page remplie de zéros;
2° en dehors du working set du processus, et dans le paging file;
3° en transition, c'est-à-dire dans la liste des pages libres ou modifiées ou bien étant actuellement lue ou écrite sur disque;

4° Dans le working set du processus.

En tenant compte de tous ces concepts, nous pourrons montrer, dans le prochain chapitre, que le mécanisme de gestion de la mémoire virtuelle n'est pas entièrement transparent vis-à-vis de l'utilisateur, et comment on peut en tenir compte lors de l'écriture de programmes, afin d'en améliorer les performances.

Chapitre 3
Comment tenir compte du mécanisme de mémoire virtuelle

Nous montrerons, dans ce chapitre, que le mécanisme de mémoire virtuelle tel que nous l'avons décrit au chapitre précédent, n'est pas totalement transparent vis-à-vis de l'utilisateur. Nous verrons de quelle manière, par une structure organisationelle de matrice adaptée, on peut espérer des gains de performances considérables par rapport à une structure matricielle classique.

3.1 Non transparence de la mémoire virtuelle

Partons, pour montrer cette non transparence de la mémoire virtuelle vis-à-vis de l'utilisateur, d'un exemple simple. Soit un programme qui initialise une matrice B carrée de 800 X 800 éléments. Il existe deux manières de résoudre ce problème :

1° on parcourt la matrice ligne par ligne, en initialisant tous les éléments de la ligne. Voici le programme (en PASCAL) :

```
for i := 1 to 800 do
    for j := 1 to 800 do
        B[i,j] := 0;
```

2° on parcourt la matrice ligne par ligne, en initialisant un seul élément de chaque ligne à chaque itération. Intuitivement, on dira que cet algorithme parcourt les éléments de la matrice colonne par colonne.

```
for j := 1 to 800 do
    for i := 1 to 800 do
        B[i,j] := 0;
```

A priori, ces deux programmes peuvent paraître semblables. En effet, ils répondent tous deux à la même spécification, la seule différence étant une différence de stratégie d'accès aux

éléments de la matrice, un programme parcourant les éléments de la matrice ligne par ligne, l'autre colonne par colonne.

Cependant, lorsqu'on exécute ces deux programmes sur VAX/VMS 780, on trouve une différence assez nette entre les temps écoulés pendant l'exécution. Vingt et une secondes se sont écoulées avant que le premier programme ne termine son exécution, et trente et une minutes avant que le second ne termine la sienne, soit un rapport de un à cent ! Comment expliquer cette énorme différence entre deux programmes à priori semblables ? Par le mécanisme de mémoire virtuelle.

Il faut tout d'abord savoir que dans langage PASCAL, la déclaration

array [1..800,1..800] **of double**;

est une abréviation de

array [1..800] **of array** [1..800] **of** double;

Nous voyons donc qu'une matrice, en PASCAL, est un tableau de lignes (tableaux); il n'y a donc pas de notion de colonne en PASCAL. Dorénavant donc, quand on fera référence aux colonnes (respectivement lignes) d'une matrice, on se référera à la notion intuitive de colonnes (respectivement lignes) d'une matrice. Lorsqu'un compilateur rencontre une déclaration de structure matricielle, il réserve de la place en mémoire ligne par ligne pour les éléments de cette matrice. Cela signifie que deux éléments se suivant dans une même ligne auront certainement des places contiguës dans une page en mémoire, alors que deux éléments se suivant dans une même colonne, ne seront certainement pas contigüs dans une page en mémoire. Pour le premier programme, il y a de grandes chances que l'accès à un élément de la ligne se réduise à un accès mémoire, puisque cet élément est très probablement dans la même page que l'élément précédent, page qui est toujours en mémoire centrale. Pour le second programme, par contre, chaque accès à un élément de la colonne provoque, si la matrice est grande par rapport au working set du programme, un défaut de page, puisque les pages des éléments précédents ne contiennent très probablement pas cet élément; il faut donc amener une nouvelle page en mémoire centrale, puis effectuer l'initialisation de l'élément.

La différence entre les deux programmes provient donc du fait que le second provoque un très grand nombre de défauts de page, c'est-à-dire d'accès au disque. L'importance de cette différence s'explique par le fait qu'il faut 40 millisecondes pour accéder à une information sur disque, alors que 2 microsecondes sont suffisantes pour effectuer un accès à une information en mémoire centrale.

3.2 Nouvelle organisation des données

Cette section décrit une structure de données matricielle et son mode d'utilisation qui permettront d'éviter l'inconvénient majeur décrit dans la section précédente.

Soit, par exemple, un programme qui calcule le produit de deux matrices A = B X C, c'est-à-dire un programme qui calcule

$$a_{ij} = \sum_{k=0}^{m-1} b_{ik} c_{kj}$$ pour i et j allant de 0 à m-1.

Un tel programme va parcourir la matrice B ligne par ligne, et la matrice C colonne par colonne. Comme nous l'a montré la section précedente, il est fort probable qu'un tel programme aura des performances déplorables, puisqu'il va parcourir m fois toutes les colonnes de C, et que cela prend un temps très important.

Une solution à ce problème serait d'utiliser une nouvelle structure de données. Cette solution consiste donc, plutôt que d'utiliser une structure matricielle classique, à déclarer une matrice composée de sous-matrices d'éléments. La matrice est donc découpée en sous-matrices qui contiennent les éléments. Par ce fait, et comme le montre la figure 3.1, les sous-matrices seront rangées ligne par ligne dans la matrice, et les éléments seront rangés ligne par ligne dans les sous-matrices. Nous décidons également que ces sous-matrices auront exactement la taille d'une page physique en mémoire centrale. Nous savons qu'une page en mémoire centrale a une taille de 512 bytes, ce qui signifie que si l'on suppose que

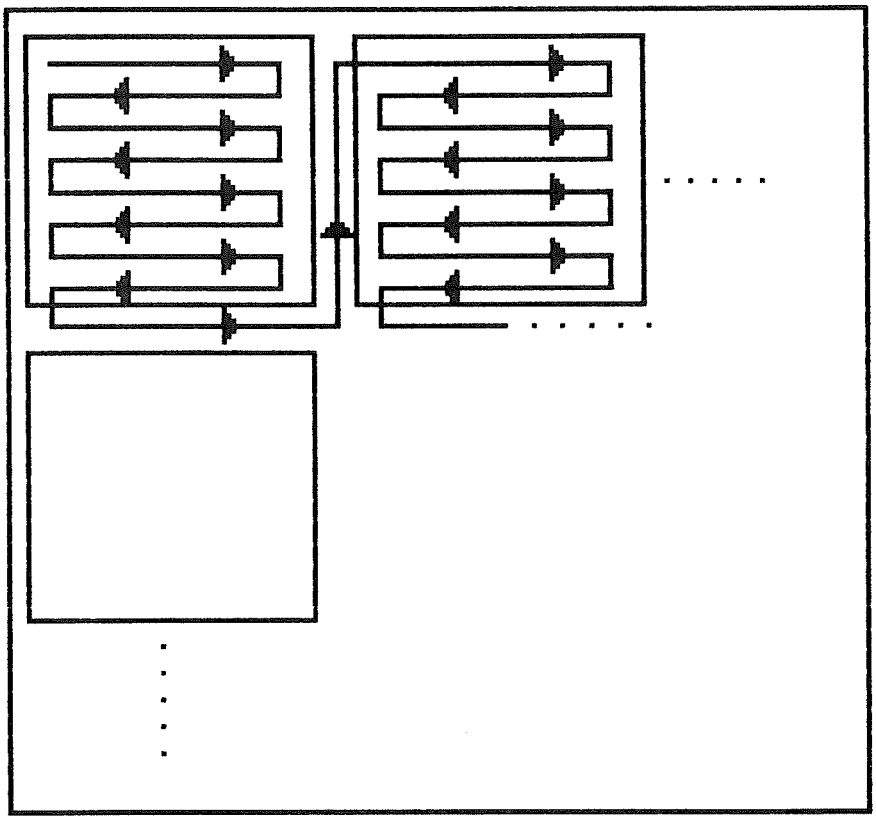


fig 3.1 stockage des elements d'une matrice de sous-matrices

les éléments de la matrice sont des doubles (8 bytes), les sous-matrices contiendront 512 / 8, c'est-à-dire 64 éléments. Celles-ci auront donc une dimension de 8 sur 8.

Avec cette structure de données, on remarque que l'accès aux éléments d'une matrice colonne par colonne ne provoque plus autant de défauts de page qu'avant. En effet, en voulant accéder à un élément d'une colonne, il est très probable que cet élément soit dans la même sous-matrice, donc dans la même page, que l'élément précédent, et on ne provoque donc plus de défaut de page, celle-ci étant déjà en mémoire centrale. C'est donc grâce à ce nombre de défauts de page, et donc d'accès disque, moins élevé, que l'on parvient à améliorer fortement les performances d'un programme.

Notons néanmoins que la correspondance entre sous-matrices et pages n'est pas aussi stricte que nous l'avons annoncée. En effet, il est peu probable qu'une sous-matrice de 512 bytes soit exactement assignée à une page. Cependant, cela n'influence en rien la bonne marche du système, puisqu'il est très probable qu'en passant à l'élément suivant d'une colonne de la matrice on reste malgré tout dans la même page, donc on ne provoque pas de défaut de page.

Voyons maintenant comment il est possible, avec un langage comme PASCAL, de créer une structure de données comme celle décrite ci-dessus, et surtout comment on peut accéder à l'information de cette structure. Considérons qu'on veuille représenter une matrice B de taille 800 X 800. On fera les déclarations PASCAL suivantes :

```
TYPE   Submat = array [0..7,0..7] of double;
       Mat    = array [0..99,0..99] of submat;
VAR    B : mat;
```

On remarque que les sous-matrices sont bien 8 X 8, et qu'il en faut 100 X 100 pour couvrir une matrice 800 X 800. Si l'on veut accéder à l'élément B[i,j] de la matrice B, on écrira

```
B[k,l][m,n]
```

où k = i div 8, l = j div 8, m = i mod 8, n = j mod 8, puisque les bornes inférieures des matrices et sous-matrices commencent à 0.

On se créera donc une fonction VAL définie comme suit :

```
Function VAL (Var B : Mat; X, Y : integer) : double;
begin
   VAL := B[X div 8, Y div 8][X mod 8, Y mod 8]
end;
```

et qui a pour but de renvoyer dans val la valeur de l'élément B[X,Y], et une procedure ASSIGN :

```
Procedure ASSIGN (Var B : Mat; X, Y : integer; EXP : double);
begin
   B[X div 8, Y div 8][X mod 8, Y mod 8] := EXP
end;
```

qui permet d'assigner la valeur exp à l'élément B[X,Y].

Ces deux procédures ont donc pour but de cacher la manière dont est implémentée la structure matricielle, et de fournir à l'utilisateur une vision classique de sa matrice.

Néanmoins, si ces deux procédures ont l'avantage d'un plus grand détachement du programmeur vis-à-vis de la structure de données qu'il manipule, elles ont également un inconvénient à ne pas négliger. En effet, l'utilisateur appelle ces procédures à chaque fois qu'il veut référencer un élément de la matrice, c'est-à-dire un très grand nombre de fois. Or, il faut savoir qu'un appel de procédure sur une machine est une opération coûteuse, car le système doit sauver des informations sur un stack en vue de, par exemple, retracer l'historique des appels en cas d'erreur. On a calculé qu'il fallait environ une trentaine de microsecondes pour entrer et sortir d'une procédure. On voit donc clairement qu'il convient d'abandonner l'idée de ces procédures car elles sont trop coûteuses en temps d'exécution. Il faudra donc, pour accéder à l'élément B[X,Y] de la matrice, écrire directement dans le programme le texte B[X div 8, Y div 8][X mod 8, Y mod 8].

Optimisation

Cette dernière phrase nous amène à proposer un moyen d'optimiser quelque peu le programme. Nous savons que nous accédons à l'élément B[X,Y] de la matrice par la formule B[X div 8, Y div 8][X mod 8, Y mod 8]. Chaque référence à un élément de la matrice requièrt le calcul de deux divisions entières et de deux restes de divisions entières. Ces opérations étant assez coûteuses sur un ordinateur, il convient de proposer une solution qui permettrait d'éviter de les répéter un grand nombre de fois. Nous nous sommes donc donné deux tables, divtab et modtab, qui contiendrons le résultat de la division entière et du reste de la division entière, pour chaque indice possible dans la matrice. Ces deux tables auront donc des longueurs égales au nombre d'équations dans le système à résoudre. Dorénavant donc, pour accéder à l'élément B[X,Y] de la matrice, on écrira directement dans le programme le texte B[divtab[X], divtab[Y]][modtab[X], modtab[Y]].

3.3 Conclusions

Nous avons montré, dans ce chapitre, que le mécanisme de mémoire virtuelle n'était pas totalement transparent pour un utilisateur du système. Nous avons également vu une méthode qui permet de tenir compte du mécanisme de mémoire virtuelle pour améliorer fortement les performances de certains programmes. Cette méthode consiste essentiellement à utiliser une matrice de sous-matrices d'éléments comme structure de données, plutôt qu'une matrice d'éléments. Nous avons montré comment utiliser au mieux cette structure de données. Il nous reste donc à découvrir comment nous avons adapté cette méthode de division de matrice en sous-matrices pour résoudre notre problème défini au chapitre 2, et voir si les résultats de cette adaptation sont concluants.

chapitre 4
Implémentations, tests et conclusions

Ce chapitre est consacré à l'exposé des solutions d'implémentation que nous avons imaginées pour répondre au problème défini au chapitre 1. Nous expliquerons aussi la manière dont nous avons mené les tests de performances pour ces solutions. Nous présenterons les résultats de ces tests et nous tirerons et argumenterons des conclusions quant à savoir si ces solutions offrent de réelles possibilités d'amélioration pour les programmes d'éléments finis.

4.1 Préliminaires

Avant d'entrer dans le vif du sujet, il est nécessaire de préciser davantage le problème posé. Cela se fera en deux temps : tout d'abord présenter les algorithmes utilisés pour résoudre le système d'équations, ensuite définir ce qu'est la bande d'une matrice, qui est une caractéristique particulière de toutes les matrices manipulées par les programmes d'éléments finis.

4.1.1 Algorithmes utilisés

Nous savons que c'est la routine de résolution du système d'équations linéaires qui est la partie critique des programmes d'éléments finis. Nous exposerons ici quelques algorithmes qui permettent de résoudre un système d'équations de la forme

$$K\ I = F \qquad (4.1)$$

où I est le vecteur des n inconues du système, K la matrice des coefficients du système et F le vecteur des seconds membres. Nous ne présenterons ici que les algorithmes de

résolution, sans montrer les étapes intermédiaires pour construire ces algorithmes. Pour plus de détails, on peut se référer à [Dhatt et Touzot 1981].

4.1.1.1 Méthode d'élimination de Gauss

Cette méthode est constituée de deux étapes :

- une étape de triangularisation : cette étape consiste à passer du système K I = F au système S I = F' ou S est une matrice triangulaire supérieure. On présente ci-dessous une algorithme qui répond aux spécifications suivantes :
  - Précondition : . $A_{i,j} = K_{i,j}$, pour tout i et j tels que $1 <= i <= n$ et $1 <= j <= n$
    . $M_j = F_j$, pour tout j tel que $1 <= j <= n$
  - Postcondition : . $A_{i,j} = S_{i,j}$, pour tout i et j tels que $1 <= i <= n$ et $1 <= j <= n$
    . Puisque S est triangulaire supérieure, $A_{i,j} = 0$, pour tout i et j tels que $1 <= i <= n$, $1 <= j <= n$ et $i > j$
    . $M_j = F'_j$, pour tout j tel que $1 <= j <= n$

```
For l := 1 to n - 1 do
    For i := l + 1 to n do
        c   := A_il / A_ll
        M_i := M_i - c * M_l
        For j := l + 1 to n do
            A_ij := A_ij - c * A_lj
        Endfor
    Endfor
Endfor
```

- une étape de résolution du système triangulaire supérieur : cette étape consiste à calculer les inconnues I de la dernière à la première, par résolution du système triangulaire (c'est le processus de "Back substitution").

- Précondition : . $A_{i,j} = S_{i,j}$, pour tout i et j tels que
  $1 <= i <= n$ et $1 <= j <= n$
  . $A_{i,j} = 0$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i > j$
  . $M_j = F'_j$, pour tout j tel que
  $1 <= j <= n$
- Postcondition : I_j est le vecteur des n inconnues du système K I = F.

Voici l'algorithme :

For i := n **downto** 1 **do**

$$I_i := (M_i - \sum_{j=i+1}^{n} A_{i,j} * I_j) / A_{i,i}$$

Endfor

4.1.1.2 Méthode de décomposition LU

Cette méthode décompose la matrice K des coefficients en un produit de deux matrices K = L U où L est une matrice triangulaire inférieure à termes diagonaux unitaires et U est une matrice triangulaire supérieure. On trouve dans [Dhatt et Touzot 1981] deux algorithmes pour décomposer la matrice K en deux matrices LU.

Les spécifications des deux algorithmes sont les suivantes :

- Précondition : $A_{i,j} = K_{i,j}$, pour tout i et j tels que
  $1 <= i <= n$ et $1 <= j <= n$
- Postcondition : . $A_{i,j} = L_{i,j}$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i < j$
  . $A_{i,j} = U_{i,j}$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i >= j$

1° For l := 2 to n do
    For i := 1 to l - 1 do
        For j := 1 to i - 1 do
            $A_{i,l} := A_{i,l} - A_{i,j} * A_{j,l}$;
            $A_{i,l} := A_{i,l} - A_{i,j} * A_{j,l}$;
        Endfor;
        $A_{i,l} := A_{i,l} / A_{i,i}$
    Endfor;
    For j := 1 to l - 1 do
        $A_{i,l} := A_{i,l} - A_{i,j} * A_{j,l}$
    Endfor
Endfor (4.1)

2° For l := 1 to n - 1 do
    For i := l + 1 to n do
        $A_{i,l} := A_{i,l} / A_{l,l}$;
        For j := l + 1 to n do
            $A_{i,j} := A_{i,j} - A_{i,l} * A_{l,j}$
        Endfor
    Endfor
Endfor (4.2)

Une fois la décomposition terminée, il ne reste plus qu'a résoudre le système L U I = F, ce qui se fait en deux étapes :

1° On résout le système L F' = F qui est un système triangulaire inférieur
2° On résout le système U I = F' qui est un système triangulaire supérieur

1° Précondition : . $A_{i,j} = L_{i,j}$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i > j$
  . $A_{i,j} = 0$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i < j$
  . $A_{i,j} = 1$, pour tout i et j tels que
  $1 <= i <= n$, $1 <= j <= n$ et $i = j$
  . $M_j = F_j$, pour tout j tel que $1 <= j <= n$
Postcondition : . $M_j = F'_j$, pour tout j tel que $1 <= j <= n$

For i := 2 **to** n **do**

$$M_i := M_i - \sum_{j=1}^{i-1} A_{ij} * M_j$$

Endfor

2° Précondition : . $A_{ij} = U_{ij}$, pour tout i et j tels que $1 <= i <= n$, $1 <= j <= n$ et $i <= j$

. $A_{ij} = 0$, pour tout i et j tels que $1 <= i <= n$, $1 <= j <= n$ et $i > j$

. $M_j = F'_j$, pour tout j tel que $1 <= j <= n$

Postcondition : . $M_j = I_j$, pour tout j tel que $1 <= j <= n$

For i := n **downto** 1 **do**

$$M_i := (M_i - \sum_{j=i+1}^{n} A_{ij} * M_j) / A_{ii}$$

Endfor

Notons que, pour des raisons d'économie de place en mémoire, seulement une matrice et un vecteur sont physiquement réservés (A et M).

4.1.2 Matrices bandes

Comme le montre la figure 4.0, une matrice bande de largeur de bande n est une matrice dont les éléments non nuls sont localisés à au plus une distance de n éléments des éléments appartenant à la diagonale principale. Tous les éléments de la matrice qui sont hors bande sont nuls.

Une caractéristique de toutes les matrices manipulées par les programmes d'éléments finis, et qu'il est important de connaître pour la suite, est qu'ils manipulent des matrices bandes. On peut donc considérer par la suite que la matrice du système à résoudre est bandée, de largeur de bande connue, et stockée sur fichier.

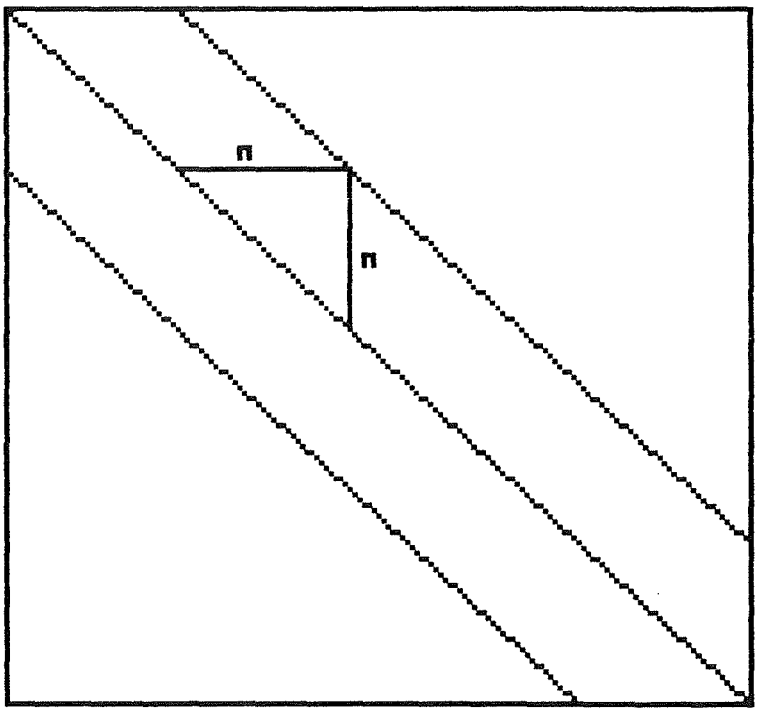


fig 4.0 Matrice bande

4.2 Implémentations envisageables

Nous avons envisagé quatre solutions différentes pour améliorer les performances des programmes d'éléments finis. Les trois premières solutions sont basées sur la même idée de division de la matrice des coefficients du système en sous-matrices. La première solution se base également sur une matrice d'indicateurs associée à la matrice divisée en sous-matrices. La seconde solution décrit les algorithmes qui travaillent uniquement sur une matrice divisée en sous-matrices, la troisième solution implémente une division de matrice en pointeurs vers des sous-matrices. Finalement, la quatrième solution, un peu différente des trois premières, envisage la gestion d'une seconde matrice, transposée de la matrice des coefficients.

4.2.1 Matrice d'indicateurs

Pour cette première solution, nous envisageons d'utiliser deux structures de données pour résoudre notre problème. La première structure est une matrice de sous-matrices d'éléments que nous avons décrite au chapitre 3, et qui sera utilisée pour représenter le système d'équations. On associe à cette matrice du système une matrice d'indicateurs. Il y a un indicateur par sous-matrice. Cet indicateur sera vrai s'il existe dans la sous-matrice correspondante un élément non nul, et sera faux si tous les éléments de la sous-matrice correspondante sont nuls. Cette solution permettrait de gagner un temps assez important de calcul. En effet, dans les algorithmes de résolution, il suffirait de tester l'indicateur de la sous-matrice. Si celui-ci indique que tous les éléments de la sous-matrice sont nuls, l'algorithme passe à la sous-matrice suivante. Par contre, si l'indicateur signale que la sous-matrice contient des éléments non nuls, on applique l'algorithme de résolution à tous les éléments de la sous-matrice. Notons que cette solution complique les algorithmes

de résolution, puisqu'il faut tester un indicateur à chaque sous-matrice, et prendre la décision adéquate.

Toutefois, si cette solution paraît offrir un avantage par rapport à une solution classique, elle ne représente certainement pas encore un optimum. Nous savons que la matrice est bandée, de largeur de bande connue. Cela signifie donc qu'il est possible de localiser les éléments non nuls de la matrice : ils se trouvent dans la bande; hors bande, les éléments sont tous nuls. La matrice des indicateurs s'avère donc inutile, puisqu'on sait quelles sont les sous-matrices qui contiennent les éléments de la bande, et quelles sont celles qui n'en contiennent pas. Cette solution de la matrice d'indicateurs a donc été très rapidement abandonnée au profit d'une solution plus simple et plus performante encore, qui consiste à avoir comme seule structure de données pour représenter la matrice des coefficients du système d'équations une matrice de sous-matrices d'éléments

4.2.2 Matrice de sous-matrices d'éléments

Cette solution consiste donc à appliquer un algorithme de résolution à la matrice de sous-matrices d'éléments, en veillant à ne pas sortir de la bande, puisque tout calcul en dehors de celle-ci est inutile.

Algorithmes

Soit LB la largeur de bande de la matrice du système.

```
1° For l := 2 to n do
      For i := l - LB + 1 to l - 1 do
          For j := l - LB + 1 to i - 1 do
              A(l,i) := A(l,i) - A(l,j) * A(j,i);
              A(i,l) := A(i,l) - A(i,j) * A(j,l);
          Endfor;
          A(l,i) := A(l,i) / A(i,i)
      Endfor;
      For j := l - LB + 1 to l - 1 do
          A(l,l) := A(l,l) - A(l,j) * A(j,l)
      Endfor
  Endfor                                          (4.3)
```

Comme le montre la figure 4.2, les instructions les plus à l'intérieur des boucles ont pour effet de :

- mettre à jour la ligne l de L ($A_{l,i} := A_{l,i} - A_{l,j} * A_{j,i}$)
- mettre à jour la colonne l de U ($A_{i,l} := A_{i,l} - A_{i,j} * A_{j,l}$)
- normaliser la ligne l de L ($A_{l,i} := A_{l,i} / A_{i,i}$)
- mettre à jour le terme diagonal $A_{l,l}$ ($A_{l,l} := A_{l,l} - A_{l,j} * A_{j,l}$)

On peut formuler deux remarques concernant cet algorithme :

- il parcourt les colonnes de la matrice. Pour améliorer les performances de cet algorithme, il conviendrait donc d'essayer que ce parcours des colonnes se fasse sans provoquer trop de défauts de page;

- on peut déterminer, à chaque itération de la boucle la plus extérieure de l'algorithme, le nombre d'éléments référencés par celui-ci. En effet, l'algorithme ne sortant jamais de la bande, il référence une partie de matrice hachurée dans la figure 4.2. On s'aperçoit donc que l'algorithme référence une partie carrée de matrice, dont le nombre d'éléments est exactement le carré de la largeur de bande, et cela à chaque itération de la boucle la plus extérieure de l'algorithme.

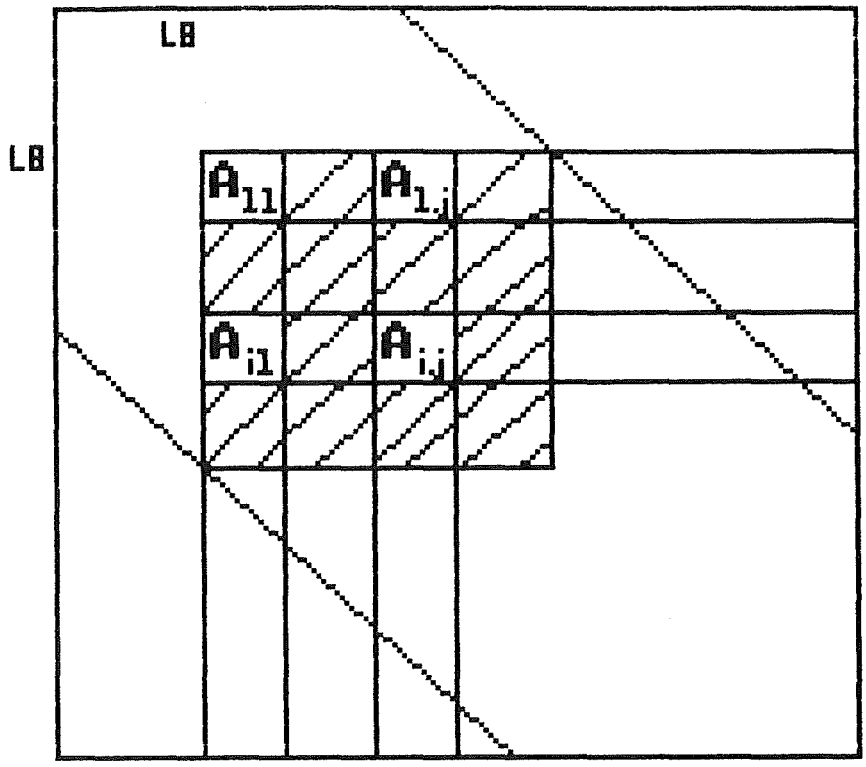


fig 4.1 Elements references par l'algorithme 4.4

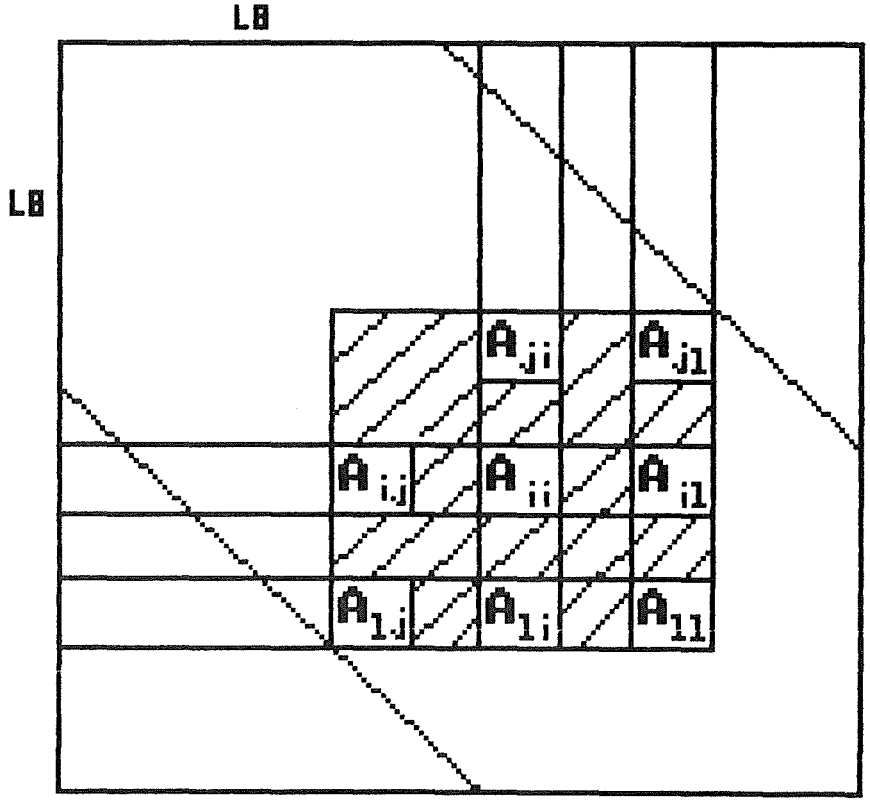


fig 4.2 elements references par l'algorithme 4.3

```
2° For l := 1 to n - 1 do
       For i := l + 1 to l + LB - 1 do
           A(i,m) := A(i,l) / A(l,l);
           For j := l + 1 to l + LB - 1 do
               A(i,j) := A(i,j) - A(i,l) * A(l,j)
           Endfor
       Endfor
   Endfor                                          (4.4)
```

Comme le montre la figure 4.1, l'instruction $A_{i,l} := A_{i,l} / A_{l,l}$, met la colonne l de l'itération courante à jour, et un bloc carré de matrice sous le $A_{l,l}$ courant est également mis à jour ($A_{i,j} := A_{i,j} - A_{i,l} * A_{l,j}$).

On peut également formuler les deux mêmes remarques concernant cet algorithme :

- Il parcourt les colonnes de la matrice. Pour améliorer les performances de cet algorithme, il conviendrait donc d'essayer que ce parcours des colonnes se fasse sans provoquer trop de défauts de page;

- on peut déterminer, à chaque itération de la boucle la plus extérieure de l'algorithme, le nombre d'éléments référencés par celui-ci. Comme le montre la figure 4.1, et par le même mécanisme de l'algorithme précédent, il référence un nombre d'éléments égal au carré de la largeur de bande.

Rappelons avant tout que la base de ces solutions est le fait que toute la matrice des éléments réside en mémoire virtuelle, ce qui permet d'éviter des accès disques systématiques et explicites dans le programme, en les rendant implicites et en fonction des besoins par le mécanisme de gestion de la mémoire virtuelle. Il faut donc que toute la matrice tienne en mémoire virtuelle. Par exemple, pour stocker une matrice de 2000 X 2000 doubles, il faudrait 250 * 250 sous-matrices, soit 62500 pages en mémoire virtuelle. Nous savons également que une page virtuelle se trouve physiquement soit en mémoire centrale (dans le working

set du processus), soit sur disque dans le paging file. La taille du working set, pour un utilisateur normal, est en général limitée à quelques centaines de pages; il faudrait donc que le paging file accueille le reste des pages virtuelles, c'est-à-dire, dans notre exemple, environ soixante mille pages.

Malheureusement, sur VAX/VMS, la taille du paging file est fixée, pour un utilisateur normal, à environ vingt-cinq mille pages. On s'aperçoit donc que la taille de la mémoire virtuelle qui peut être effectivement utilisée est limitée par la taille du paging file.

Une deuxième limitation quant à la taille de la mémoire virtuelle, réside dans le fait que l'espace d'adressage (en mémoire virtuelle) est également limité pour chaque processus; Cette limitation semble être un paramètre du système identique pour chaque utilisateur.

Nous avons cru, au départ et pendant un très long temps, que ces problèmes n'en étaient pas. Nous pensions que lorsque nous déclarions une matrice B : **array** [0..1999,0..1999] **of double**, aucune place n'était réservée en mémoire centrale, ni sur disque. Les pages virtuelles de cette matrice auraient été créées dynamiquement, c'est-à-dire au moment de la référence à cette page. Pour notre problème, nous pensions donc que seules les pages contenant des éléments de la bande auraient été physiquement créees lors de l'accès à ces éléments, les autres n'étant jamais générées, puiqu'elles contiennent des éléments hors bande nuls, donc n'étant jamais référencés. Comme la largeur de bande est relativement réduite par rapport à la taille de la matrice (environ 10 %), nous pensions pouvoir, de cette manière, stocker toute la matrice en mémoire virtuelle.

Nous nous sommes donc aperçus, très tard, que cela n'était pas exact. En effet, il est apparu que lors de la déclaration d'une matrice dans un bloc de programme PASCAL, toutes les pages nécessaires au stockage de cette matrice étaient créées lors de l'entrée dans ce bloc, surchargeant le paging file.

Nous sommes donc arrivés à la conclusion que cette solution était très restrictive quant à la taille des problèmes qu'elle permet de résoudre : elle ne permet pas de manipuler des systèmes de plus de 1200 équations.

La solution à ce problème qui consiste à augmenter la taille du paging file n'est pas très réaliste. En effet, lorsque la taille de la matrice augmente, le nombre de pages nécessaires pour stocker tous les éléments de la matrice augmente plus que proportionnellement, ce qui fait qu' une légère augmentation de la taille de la matrice requiert une augmentation considérable de la taille du paging file. De plus, cette solution ne résout rien, puisque la taille de la mémoire virtuelle est quand même limitée pour chaque utilisateur du système.

Cette solution ne permettant donc de résoudre qu'un système avec un nombre limité d'équations, nous avons donc dû en imaginer une autre, car les programmes d'éléments finis manipulent des systèmes de taille très importante, et en tout cas supérieure à celle des systèmes que cette méthode permet de résoudre.

4.2.3 Matrice de pointeurs vers des sous-matrices d'éléments

Il faut avant tout signaler que cette solution n'a pas été réellement implémentée, parce que nous nous sommes aperçus trop tard dans l'année que la solution précédente était inopérante. Cependant, elle mérite d'être développée d'un point de vue plus théorique.

Nous avons donc imaginé une solution qui nous permettrait d'amener toute la matrice du système d'équations à résoudre en mémoire virtuelle. Il fallait, pour que cela soit possible, que les pages qui correspondaient à des sous-matrices ne contenant que des éléments nuls ne soient pas physiquement

créées. Nous avons donc décidé de gérer nous même la création des sous-matrices, au lieu de laisser cette gestion au système.

Cette solution repose sur une matrice de pointeurs vers des sous-matrices d'éléments. On fera donc les déclarations PASCAL suivantes :

```
TYPE    Submat     = array [0..7,0..7] of double;
        Ptr_submat = ^Submat;
        Mat        = array [0..99,0.99] of Ptr_submat;
VAR     B : Mat;
```

La seule différence par rapport à la solution précédente est l'utilisation d'un pointeur par lequel il faut passer pour obtenir une sous-matrice particulière d'éléments.

L'avantage est que désormais, nous sommes responsables de la création des pages. C'est donc dynamiquement, à l'exécution, et par des instructions se trouvant dans nos programmes, que nous créerons les pages. C'est lors de la lecture de la matrice sur fichier, que nous créerons les pages qui contiennent au moins un élément non nul; les autres pages ne seront pas générées. Le système, quant à lui, réservera de la place pour la matrice de pointeurs, mais cette place est négligeable. Notons que l'algorithme de lecture de la matrice sur disque devient nettement plus complexe, puisque c'est lui qui est chargé de cette génération des pages. On en arrive donc, par cette méthode à ne réserver en mémoire virtuelle que les pages nécessaires pour stocker les éléments non nuls de la matrice; les autres pages ne sont pas créées.

On peut donc espérer par cela un gain de place tel qu'il nous permettrait de résoudre des systèmes plus importants, et cela avec des performances bien meilleures que celles des programmes d'éléments finis. Cependant, cela reste à démontrer, et c'est ce que tentera de faire la section 4.3.

4.2.4 Matrice transposée

Cette dernière solution repose sur l'idée de la gestion d'une seconde matrice, transposée de la matrice des coefficients du système. En fait, la matrice des coefficients serait représentée par un ensemble de tableaux, un tableau représentant une ligne de la bande. De ce fait, seule la place réellement nécessaire pour les éléments non nuls de la matrice serait réservée en mémoire. On réserverait de la même manière de la place pour une seconde matrice, transposée de la matrice des coefficients. De cette manière, lorsque les algorithmes de résolution effectuent un parcours des éléments colonne par colonne, ils l'effectuent dans la matrice transposée, ce qui revient à un parcours ligne par ligne de cette matrice. De cette manière, on évite tout parcours colonne par colonne, et on améliore grandement les performances des algorithmes. Notons d'ailleurs que ces algorithmes ne sont certainement pas triviaux.

4.3 Mesures de performances et conclusions

Nous présenterons, dans cette section, la manière dont les tests ont été effectués, les résultats de ces tests ainsi que certaines conclusions qu'ils nous ont permis de tirer.

Dorénavant, on appellera algorithme optimisé un algorithme de décomposition LU 4.3 ou 4.4 travaillant sur une représentation optimisée de la matrice des coefficients, c'est-à-dire une matrice divisée en sous-matrices. On appellera algorithme classique un algorithme 4.3 ou 4.4 travaillant sur une structure matricielle habituelle.

Des tests ont été menés à trois niveaux. A un premier niveau, il était intéressant de comparer les temps d'exécution des algorithmes optimisés et des algorithmes classiques. Au second niveau, il était également intéressant de comparer les performances obtenues au premier niveau avec les performances

des programmes d'éléments finis. Finalement, nous exposerons les raisons qui rendent la quatrième solution (matrice transposée), inconcevable pour les programmes d'éléments finis.

4.3.1 Algorithmes optimisés - Algorithme classique

Les premiers résultats intéressants proviennent d'une comparaison entre les algorithmes optimisés et un algorithme classique de résolution. Les algorithmes optimisés sont les algorithmes de décomposition L U 4.3 ou 4.4 travaillant sur une matrice divisée en sous-matrices d'éléments. Ces deux algorithmes essayent donc de minimiser le nombre de défauts de page pour optimiser la vitesse d'exécution. Suivant l'avis des personnes directement concernées, nous n'avons pas implémenté la méthode d'élimination de Gauss pour résoudre le système. Essentiellement par manque de temps et parce qu'il nous sera possible de supputer ces résultats par la suite, nous n'avons pas testé ces mêmes algorithmes travaillant sur une matrice de pointeurs vers des sous-matrices d'éléments. L'algorithme dit "classique" est constitué de l'algorithme 4.3 travaillant sur une structure matricielle habituelle.

Les tests ont été menés toutes choses étant égales par ailleurs sur un VAX 780. Les matrices utilisées étaient identiques pour les différents algorithmes. La largeur de bande a été choisie comme représentant 10% de la taille de la matrice; cela se rapproche assez bien de la largeur de bande des matrices traitées par les programmes d'éléments finis. On trouvera en annexe les résultats chiffrés et graphiques de ces tests. Ces résultats contiennent des mesures du temps CPU, du temps écoulé et du nombre de défauts de page réalisés par les différents algorithmes, en fonction de la taille de la matrice à traiter et de la taille du working set alloué au programme. On présente les résultats des tests pour des systèmes de 200, 400, 600, 800 et 1000 équations, et pour des working set de 100, 125, 150, 175, 200, 250, 300, 400 et 500 pages.

Nous résumerons ici ces résultats, et nous tenterons de les expliquer (rappelons que l'on trouvera des résultats graphiques en annexe sous l'intitulé : Résultats graphique des tests). On peut résumer ces résultats en quelques points :

1° les algorithmes 4.3 et 4.4 ont des temps d'exécution a peu près identiques, avec cependant un léger avantage à l'algorithme 4.3. C'est d'ailleurs ce qui nous a amené à ne pas présenter en annexe les mesures de performances de l'algorithme 4.4.

2° pour chaque taille de matrice, on remarque que, pour des petites tailles de working set, l'algorithme optimisé est toujours le plus rapide, alors que pour des tailles de working set plus élevées, c'est l'algorithme classique qui est le plus rapide. Cette inversion des tendance se manifeste, pour des matrices de 400, 600, 800 et 1000, à des tailles de working-set environ égales respectivement à 100, 150, 200, 250 pages .

3° pour une matrice de taille 200, l'algorithme classique est plus rapide que l'algorithme optimisé, quelle que soit la taille du working set .

4° lorsque, pour des petites tailles de working set, l'algorithme optimisé est plus performant que l'algorithme classique, il réalise également énormément moins de défauts de page.

5° lorsque, pour des grandes tailles de working set, l'algorithme classique devient plus rapide que l'algorithme optimisé, l'algorithme classique réalise toujours néanmoins un nombre de défauts de page légérement supérieur à celui réalisé par l'algorithme optimisé.

Avant de poursuivre notre étude des résultats, deux remarques peuvent être formulées :

1° Les tests ont été effectués sur des programmes qui utilisaient comme structure de données une matrice de sous-

matrices d'éléments. Nous savons que, avec cette structure de données, toutes les pages des sous-matrices de la matrice sont réservées en mémoire virtuelle; ceci explique pourquoi les tests n'ont pu être réalisés que sur des systèmes ne dépassant pas un millier d'équations.

2° Si nous avions effectué les tests sur des programmes utilisant comme structure de données une matrice de pointeurs vers des sous-matrices d'éléments, il est établi que nous aurions pu résoudre des systèmes trois fois plus importants que ceux que nous avons traités.

Ce résultat peut être démontré assez aisément. Tout d'abord, on sait que les programmes d'éléments finis manipulent des matrices dont la largeur de bande est environ équivalente à 10% de la taille de la matrice. Soit N la taille de la matrice, on trouve donc que LB = N / 10. Calculons T, le nombre d'éléments d'une matrice de taille N.

$$T = (2 * LB * N) - LB^2$$

$$T = (2 * N^2 / 10) - (N^2 / 100), \text{ puisque } LB = N / 10$$

$$T = (20 * N^2 / 100) - (N^2 / 100)$$

$$T = 19 * N^2 / 100$$

On trouve donc qu'une matrice de taille N, de largeur de bande N / 10 contient un nombre d'éléments environ égal à $N^2 / 5$. On trouve donc également la formule $N = \sqrt{(5 * T)}$.

Les tests qui ont été menés nous ont permis de dégager une taille maximum de matrice qui peut être traitée sur le VAX dont nous disposions, du fait de la limitation de la taille de la mémoire virtuelle. On peut traiter, au mieux, une matrice de 1200 X 1200, c'est-à-dire 1440000 éléments (doubles). Notons que ce résultat expérimental se vérifie théoriquement. En effet, nous savons que le paging file est limité à quelques 25000 pages. Les éléments de la matrice étant des doubles, une page en contient 64, ce qui signifie que la mémoire virtuelle est limitée à 64 * 25000 doubles, c'est-à-dire environ 1500000 éléments de matrice.

On peut donc calculer le nombre maximum d'équations traitables : N = √(5 * 1440000), c'est-à-dire environ 3000 équations.

Reprenons maintenant les résultats point par point, en émettant des hypothèses d'explication les concernant.

1° Le fait que les deux algorithmes 4.3 et 4.4 aient des temps d'exécution à peu près identiques n'est pas très étonnant en soi. En effet, ce sont deux algorithmes de décomposition LU qui, s'ils travaillent très différemment l'un de l'autre, référencent plus ou moins à chaque itération le même nombre d'éléments. Il est donc normal que les temps d'exécution des deux algorithmes soient plus ou moins semblables.

2° Comment se fait-il qu'à partir d'une certaine taille de working set, l'algorithme classique devienne plus rapide que l'algorithme optimisé ? Prenons, pour étudier ce phénomène, le cas d'un système de 1000 équations. Pour ce système, la tendance s'inverse pour un working set de 250 pages. Nous savons que les algorithmes étudiés référencent à chaque itération, pour une matrice de 1000 X 1000, un nombre d'éléments égal au carré de la largeur de bande, c'est-à-dire dans ce cas ci 100 * 100. On peut également calculer approximativement le nombre de sous-matrices qu'ils référencent : (100 / 8 =) 13 * 13, c'est-à-dire 169 sous-matrices.

Rappelons que les (ou une partie des) données et le (ou une partie du) programme se trouvent dans le working set au moment de l'exécution de ce programme. Il est bien entendu difficile d'estimer quelle partie de programme doit se trouver en mémoire centrale au moment de l'exécution, mais si on fait l'hypothèse que dans notre cas, 80 pages de working set sont nécessaires pour contenir la partie du programme qui est couramment exécutée, on peut trouver une explication logique à ce phénomène.

On observe en effet que l'algorithme classique devient plus rapide que l'algorithme optimisé à partir du moment où il a à sa disposition un working set suffisant pour contenir la partie de programme nécessaire pour l'exécution courante et la partie des données qui sont référencées par l'itération courante. C'est cela qui fait que l'algorithme classique devient plus rapide que l'algorithme optimisé à partir de cette taille de working set. En effet, lorsque l'algorithme classique passera à l'itération suivante, c'est, à peu de chose près, le même bloc de données qui sera référencé. Comme ce bloc est tout entier en mémoire centrale, très peu de défauts de page seront nécessaires lors des calculs effectués par cette itération.

On voit donc clairement qu'à partir de cette taille de working set, l'algorithme travaillant sur une division de matrice en sous-matrices perd tout son intérêt, puisque toutes les données référencées sont de toute façon en mémoire centrale. Cet algorithme devient dès lors légèrement plus lent qu'un algorithme classique, du fait des calculs supplémentaires pour accéder aux éléments de la matrice qu'il nécessite.

On peut généraliser cette hypothèse sans mal pour les matrices représentant des systèmes de 400, 600 et 800 équations.

3° Si l'on fait les mêmes calculs pour une matrices de 200 X 200, on trouve qu'il faudrait un working set d'environ 90 (3 * 3 + 80) pages pour que l'algorithme classique devienne plus intéressant que l'algorithme optimisé. Comme les tests commencent avec un working set de 100 pages, l'algorithme classique est donc immédiatement plus rapide que l'algorithme optimisé.

4° Que se passe-t'il si l'algorithme classique ne dispose pas d'un working set suffisant pour accueillir à la fois le programme, et toutes les pages correspondant aux données ?. Lors d'une itération 1, l'algorithme va référencer un certain nombre de pages. Comme son working set n'est pas

suffisant, il va, pour acceuillir les dernières pages de données, devoir éjecter les pages les plus anciennes de son working set. Les plus anciennes pages des données vont donc être sauvées sur disques, l'itération 1 va se terminer, puis le programme va passer à l'itération 1+1. Comme à cette itération, il va référencer plus ou moins les mêmes pages que lors de l'itération précédente, Il va commencer par référencer les pages qui viennent juste d'être, à l'itération précédente, sauvées sur disque. Cela va nécessiter des défauts de page. Comme le working set est toujours entièrement utilisé, il va falloir de nouveau éjecter des pages. On éjectera de nouveau les pages les plus anciennes. Malheureusement, ces pages sont justement les prochaines pages qui vont être référencées par les calculs de l'itération 1+1, ce qui nécessitera de nouveau une série de défauts de page. Et ainsi de suite pour toute l'exécution du programme. Ce mécanisme a pour effet de faire exploser le nombre de défauts de page réalisés par un algorithme classique, et donc d'en dégrader fortement les performances d'exécution.

5° A partir du moment où l'algorithme classique dispose d'un working set suffisant, ses performances d'exécution deviennent légérement meilleures que celles de l'algorithme optimisé, et cela malgré un nombre de défauts de page très légérement supérieur. Cela s'explique par les calculs supplémentaires que nécessite l'algorithme optimisé pour accéder à un élément de la matrice. En effet, on peut rappeler que l'algorithme optimisé accède à l'élément B[I,J] de la matrice en référençant l'élément B[divtab[I],divtab[J]][modtab[I],modtab[J]], ce qui nécessite plus de calculs que la référence B[I,J] qui est faite par l'algorithme classique.

Les tests effectués nous ont donc montré qu'il est possible de calculer, pour n'importe quelle taille de matrice, le working set nécessaire à un algorithme classique pour rester plus performant qu'un algorithme optimisé. Ce calcul est le suivant :

$$(LB / 8)^{*} + 80$$

Cela fournit le working set pour une matrice de taille N, étant entendu que LB = N / 10.

Cette formule va nous permettre de trancher entre algorithme optimisé et algorithme classique quant à savoir lequel est le plus intéressant pour les programmes d'éléments finis. Si l'on prend un système de 4000 équations, il faut un working set de $(400 / 8)^2 + 80$, c'est-à-dire 2580 pages pour que l'algorithme classique reste plus performant que l'algorithme optimisé. Un utilisateur normal pouvant difficilement disposer d'un working set de cette taille, l'algorithme optimisé se révèle très intéressant quant à l'amélioration des performances de cette routine de résolutions du système d'équations.

Il est intéressant de faire une dernière remarque : pour pouvoir résoudre des systèmes plus importants, on utilise une matrice de pointeurs vers des sous-matrices d'éléments, c'est-à-dire que l'on introduit un niveau d'indirection supplémentaire pour accéder à un élément de la matrice. Il est donc très probable que les algorithmes seront moins performants par rapport aux mesures présentées en annexes. Cependant, cette perte de performance, bien que nous n'en ayons pas la preuve puisque aucun test n'a été effectué, ne sera vraisemblablement pas très importante.

4.3.2 Comparaison avec les programmes d'éléments finis

On peut rappeler qu'un des objectifs fixés au départ était de comparer les procédures de résolution que nous allions écrire avec les routines existantes dans les programmes d'éléments finis du point de vue des performances d'exécution. Il nous fallait donc comparer nos algorithmes, qu'ils soient optimisés ou classique, avec les routines de résolution déjà intégrées dans les programmes d'éléments finis. Il est apparu que la seule façon d'effectuer cette comparaison, est

d'exécuter, toutes choses étant égales par ailleurs, le programme existant, puis ce même programme dans lequel on a substitué la routine de résolution du système d'équations par une de nos procédures de résolution. Malheureusement,

1° le programme de calcul d'éléments finis sujet à notre étude a été conçu il y a 15 ans déjà. Les commentaires sont rares, voir quasi inexistants, et les variables non significatives (3 lettres le plus souvent). Il semble également qu'il n'y ait personne dans le service responsable de ce programme qui en ait une connaissance suffisamment approfondie pour nous être utile.
2° le programme fait référence à des notions assez avancées de mécanique, matière dans laquelle nos connaissances sont assez limitées.
3° la comparaison ne pouvant se faire que sur des matrices identiques, l'utilisation du programme nécessite l'apprentissage préalable d'un autre programme qui permet de générer les données nécessaires à son exécution.

Ces trois raisons nous ont mis dans l'impossibilité de comparer nos procédures avec les routines existantes. Il nous est donc impossible de montrer, preuve chiffrée à l'appui, que nos procédures sont les plus rapides. Cependant, le bon sens nous amène infailliblement à penser que nos procédures, qui utilisent le mécanisme de mémoire virtuelle avec tous les avantages que ce mécanisme apporte, seront plus performantes que des routines qui opèrent explicitement et systématiquement des accès disque pour charger des blocs de matrice en mémoire centrale. Nous en sommes, en tout cas, fermement convaincus.

4.3.3 Matrice transposée

Rappelons que cette solution est basée sur l'idée d'avoir une seconde matrice, transposée de la matrice des coefficients, en mémoire centrale. De cette manière, l'accès colonne par colonne se fait dans la matrice transposée, et ne provoque donc plus autant de défauts de page.

Cette solution présente l'inconvénient de doubler la place mémoire nécessaire pour résoudre le système. Or, nous savons que la mémoire est une ressource critique. En effet, jusqu'à présent, nous savons que au mieux (c'est-à-dire en ne réservant en mémoire que la place strictement nécessaire pour stocker les éléments de la matrice à résoudre), on peut traiter des systèmes allant jusqu'à 3000 équations. Cette solution, en doublant la taille de mémoire nécessaire, réduit fortement la taille des systèmes que l'on peut résoudre, et n'arrange donc en rien le problème des programmes d'éléments finis qui traitent couramment des systèmes de cinq mille équations.

Conclusions

Les programmes d'éléments finis sont des logiciels destinés à étudier le comportement de milieux continus. Ils se subdivisent pour accomplir cette tâche en plusieurs sous-systèmes. Un de ces sous-systèmes, une routine de résolution d'un système d'équations, se révèle critique pour ces programmes du point de vue de ses performances d'exécution. Vu la taille considérable du système à résoudre par cette routine et la capacité limitée de mémoire centrale disponible sur les ordinateurs à l'époque où les premiers programmes d'éléments finis ont été conçus (début des années 70), les programmeurs devaient prévoir des routines de résolution de la matrice du système par bloc, ce qui dégradait considérablement les performances d'exécution de cette phase.

Les machines à mémoire virtuelle offrent actuellement à l'utilisateur des mécanismes performants simulant une mémoire gigantesque.

Le mécanisme de mémoire virtuelle s'avère donc être très intéressant pour les programmes d'éléments finis en ce sens qu'il pourrait permettre d'en optimiser la routine critique. L'idée de base pour améliorer les performances de cette routine est de charger toute la matrice en mémoire virtuelle. De cette manière, on substitue des transferts disque gérés par le mécanisme de mémoire virtuelle à des transferts disque systématiques et explicites dans cette routine. Comme les mécanismes d'entrées/sorties opérant au niveau de la mémoire virtuelle sont certainement plus performants que ceux servant les demandes venant d'un utilisateur, on peut espérer augmenter sensiblement les performances de la routine de résolution du système.

On peut encore optimiser davantage cette idée de base. Il faut, pour cela, minimiser le nombre de transferts disque (défauts de page) réalisés par le mécanisme de gestion de la mémoire virtuelle pour offrir à l'utilisateur une vision d'un espace mémoire très important. On adopte, pour ce faire, une

structure matricielle particulière. Plutôt qu'une structure matricielle classique, on divise la matrice des coefficients du système en sous-matrices. Cette structure permet de diminuer fortement le nombre de défauts de page, puisque l'accès aux éléments de la matrice colonne par colonne en provoque beaucoup moins. On peut donc espérer, avec cette méthode, améliorer encore davantage les performances.

Les tests devaient donc montrer que le mécanisme de mémoire virtuelle permettait d'améliorer les performances de la routine de résolution du système d'équations, et cela à deux niveaux. D'abord, les tests montreraient que l'optimisation apportée par la découpe de la matrice en sous-matrices d'éléments s'avère payante par rapport à une structure matricielle classique. Ensuite, ils montreraient que, de cette manière, on améliore les performances des routines de résolution des programmes d'éléments finis.

Au premier niveau donc, il s'agissait de montrer que les algorithmes travaillant sur une matrice découpée en sous-matrices étaient plus performants que les algorithmes travaillant sur une structure matricielle classique. Les tests réalisés à ce niveau nous ont permis de dégager une formule qui fixe la taille du working set nécessaire à un algorithme classique pour rester plus performant qu'un algorithme optimisé. Cette formule nous a permis de montrer que pour des tailles de matrices relativement élevées, les algorithmes optimisés étaient nettement plus rapides que les algorithmes classiques.

Les tests effectués au second niveau auraient dû faire apparaître le gain de performances par rapport aux routines de résolution des programmes d'éléments finis. Cependant, à ce niveau, aucun test significatif n'a pu être mené. En effet, les programmes d'éléments finis se sont révélés incompréhensibles. De ce fait, aucune mesure de performance valable n'a pu être prise pour ceux-ci, donc aucune comparaison entre nos procédures et les routines de résolution de ces programmes n'a pu être réalisée. Seul le bon sens d'une personne avertie permet de tirer une conclusion quant à

cette comparaison : les routines de résolution des programmes d'éléments finis systématisant les transferts de bloc de matrice vers les (ou venant des) disques, devraient être moins rapides que nos procédures pour lesquelles ces transferts incombent au mécanisme de gestion de la mémoire virtuelle, et donc sont gérés de façon plus efficaces.

Si les conclusions ou propositions de conclusions formulées à ces deux niveaux sont optimistes quant aux possibilités d'améliorer les performances des programmes d'éléments finis, il reste néanmoins un problème majeur à résoudre avant d'affirmer que le mécanisme de mémoire virtuelle constitue une solution à l'amélioration de ces performances. Ce problème est posé, pour un utilisateur normal, par la limitation de la mémoire virtuelle du VAX. De ce fait, la taille des systèmes que l'on peut résoudre est limitée. Elle est limitée, au mieux, à quelques trois mille équations, alors que les programmes d'éléments finis résolvent couramment des systèmes de cinq mille équations ! Il semble donc que la mémoire virtuelle des VAX soit insuffisante pour accueillir des systèmes comparables à ceux manipulés par les programmes d'éléments finis. Le mécanisme de mémoire virtuelle ne contribue donc pas à l'amélioration des performances de ces programmes, puisqu'il en limite les possibilités et est, dès lors inutilisable pour ces programmes.

BIBLIOGRAPHIE

[Dhatt et Touzot 1981] Dhatt G. et Touzot G., "Une présentation de la méthode des éléments finis", Malouine S.A., Paris 1981.

[DEC 1982] "VAX Software Handbook", Digital Equipement Corporation, 1982.

[de Marneffe 1987] de Marneffe P.A., "Algorithmique I", notes de cours de l'université de Liège, 1986-1987.

[Knuth 1984] Knuth D. E., "Literate Programming", The Computer Journal, vol. 27, n°2, pp. 96-111, February 1984.

[Bentley 1986] Bentley J., "Literate Programming", Communication of the ACM, vol. 29, n°5, pp. 364-369, May 1986.

[Bentley 1986] Bentley J., "A literate program", Communication of the ACM, vol. 29, n°5, pp. 471-483, June 1986.

Annexes

Résultats chiffrés des tests

On présente ici les résultats chiffrés des tests réalisés pour l'algorithme 4.3 version classique et optimisée. Ces résultats contiennent des mesures du temps CPU et écoulé, et du nombre de défauts de page réalisés en fonction de la taille de la matrice à traiter, et de la taille du working set alloué au programme. Ces résultats sont présentés pour des working set de 100, 125, 150, 175, 200, 250, 300, 400 et 500 équations, et pour des tailles de matrice de 200, 400, 600, 800 et 1000 équations.

TAILLE DE MATRICE = 200, LARGEUR DE BANDE = 20

	ALGO 4.3 CLASSIQUE	ALGO 4.3 OPTIMISE
working-set	100	100
Temps CPU (Sec)	3.35	4.25
Temps écoulé (Sec)	3.38	4.27
défauts de pages	1038	411
working-set	125	125
Temps CPU (Sec)	3.25	4.24
Temps écoulé (Sec)	3.25	4.24
défauts de pages	943	364
working-set	150	150
Temps CPU (Sec)	3.21	4.19
Temps écoulé (Sec)	3.25	4.23
défauts de pages	941	298
working-set	175	175
Temps CPU (Sec)	3.18	4.16
Temps écoule (Sec)	3.20	4.17
défauts de pages	854	136
working-set	200	200
Temps CPU (Sec)	3.18	4.16
Temps écoulé (Sec)	3.18	4.17
défauts de pages	834	131
working-set	250	250
Temps CPU (Sec)	3.18	4.15
Temps écoulé (Sec)	3.20	4.15
défauts de pages	794	42
working-set	300	300
Temps CPU (Sec)	3.09	4.11
Temps écoulé (Sec)	3.09	4.12
défauts de pages	557	9
working-set	400	400
Temps CPU (Sec)	2.92	4.12
Temps écoulé (Sec)	2.94	4.13
défauts de pages	6	7
working-set	500	500
Temps CPU (Sec)	2.93	4.14
Temps écoulé (Sec)	2.94	4.26
défauts de pages	6	9

TAILLE DE MATRICE = 400, LARGEUR DE BANDE = 40

	ALGO 4.3 CLASSIQUE	ALGO 4.3 OPTIMISE
working-set	100	100
Temps CPU (Sec)	33.97	31.60
Temps écoulé (Sec)	34.22	31.74
défauts de pages	19812	1268
working-set	125	125
Temps CPU (Sec)	23.13	31.35
Temps écoulé (Sec)	23.64	31.45
défauts de pages	2692	1156
working-set	150	150
Temps CPU (Sec)	22.88	31.30
Temps écoulé (Sec)	23.19	31.41
défauts de pages	2418	1152
working-set	175	175
Temps CPU (Sec)	22.75	31.36
Temps écoulé (Sec)	23.01	31.58
défauts de pages	2305	1150
working-set	200	200
Temps CPU (Sec)	22.75	31.36
Temps écoulé (Sec)	23.28	31.73
défauts de pages	2297	1101
working-set	250	250
Temps CPU (Sec)	22.69	31.26
Temps écoulé (Sec)	22.84	31.32
défauts de pages	2225	976
working-set	300	300
Temps CPU (Sec)	22.65	31.24
Temps écoulé (Sec)	22.84	31.30
défauts de pages	2219	880
working-set	400	400
Temps CPU (Sec)	22.64	31.21
Temps écoulé (Sec)	22.70	31.57
défauts de pages	2133	742
working-set	500	500
Temps CPU (Sec)	22.57	30.99
Temps écoulé (Sec)	22.89	31.13
défauts de pages	1970	1609

TAILLE DE MATRICE = 600, LARGEUR DE BANDE = 60

	ALGO 4.3 CLASSIQUE	ALGO 4.3 OPTIMISE
working-set	100	100
Temps CPU (Min)	4.55	1.56
Temps écoulé (Min)	5.34	2.07
défauts de pages	367879	21040
working-set	125	125
Temps CPU (Min)	3.56	1.50
Temps écoulé (Min)	3.58	1.50
défauts de pages	264002	12907
working-set	150	150
Temps CPU (Min)	1.40	1.43
Temps écoulé (Min)	1.41	4.44
défauts de pages	54060	2615
working-set	175	175
Temps CPU (Min)	1.15	1.43
Temps écoulé (Min)	1.16	1.44
défauts de pages	6192	2401
working-set	200	200
Temps CPU (Min)	1.14	1.43
Temps écoulé (Min)	1.15	1.44
défauts de pages	4674	2412
working-set	250	250
Temps CPU (Min)	1.14	1.43
Temps écoulé (Min)	1.14	1.44
défauts de pages	4029	2384
working-set	300	300
Temps CPU (Min)	1.14	1.43
Temps écoulé (Min)	1.14	1.44
défauts de pages	3964	2316
working-set	400	400
Temps CPU (Min)	1.14	1.43
Temps écoulé (Min)	1.14	1.44
défauts de pages	3898	2129
working-set	500	500
Temps CPU (Min)	1.14	1.43
Temps écoulé (Min)	1.14	1.44
défauts de pages	3852	1948

TAILLE DE MATRICE = 800, LARGEUR DE BANDE = 80

	ALGO 4.3 CLASSIQUE	ALGO 4.3 OPTIMISE
working-set	100	100
Temps CPU (Min)	18.43	5.00
Temps écoulé (Min)	20.26	5.35
défauts de pages	1488190	86527
working-set	125	125
Temps CPU (Min)	19.06	5.02
Temps écoulé (Min)	20.03	5.19
défauts de pages	1473067	83803
working-set	150	150
Temps CPU (Min)	20.04	5.02
Temps écoulé (Min)	20.18	5.05
défauts de pages	1438479	80120
working-set	175	175
Temps CPU (Min)	8.21	4.18
Temps écoulé (Min)	8.24	4.20
défauts de pages	600816	42729
working-set	200	200
Temps CPU (Min)	4.08	4.04
Temps écoulé (Min)	4.10	4.05
défauts de pages	159291	5245
working-set	250	250
Temps CPU (Min)	3.00	4.03
Temps écoulé (Min)	3.01	4.04
défauts de pages	19948	3920
working-set	300	300
Temps CPU (Min)	2.55	4.03
Temps écoulé (Min)	2.56	4.04
défauts de pages	6529	3803
working-set	400	400
Temps CPU (Min)	2.54	4.03
Temps écoulé (Min)	2.55	4.04
défauts de pages	6020	3703
working-set	500	500
Temps CPU (Min)	2.55	4.03
Temps écoulé (Min)	2.56	4.04
défauts de pages	5971	3516

TAILLE DE MATRICE = 1000, LARGEUR DE BANDE = 100

	ALGO 4.3 CLASSIQUE	ALGO 4.3 OPTIMISE
working-set	100	100
Temps CPU (Min)	43.57	10.31
Temps écoulé (Min)	49.00	12.15
défauts de pages	3682345	272727
working-set	125	125
Temps CPU (Min)	48.03	10.40
Temps écoulé (Min)	50.52	11.40
défauts de pages	3656131	263149
working-set	150	150
Temps CPU (Min)	54.12	10.48
Temps écoulé (Min)	56.05	11.13
défauts de pages	2654797	251270
working-set	175	175
Temps CPU (Min)	56.06	10.52
Temps écoulé (Min)	56.55	11.04
défauts de pages	3646598	246223
working-set	200	200
Temps CPU (Min)	52.25	8.59
Temps écoulé (Min)	52.50	9.02
défauts de pages	3374205	119628
working-set	250	250
Temps CPU (Min)	10.05	8.23
Temps écoulé (Min)	10.08	8.25
défauts de pages	522794	91657
working-set	300	300
Temps CPU (Min)	6.15	7.54
Temps écoulé (Min)	6.18	7.57
défauts de pages	112549	7948
working-set	400	400
Temps CPU (Min)	5.37	7.53
Temps écoulé (Min)	5.40	7.56
défauts de pages	9181	5810
working-set	500	500
Temps CPU (Min)	5.37	7.53
Temps écoulé (Min)	5.40	7.55
défauts de pages	8469	5712

Résultats graphiques des tests

On présente ici les résultats graphiques des tests effectués pour l'algorithme 4.3 version classique et optimisée. On présente d'abord les graphiques reprenant, pour chaque taille de matrice (200, 400, 600, 800 et 1000) les temps CPU des deux algorithmes en fonction du working set alloué au programme. On présente ensuite les graphiques reprenant, pour chaque taille de matrice (200, 400, 600, 800 et 1000) le nombre de défauts de page effectués par les deux algorithmes en fonction du working set alloué.

TEMPS CPU

NEQ = 200, LB = 20

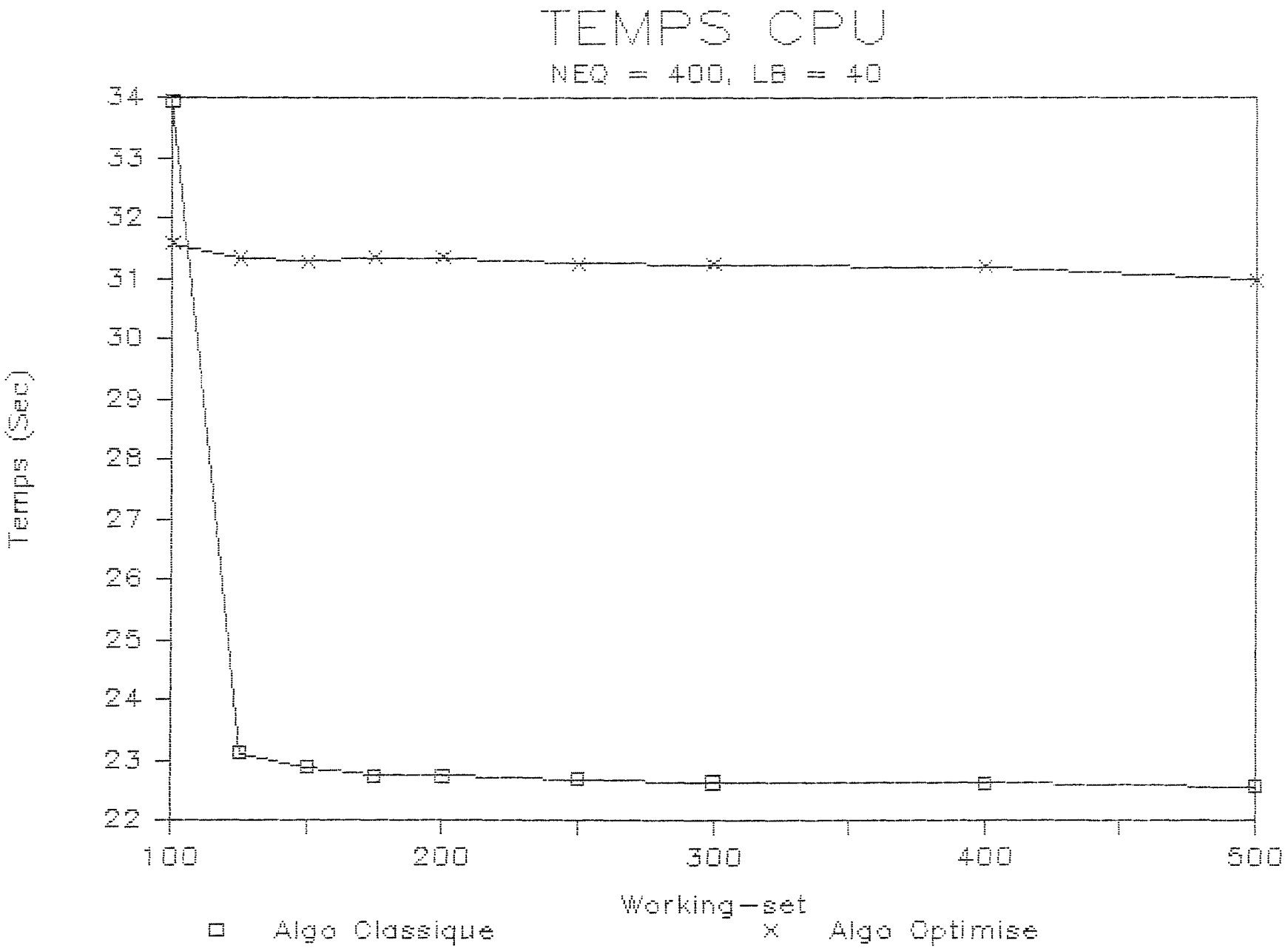
Temps (Sec)

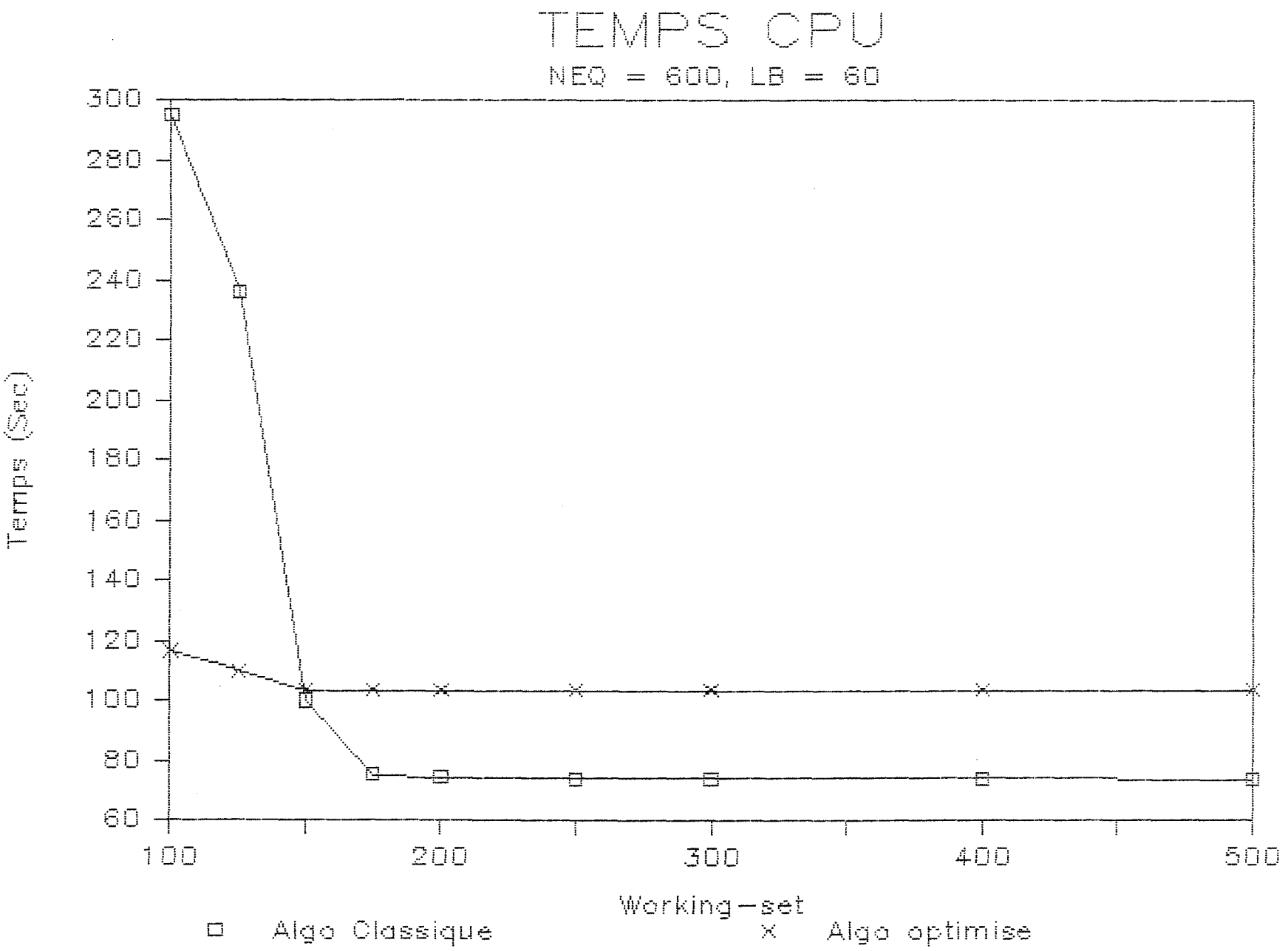
4.3
4.2
4.1
4
3.9
3.8
3.7
3.6
3.5
3.4
3.3
3.2
3.1
3
2.9

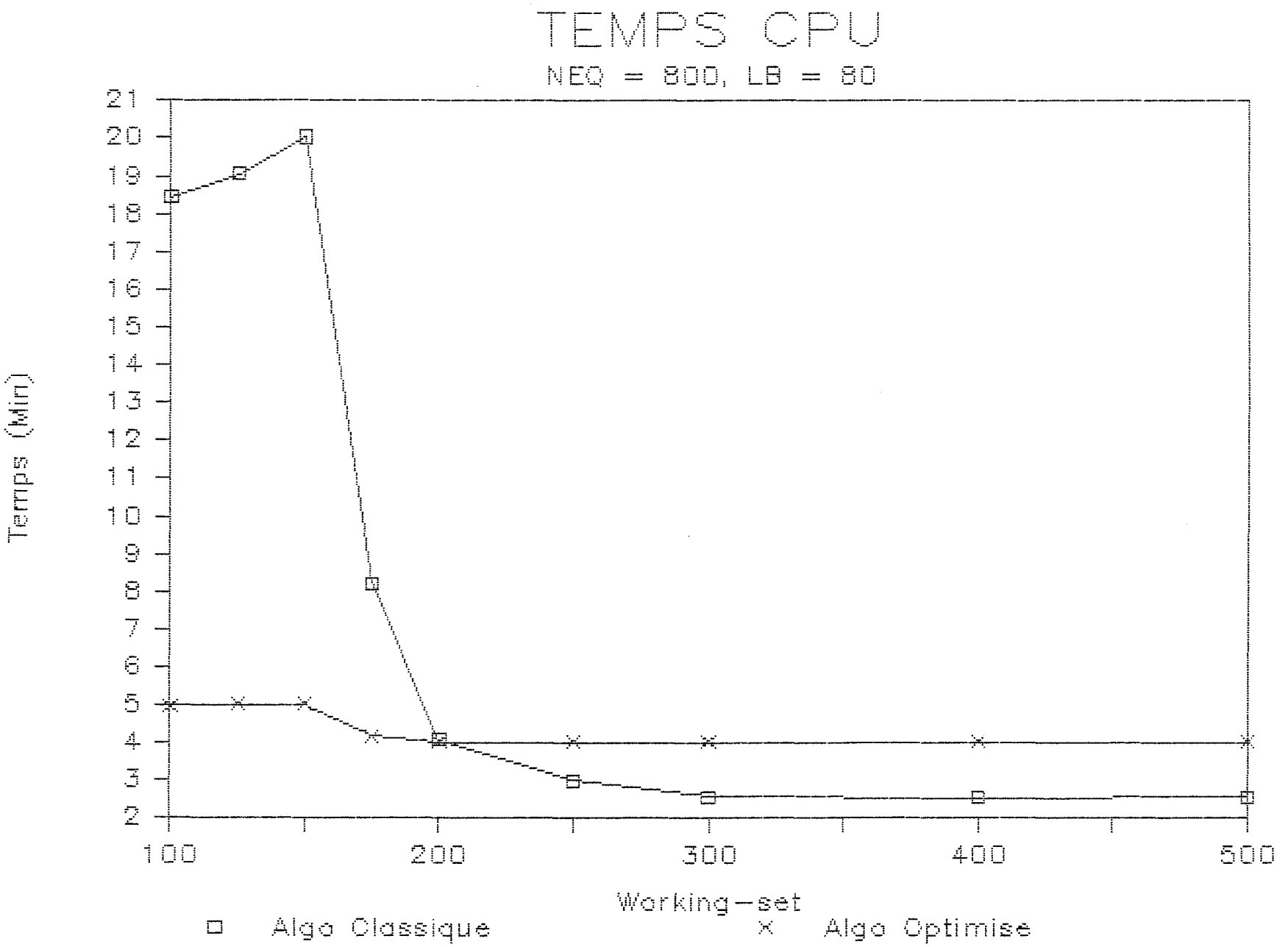
100 200 300 400 500

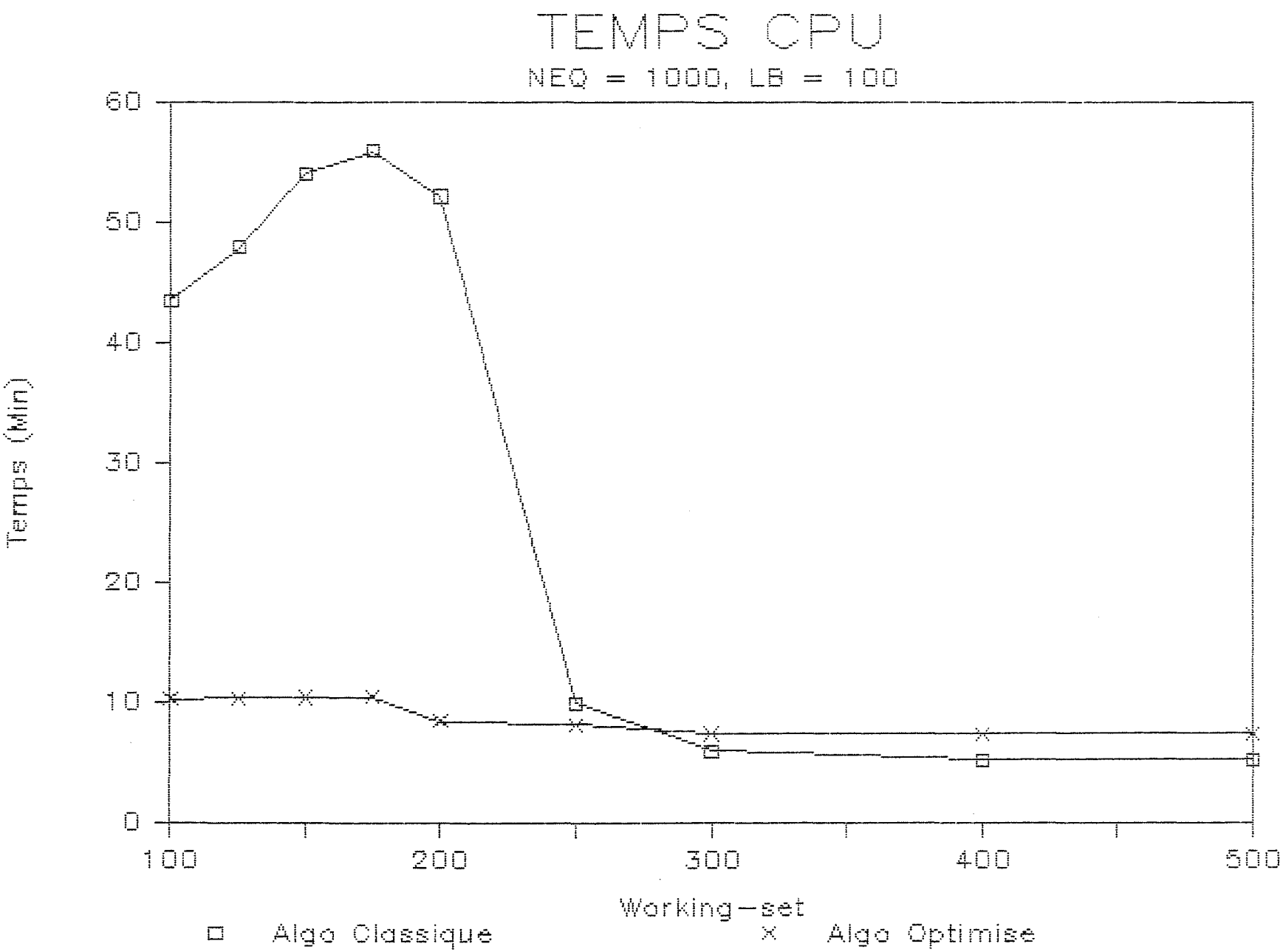
Working-set

□ Algo Classique × Algo Optimise


TEMPS CPU
NEQ = 400, LB = 40
Temps (Sec)
34
33
32
31
30
29
28
27
26
25
24
23
22
100
200
300
400
500
Working-set
Algo Classique
Algo Optimise


TEMPS CPU
NEQ = 600, LB = 60
Temps (Sec)
300
280
260
240
220
200
180
160
140
120
100
80
60
100
200
300
400
500
Working-set
Algo Classique
Algo optimise


TEMPS CPU
NEQ = 800, LB = 80
Temps (Min)
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
100
200
300
400
500
Working-set
Algo Classique
Algo Optimise

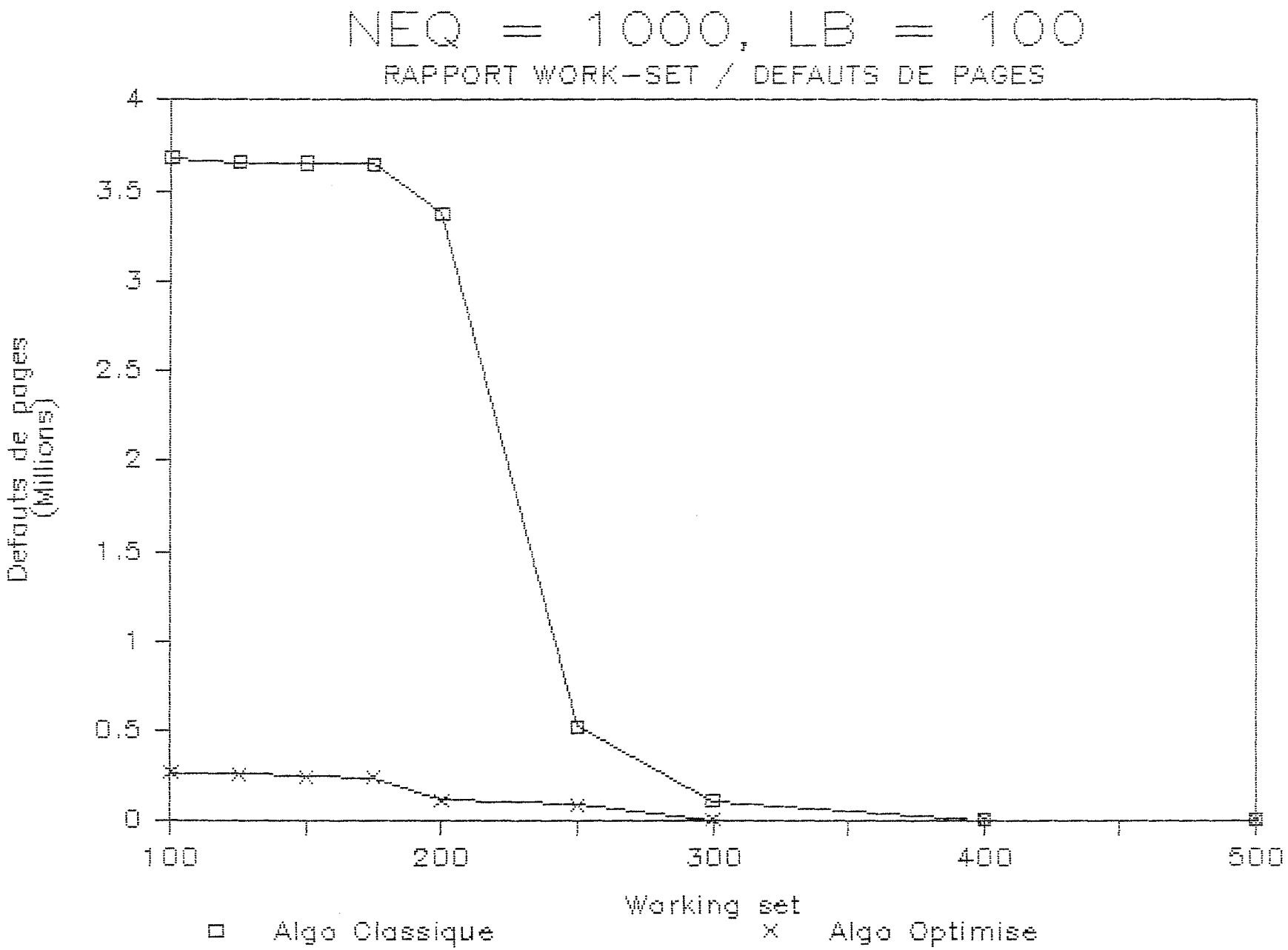

TEMPS CPU
NEQ = 1000, LB = 100
Temps (Min)
60
50
40
30
20
10
0
100
200
300
400
500
Working-set
Algo Classique
Algo Optimise

NEQ = 200, LB = 20
RAPPORT WORK-SET / DEFAUTS DE PAGES
Defauts de pages
(Thousands)
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
Working set
Algo Classique
Algo Optimise

NEQ = 400, LB = 40
RAPPORT WORK-SET / DEFAUTS DE PAGES
Defauts de pages
(Thousands)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
100
200
300
400
500
Working set
Algo Classique
Algo Optimise

NEQ = 600, LB = 60
RAPPORT WORK-SET / DEFAUTS DE PAGES
Defauts de pages (Thousands)
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
100
200
300
400
500
Working set
Algo Classique
Algo Optimise

NEQ = 800, LB = 80
RAPPORT WORK-SET / DEFAUTS DE PAGES
defauts de pages
(Millions)
2
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
Working set
Algo Classique
Algo Optimise


NEQ = 1000, LB = 100
RAPPORT WORK-SET / DEFAUTS DE PAGES
Defauts de pages (Millions)
4
3.5
3
2.5
2
1.5
1
0.5
0
100
200
300
400
500
Working set
Algo Classique
Algo Optimise