INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 653

PILOTAGE D'UN CODE MODULAIRE D'ÉLÉMENTS FINIS PAR UN SYSTÈME EXPERT

Patrick LAUG

Mars 1987

Pilotage d'un code modulaire d'éléments finis

par un système expert

The steering of a modular finite element program

by an expert system

Patrick LAUG

I.N.R.I.A.

Mars 1987

PAPIER RECUPERE ET RECYCLE

ABSTRACT

More and more engineering problems are now being tackled using numerical simulation (aeronautics, space, nuclear industry, etc). However, programs are often difficult to use and require a lot of "know-how" : validity field of the algorithms, selection of the optimal algorithm, external specifications of every module, ... The purpose of the Expert System called DOMINO is to enable the engineer to work without necessarily having a great knowledge of algorithmics, numerical analysis and computer science.

This report considers the possibilities of Artificial Intelligence adapted to numerical programs, then presents DOMINO with the help of some examples and shows the future improvements.

Keywords : *Artificial Intelligence, Expert System, Finite Elements.*

RESUME

Des problèmes d'ingénierie de plus en plus variés sont maintenant traités par simulation numérique (aéronautique, espace, industrie nucléaire, etc). Mais les logiciels mis en jeu sont souvent complexes et demandent un savoir-faire important : domaine de validité des algorithmes, choix de l'algorithme optimal, respect des spécifications externes de chaque module, ... Le Système Expert appelé DOMINO a pour but de dégager l'ingénieur de toutes ces connaissances algorithmiques, numériques et informatiques.

Ce rapport étudie les possibilités de l'Intelligence Artificielle adaptée aux codes de calculs, puis présente DOMINO à l'aide de quelques exemples et indique les extensions prévues.

Mots-clés : Intelligence Artificielle, Système Expert, Eléments Finis.

REMERCIEMENTS

Le Système Expert DOMINO a bénéficié des compétences en Eléments Finis de l'équipe MODULEF dirigée par **M. BERNADOU** à l'INRIA, en particulier **A. PERRONNET, A. HASSIM, B. MULLER, P.L. GEORGE et M. VIDRASCU.**

Ce travail a été développé dans le cadre d'un projet retenu par le **GIS/GRECO** "Calcul des Structures" en 1987.

SOMMAIRE

1. Introduction

2. Analyse de l'existant
2.1 Les systèmes experts
2.2 Les moteurs d'inférences
2.3 Les langages

3. Utilisation de DOMINO par l'ingénieur
3.1 Exemple explicatif
3.2 Sauvegardes et reprises
3.3 Les commandes

4. Mise à jour de DOMINO par l'expert
4.1 Le savoir-faire de l'expert
4.2 Mémorisation de la stratégie à appliquer
4.3 Forme informatique des jobs

5. Conclusion

Annexe A : Les commandes de DOMINO

Annexe B : Organisation de la base de règles

Annexe C : Description interne de DOMINO
C1 Le point d'entrée principal
C2 Les variables globales de DOMINO
C3 La boucle principale
C4 Le moteur d'inférence
C5 Les reprises
C6 La génération d'un job
C7 Les fonctions de lecture
C8 Les fonctions d'impression
C9 Manipulation de la base de connaissances
C10 Utilitaires pour les ensembles
C11 Utilitaires pour les vecteurs
C12 Utilitaires divers
C13 Index

Annexe D : Installation sur Apollo
D1 Portage de Le_Lisp
D2 Installation de DOMINO

Annexe E : Exemple d'utilisation
E1 Le problème
E2 Résolution avec DOMINO

Références

ABREVIATIONS USUELLES

BC Base de Connaissances
BF Base de Faits
BR Base de Règles
EH Expert Humain
IA Intelligence Artificielle
SE Système Expert

1. INTRODUCTION

L'**Ingénierie Assistée par Ordinateur (IAO)** connaît de nombreux domaines d'application : aéronautique, espace, industrie nucléaire, génie civil, biomécanique, géophysique, ... [INRIA 84]. De plus en plus, les ingénieurs sont amenés à utiliser des codes scientifiques capables de simuler une grande variété de problèmes physiques.

Cependant, du fait même de leur richesse, ces codes exigent souvent un long effort d'apprentissage pour être correctement exploités. Seuls des experts ayant une bonne expérience d'un code de calcul sont capables de répondre aux questions de l'ingénieur :

- Quelle est la méthode numérique la mieux adaptée à tel ou tel problème physique ?
- Quels sont les programmes correspondants dans le code considéré ?
- Comment mettre en œuvre ces programmes ?
- etc...

Il semble opportun, profitant des nouvelles techniques de l'**Intelligence Artificielle (IA)**, de construire un **Système Expert (SE)** faisant bénéficier les ingénieurs des connaissances des experts : les premiers disposent ainsi d'une aide permanente sur leur propre site, tandis que les seconds peuvent consacrer plus de temps aux problèmes les plus "pointus". En outre, grâce à ses capacités de raisonnement et d'explication, un SE s'avère un excellent outil d'enseignement [Bonnet 81].

Les connaissances mises en jeu paraissent bien appropriées à des manipulations par un SE, puisqu'elles sont :

- spécifiques d'un domaine bien précis,
- éparses (nombreux manuels ou expérience de plusieurs experts),
- non algorithmiques,
- évolutives.

Afin de prouver la faisabilité et les avantages d'un tel système, nous avons conçu et réalisé une maquette de SE, appelée DOMINO, qui pilote la bibliothèque d'éléments finis MODULEF [Modulef 86] et qui est présentée dans ce rapport.

La **première partie** du rapport est une analyse de différents outils de l'IA applicables au calcul scientifique (§ 2). La **deuxième partie** montre deux aspects de DOMINO : utilisation par l'ingénieur (§ 3) et mise à jour par l'expert (§ 4). La **conclusion** dresse un bilan des travaux effectués et décrit les points qui doivent maintenant être approfondis (§ 5). Enfin, les **annexes** fournissent des détails sur DOMINO, ainsi qu'un exemple d'utilisation.

2. ANALYSE DE L'EXISTANT

Avant de développer la version actuelle de DOMINO, nous avons étudié et parfois expérimenté différents outils de l'Intelligence Artificielle : *systèmes experts, moteurs d'inférences, langages.* Nous présentons dans ce paragraphe certains de ces outils et nos conclusions.

2.1 Les systèmes experts

L'intérêt sans cesse croissant pour les SE, dans le monde de l'industrie et de la recherche de nombreux pays, n'est plus à démontrer (voir en particulier [Avignon 86]). Depuis quelques années, de nombreux SE sont proposés sur le marché (maquettes, prototypes, produits commercialisés). La liste ci-dessous décrit brièvement quelques SE concernant le calcul scientifique :

- FLUX EXPERT [Massé 83] s'adresse aux numériciens qui désirent programmer la résolution d'un nouveau système d'équations aux dérivées partielles.

- MATHEXPERT [Lassalle Balier 86] choisit les méthodes et les logiciels les mieux adaptés à un problème mathématique. Ce prototype est une extension de la maquette PELAGIE [Saurel 85].

- PANDORE [INRIA 85, IMA 86] est un système expert en contrôle stochastique qui génère à la fois des programmes numériques et un rapport final (contenant les programmes générés, les graphiques obtenus, des explications sur les méthodes de résolution choisies, et des références). Le système de calcul formel MACSYMA [MIT 83] est appelé.

- SACON [Bennett 79] conseille les ingénieurs dans l'utilisation du code de calcul de structures MARC, en utilisant le moteur d'inférences EMYCIN. Son rôle est purement consultatif et aucun programme n'est généré.

- SYMATRAU [Trau 85] s'applique à la première étape de la résolution d'un problème par éléments finis : la génération du maillage.

- TITUS [Lagache 84] est un code de calcul des structures au-dessus duquel un système expert spécifique a été construit.

- Enfin, le GRECO/GIS "Calcul des Structures" [GRECO/GIS 87] a montré l'intérêt des techniques de l'Intelligence Artificielle dans ce domaine (systèmes experts, calcul formel, etc), et la maquette DOMINO a été développée dans ce cadre. Parmi les projets appartenant à ce groupement, citons un logiciel destiné à superviser les codes employés au CEA (générateur de plans TEASE).

Conclusion

Tous les SE décrits précédemment sont bien souvent **complémentaires** de DOMINO : tout en étant voisins de notre domaine, ils se distinguent par leur fonctionnalité, leur champ d'application, ou le contexte dans lequel ils sont développés.

Par exemple, FLUX EXPERT génère un programme fermé, tandis que nous prenons en compte une bibliothèque scientifique déjà programmée et testée. Des systèmes comme SYMATRAU s'intéressent à la phase bien particulière du maillage. Enfin, TEASE est développé dans le contexte industriel du CEA.

2.2 Les moteurs d'inférences

L'architecture particulière des systèmes experts permet en principe de rendre le moteur d'inférence indépendant du domaine traité. De nombreux moteurs sont disponibles, et nous avons étudié plus particulièrement ALOUETTE [Mulet-Marquis 86], GOSSEYN [Fouet 83a,b], CRIQUET [Vignard 85a,b] et SHIRKA [Rechenmann 85].

Ces moteurs ne répondent qu'à des degrés divers à nos besoins, qui sont notamment :

- une bonne compatibilité avec LISP, dont les avantages sont reconnus par les chercheurs en IA (§ 2.3) ; ce langage nous a permis d'écrire à peu de frais le générateur de programmes,
- des dialogues dans le vocabulaire des ingénieurs (explication du raisonnement, questions, etc),
- une taille mémoire et un temps de réponse raisonnables,
- une modularisation de la base de règles.

Conclusion

Après plusieurs essais infructueux, nous nous sommes résolus à développer nous-mêmes un moteur d'inférence simplifié (§ C4).

2.3 Les langages

Les langages classiques de l'IA sont LISP [Winston 84], PROLOG [Roussel 75, Colmerauer 83] ou des combinaisons de ces deux langages : LOGLISP développé à Syracuse University [Robinson 82], PROLISP de l'ONERA/CERT [Zanon 84], LISLOG du CNET [Bourgault 84], etc...

LISP, inventé par Mac Carthy (MIT) en 1962, est maintenant très répandu. Il possède en effet de nombreux atouts :
- l'interprète est interactif,
- une masse importante de logiciel a déjà été développée et de nombreux résultats ont été publiés,
- le langage a été conçu spécialement pour la manipulation d'expressions symboliques ; en outre, il se prête bien à un style de programmation fonctionnel (sans affectation) avec récursivité,
- les fonctions et les données se présentent de manière uniforme (un programme LISP peut construire un nouveau programme et l'exécuter),
- un environnement très riche est rapidement réalisable.

PROLOG est un langage plus récent, défini puis implémenté pour la première fois en 1972 par l'équipe de A. Colmerauer à l'Université de Marseille. Il se distingue des langages procéduraux classiques, car un programme PROLOG est en fait une série de *clauses de Horn* décrivant le problème à résoudre. Si ce dernier est bien spécifié, le système trouvera lui-même la solution.

Pour un concepteur de SE, ce langage est particulièrement attrayant : il paraît facile d'écrire les bases de règles et de faits sous forme de clauses PROLOG, puis de lancer le moteur d'inférence interne au langage. Toutefois, la conception de systèmes basés sur ce principe a mis en évidence certaines difficultés [Dincbas 83].

Pour notre part, nous avons tenté une première expérience avec le langage **LISLOG** : nous avons appliqué la méthode qui vient d'être énoncée (règles écrites en PROLOG), mais les clauses pouvaient appeler des fonctions LISP. Cette manière de procéder présentait certains **avantages :**
- pour le programmeur du SE, qui profitait des possibilités de LISP et de PROLOG réunies,
- pour l'utilisateur, qui pouvait poser un grand nombre de questions à partir d'un même ensemble de règles et de faits.

Toutefois, nous avons constaté plusieurs **inconvénients :**
- Dans notre application, l'expert exprime plus naturellement les règles en se basant sur un mécanisme de chaînage avant.
- L'utilisateur désire souvent prendre le moins d'initiatives possible et ne profite donc pas de tous les avantages d'un système ouvert.
- L'explication du raisonnement est soit inexploitable, soit lourde à programmer : ou bien des mécanismes généraux de trace sont utilisés, mais alors les résultats obtenus sont trop proches de la représentation interne des clauses ; ou bien des réponses aux questions "pourquoi" et "comment" sont spécialement programmées [Clark 82], mais alors il faut ajouter des variables qui sont instanciées à l'exécution, et les règles deviennent surchargées.
- De nombreuses informations sont dupliquées. Dans l'exemple donné en annexe, pour tous les modules de résolution (CHOLPC, CHOLPS, CROUPC, etc), il faut préciser que les Structures de Données sont NDL1, BDCL, TAE en Entrée et B en Sortie (*).

Conclusion

Finalement, compte tenu de la simplicité des règles que nous avions à exprimer, nous avons construit un petit moteur d'inférence écrit en LISP, qui répond aux critères que nous avions définis au § 2.2 (compatibilité LISP, dialogues, taille mémoire, temps de réponse, hiérarchisation). Il permet en outre de décrire très facilement un séquencement dans le temps : si l'étape i est terminée, alors passer à l'étape j, etc...

Le paragraphe suivant montre comment ce système est perçu par un utilisateur.

(*) Les *frames* [Minsky 75] apportent une solution originale à ce problème, avec le mécanisme d'héritage des propriétés d'un concept général. Cependant, les systèmes à règles de production sont plus répandus [Laurière 82] en raison de leur souplesse, de leur capacité d'explication, et de leur adéquation à la manière de penser des experts.

3. UTILISATION DE DOMINO PAR L'INGENIEUR

Dans sa version actuelle, DOMINO est une maquette restreinte aux problèmes thermiques stationnaires linéaires, et s'appuie sur la bibliothèque d'éléments finis MODULEF [Modulef 86]. Elle a cependant été conçue avec un souci constant de généralité et pourra être étendue à d'autres types de problèmes, voire même d'autres codes Eléments Finis.

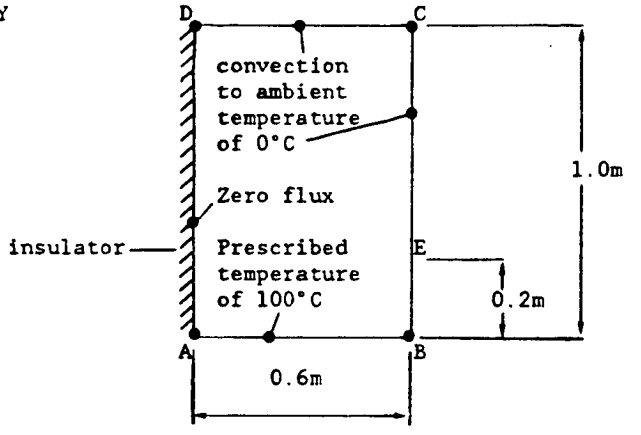
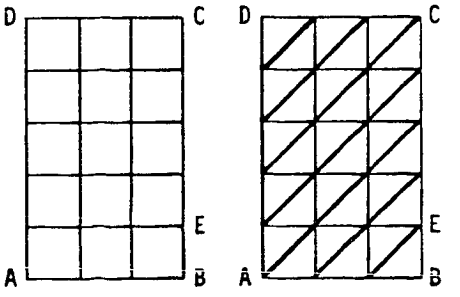
L'utilisateur fournit au départ un fichier de maillage (obtenu par un mailleur MODULEF quelconque : APNOXX, APN3XX, EMC2, ... [George 86]). Le système lui pose alors différentes questions sur le problème physique, en particulier les caractéristiques des matériaux et les conditions imposées. Pendant ce temps, le système décide d'une stratégie à appliquer, et des programmes Fortran sont automatiquement générés et exécutés. Finalement, la solution du problème est affichée sous forme de tableaux de valeurs ou de graphiques.

L'utilisateur bénéficie des caractéristiques suivantes :
- **Convivialité :** l'utilisateur est constamment guidé par le système expert, et fournit un minimum de données.
- **Justification du raisonnement :** à tout moment, il est possible d'obtenir des informations sur la base de faits, la base de règles, ou l'historique de la session.
- **Reprises :** il est facile de relancer une exécution tout en modifiant certains paramètres d'un problème, et sans redonner les autres (§ 3.2).
- **Evolutivité :** celle-ci est assurée par des experts MODULEF, afin de profiter pleinement des possibilités de la bibliothèque.

3.1 Exemple explicatif

Pour illustrer les caractéristiques que nous venons d'évoquer, considérons le test [NAFEMS 86] schématisé page suivante (un exemple industriel plus complexe est donné en annexe).

N.B. Des explications en français sont fournies page suivante.

	TWO DIMENSIONAL HEAT TRANSFERT WITH CONVECTION	TEST N° T4	DATE/ISSUE 1-7-86/1
ORIGIN	YARD (preliminary) report 3087		
ANALYSIS TYPE	Heat conduction		
GEOMETRY	 Uniform thickness		
LOADING	Zero internal heat generation		
BOUNDARY CONDITIONS	Edge AB, temperature = 100°C Edge DA, zero heat flux Edges BC, CD, convection to ambient temperature of 0°C		
MATERIAL PROPERTIES	Conductivity = 52.0W/m°C Surface convective heat transfer coefficient (edges BC, CD) = 750.0Wm2°C		
ELEMENT TYPES	Two-dimensional quadrilateral or triangular heat transfer elements		
MESHES	 Uniform mesh spacing		
OUTPUT	Temperature of point E	TARGET	18.3°C

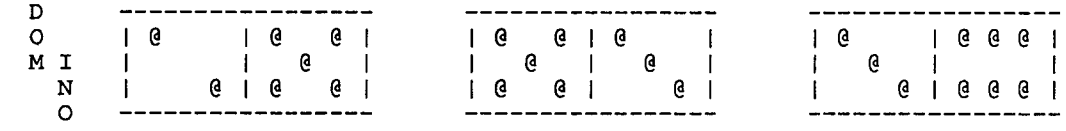
Le problème

Soit un rectangle ABCD (AB=0.6m, BC=1.0m) dont la conductivité est 52.0W/m°C. Le côté AB a une température imposée de 100°C. Le côté DA est isolé (flux de chaleur nul). Les côtés BC et CD baignent dans un fluide ayant une température ambiante de 0°C et un coefficient de transfert de 750.0W/m^2°C. A l'aide des deux maillages indiqués au bas de la page précédente, on se propose d'obtenir les courbes isothermes dans le rectangle (le test réel consiste à vérifier que la température au point E est bien 18.3°C).

Résolution du problème avec DOMINO

L'utilisateur crée tout d'abord les maillages demandés et obtient ainsi les fichiers *quad.nopo* (quadrangles) et *tria.nopo* (triangles). Il appelle ensuite DOMINO, ce qui donne lieu au dialogue ci-dessous. Chaque fois qu'il doit taper une réponse, il voit apparaître un point d'interrogation. Ce point d'interrogation et cette réponse sont imprimés ici en **gras souligné** (y compris si la réponse est vide, c'est-à-dire un simple retour-chariot).

? (:domino)




```
Version 3

Tapez ? si vous voulez obtenir la liste des commandes disponibles


Vous devez avoir cree un fichier de maillage (nopo) avec noeuds et
points confondus, a l'aide du mailleur APNOXX ou APN3XX, EMC2...
Si ce n'est pas le cas, tapez ?a pour abandonner.

Nom du fichier de maillage ? quad.nopo

Je genere et execute le job crebf

TITRE : NAFEMS
```

```
Type du probleme ?
  0 thermique  stationnaire lineaire
  1 thermique  transitoire  lineaire
  2 thermique  transitoire  non lineaire
  3 elasticite stationnaire lineaire
  4 modes propres
  5 mecanique des fluides
? 
==> thermique  stationnaire lineaire

Est-ce un probleme axisymetrique ? n

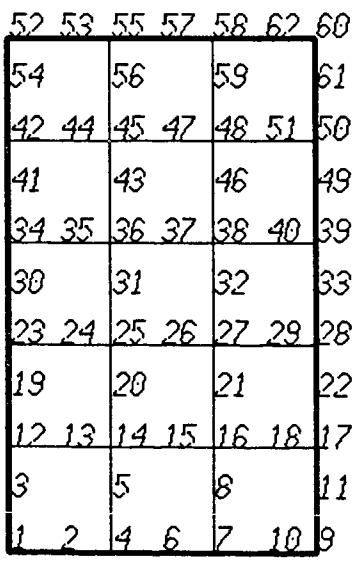

Quels resultats voulez-vous privilegier :
  0 temperatures     (elements de Lagrange)
  1 flux de chaleur  (elements mixtes)
? 
==> temperatures     (elements de Lagrange)

Choisissez le degre d'interpolation parmi (1 2)
Plus le degre est grand, plus le temps-calcul est grand, mais meilleure
est la precision.   Le degre le plus eleve est recommande : 2
? 2

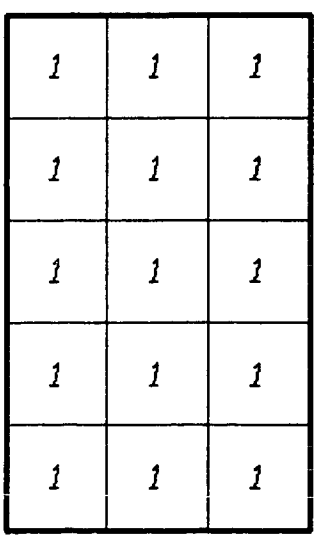
Je genere et execute le job comaco

Voulez-vous voir votre maillage apres interpolation ? o

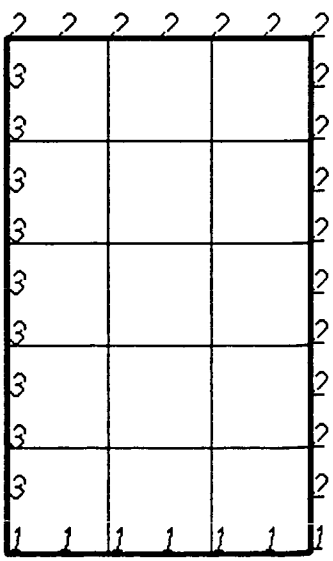
Je genere et execute le job trnoxx
```



NOEUDS



SOUS-DOMAINES



REFERENCES

```
Y a-t-il des conditions aux limites en relation lineaire ? n

-------------------------------------------------------------------------

VEUILLEZ DECRIRE LES 1 SOUS-DOMAINES DU MAILLAGE :

-------------------------------------------------------------------------

SOUS-DOMAINE NUMERO 1 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples :      1      (3 -2 5)
? 52

Valeur de la source de chaleur (W / m2) ? 0

-------------------------------------------------------------------------

VEUILLEZ DECRIRE LES CONDITIONS AUX LIMITES SUR LES 3 REFERENCES :

-------------------------------------------------------------------------

REFERENCE NUMERO 1 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
?
==> Dirichlet : temperature imposee (u_barre)

Expression en (x,y) de la temperature imposee (degre) ? 100

-------------------------------------------------------------------------

REFERENCE NUMERO 2 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 1
==> Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
Valeur du coefficient de transfert (W / m2 degre) ? 750

Valeur de la temperature du fluide exterieur (degre) ? 0

-------------------------------------------------------------------------

REFERENCE NUMERO 3 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 2
==> Neumann   : flux normal de chaleur (f_gamma)

Valeur du flux normal de chaleur (W / m2) ? 0
```

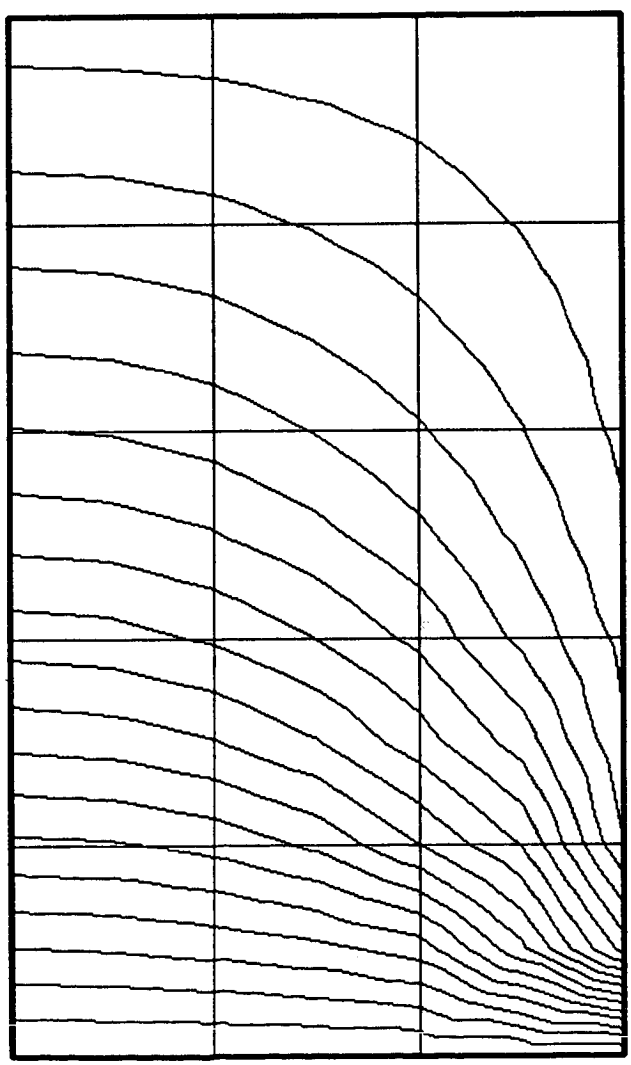
Tapez eventuellement des commandes, puis RETURN 2

Je genere et execute le job cobdcl

Je genere et execute le job valpro

Je genere et execute le job cholxx

Je genere et execute le job trmcxx



```
Aucune regle n'est applicable. Arret.

Si vous voulez reprendre plus tard cette session, tapez un tiret (-)
suivi d'une ligne de commentaires. Sinon tapez un tiret tout seul :
? -maillage forme de quadrangles
==> Je sauve cette session dans reprise.0

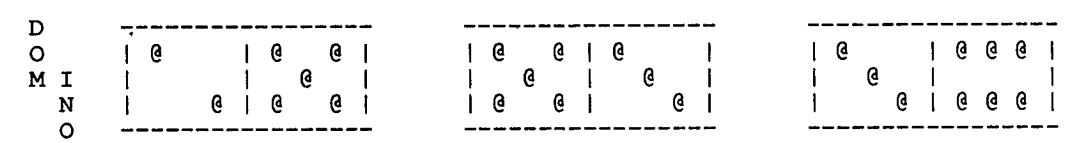
AU REVOIR !
Pour relancer DOMINO, tapez :    (:domino)
Pour sortir de LISP,  tapez :    (end)
```

==========

L'utilisateur vient de terminer la session partant d'un maillage formé de quadrangles. Il rappelle maintenant DOMINO pour traiter le cas des triangles. Le système lui propose alors de **reprendre** la session précédente, ce qui lui évite de retaper les mêmes données :

==========

```
? (:domino)
```




```
Version 3

Tapez ? si vous voulez obtenir la liste des commandes disponibles



Vous pouvez faire les reprises suivantes :

0 : quadrangles

Donnez un numero de reprise, ou tapez RETURN
si vous ne voulez pas reprendre une ancienne session :
? 0

Vous devez avoir cree un fichier de maillage (nopo) avec noeuds et
points confondus, a l'aide du mailleur APNOXX ou APN3XX, PEC...
Si ce n'est pas le cas, tapez ?a pour abandonner.

Nom du fichier de maillage [quad.nopo] ? tria.nopo

Je genere et execute le job crebf

TITRE : NAFEMS

Type du probleme ?
  0 thermique  stationnaire lineaire
  1 thermique  transitoire  lineaire
  2 thermique  transitoire  non lineaire
  3 elasticite stationnaire lineaire
  4 modes propres
  5 mecanique des fluides
[] 2
==> thermique  stationnaire lineaire
```

```
Est-ce un probleme axisymetrique [n] 2

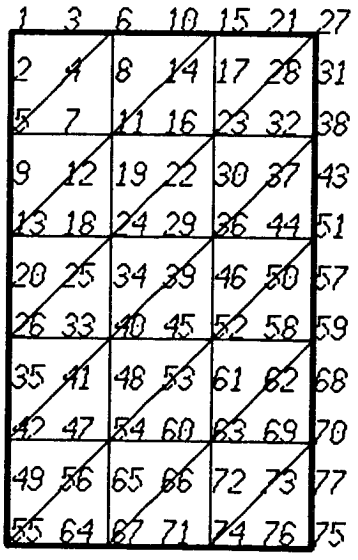
Quels resultats voulez-vous privilegier :
  0 temperatures     (elements de Lagrange)
  1 flux de chaleur  (elements mixtes)
[] 2
==> temperatures     (elements de Lagrange)

Choisissez le degre d'interpolation parmi (1 2)
Plus le degre est grand, plus le temps-calcul est grand, mais meilleure
est la precision.   Le degre le plus eleve est recommande : 2
[2] 2

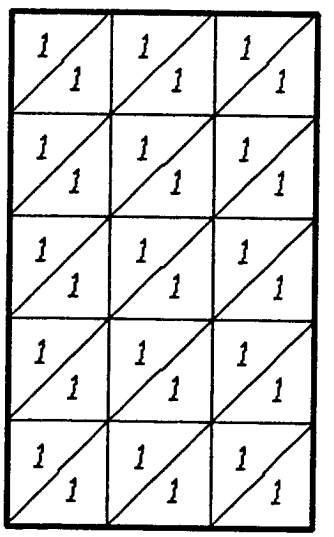
Je genere et execute le job comaco

Voulez-vous voir votre maillage apres interpolation [o] 2

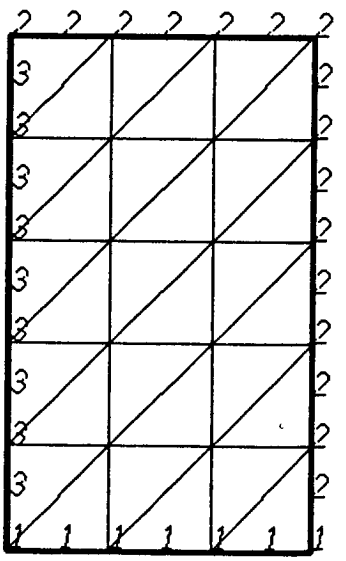
Je genere et execute le job trnoxx
```



NOEUDS



SOUS-DOMAINES



REFERENCES

La suite est analogue à la session précédente, mais l'utilisateur n'a pas besoin de retaper les mêmes valeurs numériques, qui continuent à apparaître entre crochets à la fin de chaque question. Le dialogue se termine ainsi :

```
Fin des reprises

Aucune regle n'est applicable. Arret.

Si vous voulez reprendre plus tard cette session, tapez un tiret (-)
suivi d'une ligne de commentaires. Sinon tapez un tiret tout seul :
? -maillage forme de triangles
==> Je sauve cette session dans reprise.1

AU REVOIR !
Pour relancer DOMINO, tapez :    (:domino)
Pour sortir de LISP,  tapez :    (end)

? (end)
```

L'utilisateur dispose maintenant de deux fichiers de reprise, repérables par les commentaires "maillage formé de quadrangles" et "maillage formé de triangles".

Conclusion

L'exemple précédent met en évidence plusieurs points :
- une session se présente comme un simple dialogue en langage naturel avec l'utilisateur,
- l'utilisateur ne fournit que les données strictement nécessaires,
- les données sont vérifiables (ex : conductivité > 0),
- c'est le système, et non pas l'utilisateur, qui "choisit" les bons algorithmes, "programme" les appels au code de calcul, se charge des détails informatiques (ex : la prise en compte des conditions aux limites est différente pour les éléments de Lagrange et pour les éléments mixtes),
- la reprise d'une ancienne session est très commode.

Le dernier point, concernant les *sauvegardes* et les *reprises,* est détaillé au paragraphe suivant (§ 3.2). Par ailleurs, en dehors de l'utilisation "normale" montrée ci-dessus, il est possible d'appeler différentes *commandes* qui sont présentées § 3.3.

3.2 Sauvegardes et reprises

Après avoir obtenu un premier résultat, l'utilisateur désire souvent refaire le même calcul avec des paramètres légèrement différents. De nombreux cas justifient une telle reprise de calcul, notamment :
- l'utilisateur s'est trompé dans une donnée,
- il veut étudier le comportement d'un phénomène physique lorsque certains paramètres sont modifiés (conductivité, température extérieure, source de chaleur, etc),
- il veut comparer les résultats obtenus par différentes méthodes de résolution (éléments de Lagrange ou mixtes, de degré 1 ou 2, etc).

Dans ce but, DOMINO mémorise toutes les informations nécessaires à une reprise (§ C5). Au moment de la sauvegarde, il ne demande pas à l'utilisateur un nom de fichier, mais un simple **commentaire** sur la session qu'il vient d'effectuer. A la session suivante, l'utilisateur voit apparaître en clair les commentaires de toutes les sessions mémorisées et peut en choisir une. Il n'a donc pas à gérer lui-même les fichiers de reprise.

S'il a décidé de reprendre une session, l'utilisateur se trouve dans les mêmes conditions qu'au premier appel, mais les valeurs qu'il avait données auparavant apparaîssent entre crochets. Il lui suffit alors d'appuyer sur la touche Retour pour obtenir les mêmes valeurs. Tant que le SE évolue de la même façon qu'au premier appel, les actions déclenchées par la base de règles ne sont que simulées et la session s'exécute donc bien plus **rapidement.**

Remarque

L'avantage de cette manière de procéder est l'homogénéité de la présentation : l'utilisateur se trouve toujours dans les mêmes conditions et voit apparaître les mêmes questions, avec un vocabulaire uniforme. Par contre, il risque d'être lassé d'appuyer trop souvent sur la touche Retour.

Une autre solution avait été envisagée : demander au départ le ou les paramètres à changer, puis lancer toute l'exécution sans intervention de l'utilisateur. Cette solution a été rejetée car le dialogue initial risquait de laisser transparaître la représentation interne des faits et de devenir incompréhensible pour l'utilisateur.

3.3 Les commandes

En cours de session, l'utilisateur répond habituellement aux questions posées par le système. Il peut aussi, à tout moment, taper une ou plusieurs commandes conçues pour lui fournir :
- des modes d'emploi toujours disponibles (commandes ? et ??),
- des explications sur les raisonnements du SE (?f, ?h, ?r),
- des contrôles sur l'exécution du système (?a, ?l, ?q, ?s).

Les différentes commandes de DOMINO sont présentées brièvement ci-dessous (et de manière plus détaillée § A) :

? imprime la liste des commandes disponibles et un rappel succinct de leurs rôles respectifs.
?? imprime un mode d'emploi général.
?a permet à tout moment d'abandonner DOMINO, sans faire de sauvegarde (voir aussi ?q).
?f imprime l'état courant de la base de faits.
?h imprime l'historique de la session, c'est-à-dire les principales actions effectuées. La trace des règles activées montre les raisonnements effectués par le SE.
?l permet d'évaluer n'importe quelle expression LISP, y compris un appel aux fonctions internes de DOMINO.
?q permet de quitter DOMINO moins brutalement que ?a : l'utilisateur est invité à faire une sauvegarde avant de sortir.
?r imprime la base des règles chargées en mémoire.
?s permet de faire des sauvegardes intermédiaires, en cours de session. Normalement, DOMINO ne propose cela qu'en fin de session.

Nous venons de montrer les avantages de DOMINO vu par un *ingénieur*. Plaçons-nous maintenant du point de vue des *experts* en calcul scientifique.

4. MISE A JOUR DE DOMINO PAR L'EXPERT

Nous nous intéressons ici à un expert humain **(EH)** qui connaît la méthode des éléments finis et sait utiliser un code de calcul. Le but est de transmettre les connaissances de l'EH au système expert **(SE)** DOMINO.

4.1 Le savoir-faire de l'expert

Examinons la méthode avec laquelle un EH résoud un problème d'éléments finis.

Il doit tout d'abord savoir à quelle **classe** appartient le problème :
- thermique stationnaire linéaire,
- thermique transitoire linéaire,
- thermique transitoire non linéaire,
- élasticité stationnaire linéaire,
- modes propres,
- mécanique des fluides,
- etc...

Etant donné la classe du problème, l'EH connaît les principales **étapes** à effectuer. Par exemple, pour un problème de thermique stationnaire linéaire :
- *étape 1* : maillage,
- *étape 2* : choix de l'interpolation,
- *étape 3* : définition des conductivités, des sources de chaleur, et des conditions aux limites,
- *étape 4* : création des tableaux élémentaires (rigidité, second membre, etc),
- *étape 5* : résolution du système linéaire (avec prise en compte des conditions aux limites),
- *étape 6* : visualisation des résultats.

A chaque étape, des choix se présentent (par exemple, à l'étape 2, le degré des polynômes d'interpolation détermine la précision des résultats ; à l'étape 5, de nombreuses méthodes de résolution existent, avec des variantes en mémoire centrale ou secondaire : Cholesky, Crout; Gauss, gradient conjugué, etc). Bien souvent, un choix à une étape donnée dépend des résultats des étapes précédentes.

L'EH procède donc de la manière suivante :

1. Créer le "job" qui matérialise la première étape (dans le cas du code Modulef, un job est formé d'un programme principal qui appelle des modules, éventuellement accompagné de fonctions auxiliaires et de données).
2. Compiler et exécuter ce job au moyen des commandes du système hôte.
3. D'après les résultats obtenus, choisir les modules les mieux adaptés et les appeler dans un nouveau job (retour en 1).
4. Et ainsi de suite, jusqu'à l'obtention des résultats cherchés.

Cette façon de procéder permet de profiter pleinement de la modularité d'un code. Dans le cas de Modulef, il serait maladroit de créer un job unique traitant globalement un problème en envisageant tous les cas possibles, car alors :

- le programme principal contiendrait de nombreux appels de modules inutiles à l'exécution, ce qui serait pénalisant sur la plupart des ordinateurs (l'éditeur de liens étant souvent statique),
- les paramètres de **tous** les modules devraient être fournis par l'utilisateur, y compris ceux des modules inutiles,
- les choix seraient matérialisés par des structures de contrôle Fortran, qui sont peu adaptées à ce type d'applications et rendraient le programme peu lisible (nombreux IF imbriqués),
- l'extensibilité serait très difficile à assurer.

De la méthodologie que nous venons d'évoquer, il ressort que deux niveaux de connaissances entrent en jeu :

- la **stratégie** à appliquer pour choisir les algorithmes,
- la mise en œuvre informatique de ces algorithmes, c'est-à-dire la **forme** de leurs appels.

Dans le système DOMINO, ces deux niveaux sont bien distincts, puisque le premier relève de l'IA alors que le second est beaucoup plus classique. En pratique, pour une application donnée, l'EH doit créer deux fichiers dans le "catalogue DOMINO" :

- un fichier suffixé par **ll** (suffixe traditionnel de Le_Lisp [Chailloux 85]), qui contient l'ensemble des règles définissant la stratégie,
- un fichier suffixé par **forme**, qui contient la forme informatique des jobs à exécuter.

Ces deux aspects sont présentés séparément dans les paragraphes suivants (§ 4.2 et § 4.3).

4.2 Mémorisation de la stratégie à appliquer

Comme nous venons de le voir, l'EH explicite les règles de stratégie dans un fichier suffixé par **ll**. Pour le moment, ces règles sont données directement sous leur forme interne, c'est-à-dire une expression LISP (§ B). Cette situation est naturellement **provisoire**, et un interfaçage est prévu pour que l'EH puisse manipuler la base de règles **(BR)** sans connaître LISP.

L'exemple ci-dessous donne une idée des règles telles qu'elles se présentent actuellement (une présentation plus lisible est donnée ensuite). Il s'agit de la BR initiale (commune à tous les types de problèmes), suivie d'un extrait de la BR des problèmes thermiques stationnaires linéaires :

Fichier init.ll

```
(setq :base_regles
   '(((not (:val_fait vient_de_faire))
      (ques_crebf) (:exec crebf) (ques_probleme)
      (:charger_bc (:val_fait base_connaissance)))))
```

Fichier tsl.ll

```
(setq :base_regles
   '((...)

     ((:vdf :exec cobdcl) (:exec therct))

     ((and (:vdf :exec therct))
           (eq (:val_fait taille_probleme) 'petit)
           (eq (:val_fait conditionnement) 'bien))
      (:exec ichrgc))

     (...)))
```

Explication de l'exemple

Le fichier *init.ll* contient une seule règle qui se lit :

SI - rien n'a encore été exécuté,
ALORS - demander les données du job *crebf* (qui crée la BF initiale),
- puis exécuter le job *crebf*,
- puis demander le type du problème,
- puis charger la base de connaissances correspondante.

Si l'utilisateur répond qu'il veut traiter un problème thermique stationnaire linéaire, la fonction *(:charger_bc 'tsl)* est appelée. Cette fonction permet de **modulariser** la base de connaissances (BC), puisqu'elle a pour effet de charger une autre BC, ici *tsl* (fichiers *tsl.ll* et *tsl.forme*).

Le fichier *tsl.ll* contient une série de règles, dont les deux précédentes qui signifient :

SI - l'étape "définition des conditions aux limites" vient d'être exécutée,
ALORS - exécuter l'étape "création des tableaux élémentaires".

SI - l'étape "création des tableaux élémentaires" vient d'être exécutée,
- et le problème est de petite taille,
- et le problème est bien conditionné,
ALORS - exécuter l'étape "gradient conjugué préconditionné par factorisation incomplète de Cholesky".

La première règle ne définit qu'un **ordre chronologique** entre deux étapes. Par contre, la deuxième définit aussi une **stratégie** (choix entre diverses possibilités).

Ainsi, l'EH fournit un ensemble de règles exploitées en chaînage avant. Ces règles sont **indépendantes du code utilisé,** dans la mesure où celui-ci est modulaire et contient les algorithmes nécessaires.

Il est vrai qu'un tel ensemble de règles ressemble à une "cascade de IF" d'un langage procédural classique, mais il convient de souligner certaines **différences:**
- lorsqu'il parcourt ces règles, le SE mémorise les chemins par lesquels il passe et peut ainsi expliquer ses "raisonnements",
- des techniques d'optimisation des recherches sont souvent appliquées par les SE,
- la base de règles est facilement extensible (des outils de contrôle existent) et modularisable (cf fichiers indépendants *init.ll* et *tsl.ll* dans l'exemple précédent).

La **communication entre le SE et le code de calcul** est réalisée par la fonction *:exec*, qui est appelée dans la partie "ALORS" des règles. Pour clarifier le rôle de cette fonction LISP, supposons ici que les jobs sont tous programmés en Fortran. La fonction *:exec* agit de la manière suivante :
- génération du programme Fortran dont le nom est passé en paramètre,
- compilation et exécution de ce programme Fortran, qui doit créer un fichier de nom conventionnel *interface.ll,*
- chargement pour le SE du fichier *interface.ll* (d'où lecture par un programme Lisp des informations calculées par le programme Fortran).

Chaque programme Fortran est généré en fonction de la **base de faits** et de la **forme** du job donnée par l'EH, comme indiqué dans le paragraphe suivant.

4.3 Forme informatique des jobs

Comme nous l'avons vu précédemment (§ 4.1), la forme informatique des jobs doit être décrite dans un fichier suffixé par **forme.** Ce fichier contient tous les jobs susceptibles d'être générés par la fonction *:exec* (§ 4.2).

Les jobs générés dépendent le plus souvent de valeurs présentes dans la base de faits (*). Par suite, leur création fait appel à des techniques de généricité ou de macro-expansion. Contrairement aux règles de stratégie vues précédemment, le domaine abordé maintenant est relativement **classique** et fortement **dépendant** du code de calcul.

Pour certains codes, un job est un ensemble de données fournies à un préprocesseur. Le cas de Modulef est plus général, puisqu'on trouve aussi des programmes d'appel écrits en Fortran. Pour fixer les idées, l'exemple suivant est un extrait simplifié du fichier *tsl.forme,* qui décrit la forme informatique des jobs de thermique stationnaire linéaire de Modulef :

```
-j cholpc
-d
      DIMENSION M(?LM)
-e
      CALL INITIS (M, ?LM, ...)
      NTY = ?(IF (LAGRANGE) 2 5)
      CALL PREPAC (..., NTY, ...)
      CALL ASMAPC (...)
      CALL ASMBMC (...)
      CALL CLIMPC (...)
      CALL CHOLPC (...)
      CALL DRCHPC (...)
      ... Ecriture dans le fichier interface.ll  de
      ... certaines valeurs venant d'être calculées.
      ... Ce fichier sera relu par le S.E. DOMINO
      ... pour charger ces valeurs dans la base de faits.
      END
-i
      ... Données.
```

(*) *Conséquence pratique :* la durée de vie de la base de faits étant égale à une session entière (et non un job seulement), on évite de poser les mêmes questions à l'utilisateur d'un job à l'autre.

Cet extrait décrit le job "Cholesky Profil Mémoire Centrale". Du fait de la modularité du code Modulef, un utilisateur doit habituellement écrire plusieurs appels de sous-programmes (et gérer lui-même les Structures de Données intermédiaires) :

INITIS	initialisation des *common*
PREPAC	calcul des pointeurs, nombre de mots, ... de la matrice A
ASMAPC	assemblage de A (profil, mémoire centrale)
ASMBMC	assemblage de B (mémoire centrale)
CLIMPC	prise en compte des conditions aux limites
CHOLPC	factorisation de Cholesky
DRCHPC	descente-remontée

Lorsque le job à générer dépend de la base de faits, l'EH fournit une expression commençant par un point d'interrogation. Dans notre exemple :

?LM est remplacé par la longueur du tableau M,

?(IF (LAGRANGE) 2 5) est remplacé par l'évaluation de cette expression LISP, qui vaut 2 si les éléments sont de type Lagrange et 5 sinon.

Voyons à présent de manière plus détaillée :
- comment se présente l'**organisation du fichier** qui décrit les jobs,
- comment paramétrer ces jobs : notion de **généricité**,
- comment écrire le **fichier d'interface** qui est ensuite lu par le SE.

4.3.1 Organisation du fichier

Il s'agit d'un fichier éditable divisé en plusieurs blocs. Chaque bloc est repéré par un tiret (-) en colonne 1.

Les blocs les plus globaux, qui décrivent chacun un job, commencent par le mot-clé -j :

-j <x> annonce le début du job <x>.

Ces blocs globaux sont à leur tour divisés en sous-blocs. En effet, un job est généralement formé d'un programme principal, de **"fonctions-utilisateurs"** (sous-programmes auxiliaires très courts) et de **données.** Le programme principal, lui, comprend une partie **déclarative** et une partie **exécutable.** Il est commode de différentier ces deux parties sachant que le générateur traite la partie exécutable *avant* la partie déclarative : on peut ainsi générer toutes les affectations d'un tableau et déclarer ensuite ce tableau selon la longueur obtenue (fonctions *:v_aff* et *:v_dcl*, cf § C11).

Chacun de ces sous-blocs commence par l'une des lignes spéciales suivantes :

- **-d** précède les instructions de déclaration,
- **-e** précède les instructions exécutables,
- **-f** précède les fonctions-utilisateurs,
- **-i** précède les données (input).

Par extension, la zone -e peut aussi contenir un programme LISP (qui commence nécessairement par "(" ou ";" contrairement aux programmes FORTRAN).

Enfin, pour améliorer la **présentation,** l'EH peut insérer des lignes blanches dans le texte (celles-ci sont ignorées par l'analyseur) et il dispose des deux commandes suivantes :

- **--** annonce une ligne de commentaires,
- **-b** permet de générer effectivement une ligne blanche (cas rare).

4.3.2 Généricité

Les programmes à générer dépendent presque toujours de la base de faits. Ainsi, l'EH souhaite insérer dans la *forme* des jobs un fait simple *(ex : dimension de l'espace étudié = 2 ou 3)* ou une expression plus complexe *(ex : si a < 1 alors 0 sinon sin(a+b+c), où les valeurs de a, b, c sont dans la base de faits).* Il suffit alors pour l'EH de mettre dans le fichier un point d'interrogation suivi d'une expression LISP quelconque (y compris un simple atome). Par exemple :

```
NDIM = ?dimension_espace
S = (if   (< a 1)   0   (sin (+ a b c)))
```

Si la fonction LISP retourne une liste, plusieurs lignes de texte sont générées. Par exemple, une ligne de fichier contenant :

```
?(list   "      X = 1"   "      Y = 2"   "      Z = 3")
```

est générée sur trois lignes :

```
X = 1
Y = 2
Z = 3
```

Les fonctions de génération les plus utiles sont prédéfinies, comme *:v_aff* qui affecte un tableau Fortran, *:v_dcl* qui déclare les tableaux, etc (§ C11). L'EH a bien évidemment la possibilité d'y ajouter son propre jeu de fonctions, comme cela est habituel en LISP.

Des détails sur le mécanisme permettant de passer du texte générique au texte définitif sont donnés au § C6 (fonctions *:lire_instruction* et *:imp_ins*).

4.3.3 Fichier d'interface

Pour fournir au SE les informations qui lui sont utiles, chaque programme généré (en Fortran le plus souvent) doit créer un fichier de nom *interface.ll,* qui est ensuite relu par la fonction LISP *loadfile.* Ce moyen est tout à fait portable et évite de présupposer des compatibilités entre LISP et FORTRAN (fonction non standard *defextern* en Le_Lisp).

Si le fichier *interface.ll* est absent ou ne contient pas d'appel à une fonction spéciale appelée *:continue,* le SE s'arrête. Ceci permet d'empêcher le SE de boucler indéfiniment en cas d'erreur.

Exemple de création d'un fichier d'interface

L'exemple est tiré de Modulef. L'utilitaire TRUNIT retourne un numéro d'unité de fichier, et OUVRIS ouvre ce fichier.

```
      ...
C
C OUVERTURE DU FICHIER D'INTERFACE
      CALL TRUNIT (INTERF)
      CALL OUVRIS (INTERF, 'INTERFACE.LL', 'FORMATTED', 0)
C
C CALCUL DE LLA = TAILLE DE LA MATRICE A
      ...
C
C ECRITURE ET FERMETURE DU FICHIER D'INTERFACE
      WRITE (INTERF,*) '(:AFF_FAIT TAILLE_A ', LLA, ')'
      WRITE (INTERF,*) '(:CONTINUE)'
      CLOSE (INTERF)
      END
```

5. CONCLUSION

La maquette DOMINO que nous venons de décrire est déjà utilisable par des ingénieurs qui ne connaissent pas le code Modulef. Elle prouve la faisabilité du projet et en démontre les principaux **avantages :**

- un minimum de connaissances est demandé à l'utilisateur (questions en langage naturel, réponses vérifiées),
- les questions posées par le système sont peu nombreuses et non redondantes,
- les interactions avec le code numérique sont transparentes (programmes générés et exécutés automatiquement durant la consultation, résultats intermédiaires communiqués au système expert),
- les sessions sont mémorisées en vue de reprises éventuelles,
- le système est capable de justifier ses décisions,
- la base de connaissances est modulaire et peut s'enrichir sans perturber le système existant,
- les règles de stratégie indépendantes du code de calcul sont isolées des connaissances plus "informatiques".

Ces résultats encourageants nous incitent à poursuivre notre effort dans plusieurs directions :

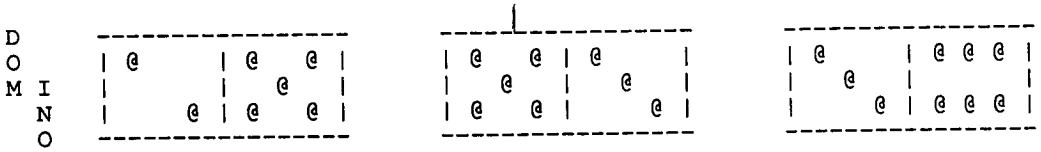
- **Perfectionner le moteur d'inférence** : le moteur actuel ne permet pas d'exprimer des règles aussi générales que PROLOG. Cependant, la modularisation de la base de règles doit être conservée.
- **Améliorer l'interface avec l'ingénieur** : ceci sera réalisé dès l'acquisition d'une version complète de Le_Lisp, qui comprend un terminal virtuel avec possibilités vidéo (effacement total ou partiel de l'écran, positionnement du curseur, mise en valeur d'une partie de texte, etc).
- **Développer l'interface avec l'expert** : l'utilisateur a volontairement été privilégié dans cette maquette, au détriment de l'expert. A terme, ce dernier doit être capable de manipuler lui-même la base de connaissances.
- **Traiter d'autres types de problèmes de l'ingénierie** : nous avons tenu à nous limiter à un domaine que les experts maîtrisent parfaitement, puisque notre but était avant tout d'approfondir les techniques de l'IA. A présent, la structure du système DOMINO permet d'ajouter rapidement de nouvelles classes d'applications.

Ces différentes actions devraient être menées à bien dans un avenir proche.

ANNEXE A : LES COMMANDES DE DOMINO

Bien que le système soit essentiellement auto-documenté, l'utilisation des différentes commandes proposées est détaillée ci-dessous (une vision plus générale de l'utilisation de DOMINO est fournie § 3).

En début de session, la bannière suivante apparaît :




```
Version 3

Tapez ? si vous voulez obtenir la liste des commandes disponibles
```

L'utilisateur répond alors aux questions posées par le système. Il peut aussi, à tout moment, utiliser l'une des commandes suivantes (cette liste est obtenue en tapant un **?** tout seul) :

```
??   mode d'emploi
?a   abandon
?f   base de faits
?h   historique de la session
?l   expression Lisp
?q   quitter
?r   base de regles
?s   sauver la session
```

Considérons à présent chacune de ces commandes.

?? imprime un mode d'emploi général :

```
DOMINO est un Systeme Expert qui permet de resoudre des Problemes
d'Ingenierie a l'aide de la bibliotheque MODULEF.
Pour cela, il genere des jobs et les enchaine les uns
a la suite des autres (comme des dominos !).

Actuellement, il est restreint aux problemes thermiques stationnaires
lineaires.

Il suppose au depart l'existence d'un fichier de maillage de degre 1,
qui peut etre genere par l'un des mailleurs APNOXX, APN3XX, EMC2, ...

Le systeme vous guide essentiellement par des menus.
Chaque menu comprend une suite d'options numerotees,
suivies par un point d'interrogation.
Vous pouvez alors taper :
- le numero d'une option ==> execution de l'action correspondante,
- RETURN ==> l'option 0 est prise par defaut,
- une commande commencant par ?
  (? tout seul en donne la liste complete).

Des reprises de sessions anterieures sont possibles. Dans ce cas,
l'ancienne valeur (imprimee entre crochets) est prise en compte
chaque fois que la touche Retour est appuyee en reponse a une question.
```

?a permet à tout moment d'abandonner DOMINO, sans faire de sauvegarde (voir aussi ?q).

?f imprime l'état courant de la base de faits. Dans la version actuelle, on obtient par exemple la liste ci-dessous. Les faits seront présentés plus clairement dans les versions futures :

```
BASE DE FAITS :

courbes = ()
dimension_espace = 2
fichiers = ((nopo . tria.nopo))
nb_references = 3
nb_sous_domaines = 1
sous_domaine_courbe = (())
sous_domaine_geometrie = ((1 . triangle))
supertab_l = 200
supertab_lmax = 200
titre = NAFEMS
vient_de_faire = (:exec crebf)
```

?h imprime l'historique de la session, c'est-à-dire les principales actions effectuées. La trace des règles activées montre les raisonnements effectués par le SE :

```
Voulez-vous :
  0 tout l'historique
  1 les dernieres actions seulement
?
==> tout l'historique

J'ai charge la base de regles : init

J'ai lu : 3

J'ai applique la regle init 1 :
SI    not (:val_fait vient_de_faire)
ALORS ques_crebf

J'ai lu : tria.nopo

J'ai affecte le fait : fichiers = '((nopo . tria.nopo))

J'ai affecte le fait : vient_de_faire = '(ques_crebf)

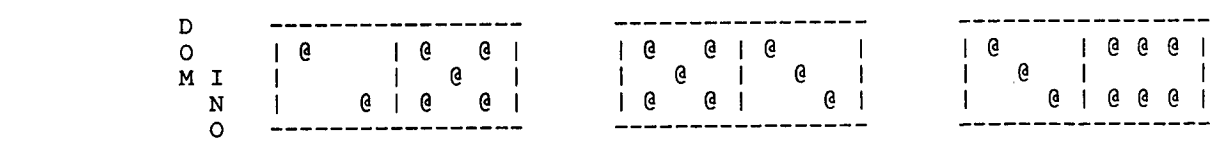
J'ai applique la regle init 2 :
SI    :vdf ques_crebf
ALORS :exec crebf

...
```

?l permet d'évaluer n'importe quelle expression LISP, y compris un appel aux fonctions internes de DOMINO :

```
? ?l (+ 5 7)
? (+ 5 7)
= 12

? ?l (:banniere)
```



?q permet de quitter DOMINO moins brutalement que ?a : l'utilisateur est invité à faire une sauvegarde avant de sortir :

```
Si vous voulez reprendre plus tard cette session, tapez un tiret (-)
suivi d'une ligne de commentaires. Sinon tapez un tiret tout seul :
? -lagrange
==> Je sauve cette session dans reprise.3

AU REVOIR !
Pour relancer DOMINO, tapez :    (:domino)
Pour sortir de LISP,  tapez :    (end)
```

?r imprime la base des règles chargées en mémoire. Comme pour la commande ?f, ces impressions seront améliorées :

```
BASE DE REGLES tsl :

REGLE 1 :
SI    :vdf ques_probleme
ALORS ques_comaco

REGLE 2 :
SI    :vdf ques_comaco
ALORS donner_elements

REGLE 3 :
SI    :vdf donner_elements
ALORS :exec comaco

...
```

?s permet de faire des sauvegardes intermédiaires, en cours de session. Normalement, DOMINO ne propose cela qu'en fin de session.

ANNEXE B : ORGANISATION DE LA BASE DE REGLES

Comme indiqué au § 4.2, la base de règles est contenue dans la variable LISP **:base_regles** (qui peut évoluer dynamiquement grâce à la fonction :charger_bc). Nous précisons ci-dessous la structuration de cette variable.

:base_regles contient une liste de règles (r_1 ... r_n).

Chaque règle r_i est elle-même une liste ayant la structure suivante :

(p a_1 ... a_m)

Le moteur examine les règles une à une.

Si la prémisse p d'une règle est vraie, alors les actions a_1 ... a_n sont exécutées et le moteur recommence un cycle.

Sinon le moteur s'arrête.

Les prémisses et les actions sont des expressions LISP quelconques. Elles peuvent donc appeler les fonctions qui manipulent la base de connaissances (§ C9).

Par exemple, la base de règles :

```
SI rien n'a encore été exécuté ALORS exécuter a
SI a vient d'être exécuté      ALORS epsilon vaut 10^-3
```

s'écrit :

```
'(((not (:val_fait vient_de_faire)) (a))
  ((:vdf a) (:aff_fait epsilon 1e-3)))
```

ANNEXE C : DESCRIPTION INTERNE DE DOMINO

La maquette DOMINO est écrite en Le_Lisp Version 15 [Chailloux 85]. Elle a été développée sur matériel APOLLO DN320 et DN600, mais fonctionne sur toute machine possédant Le_Lisp et Fortran77.

Les paragraphes qui suivent montrent certains aspects spécifiques du programme. Un **index** final (§ C13) renvoie aux principales fonctions décrites.

C1 Le point d'entrée principal

En préliminaire, voici le texte source de la fonction principale **:domino :**

```
(de :domino ()
    (:banniere) ;;; premiere chose a faire !
    (:bouffe_return) ;;; bouffe le return qui suit #s ou (:domino)
    (with ((inchan nil))

; ************************************
; * LES VARIABLES GLOBALES DE DOMINO *
; ************************************

    (let (
    (:base_regles)
    (:base_faits 'bf) ;;; on travaille implicitement sur la bf courante
    (:continuer)
    (:declarations)
;;; (:dir_domino) doit etre defini a l'exterieur
    (:escape #/?)
    (:historique)
    (:pref_br) ;;; prefixe les fichiers .ll (base de regles) et .forme
    (:pref_job)      ;;; prefixe les fichiers .dat .ftn _peb ...
    (:reprise)
    (:tirets (makestring 72 #/-))
    )

; ************************************

    (mapc 'remob (oblist :base_faits));;; assure que la base de faits est vide
    (mapc 'remob (oblist 'bf_anc))
    (:charger_br 'init)
```

```
;;; debut de la zone ou l'utilisateur peut taper des commandes

(catcherror t
   (tag boucle_principale

      (:lire_reprise)

      ;;; le moteur d'inferences :
      (while (:inference) ())

      (:tprint "Aucune regle n'est applicable. Arret.")))

(when :historique
      (catcherror t
         (tag boucle_principale
            (:ecrire_reprise))))

;;; fin de la zone ou l'utilisateur peut taper des commandes

(mapc 'remob (oblist :base_faits)) ;;; vide la base de faits
(mapc 'remob (oblist 'bf_anc))
(:tprint "AU REVOIR !")
( print "Pour relancer DOMINO, tapez :    (:domino)")
( print "Pour sortir de LISP,  tapez :    (end)")
(terpri)
t))))
```

C2 Les variables globales de DOMINO

Comme le montre le programme précédent, certaines variables sont déclarées dans la fonction principale *:domino* et sont ainsi accessibles par toutes les fonctions appelées en cours d'exécution :

:base_regles contient la base de règles, comme indiqué § 4.2.

:base_faits contient le nom du *package* contenant les faits ; ceux-ci sont gérés comme des variables Lisp classiques, ce qui permet des accès très rapides.

:continuer est vrai si le moteur doit continuer, faux sinon (§ 4.3.3).

:déclarations est utilisé par *:v_aff, :v_dcl, :v_dim* et *:generer_job* pour déclarer les tableaux Fortran à générer.

:escape est le caractère d'échappement qui précède les commandes de l'utilisateur (§ 3.3) ou les expressions Lisp contenues dans la forme des jobs (§ 4.3). Par défaut, il vaut "?".

:historique est une pile qui contient l'historique de la session en cours.
:pref_br est le préfixe des fichiers contenant la base de règles et la forme des jobs (§ 4.1).
:pref_job est le préfixe des fichiers formant un job : source, objet, données.
:reprise contient éventuellement l'historique d'une ancienne session. Dans ce cas, le système fonctionne en mode "reprise" et l'utilisateur ne doit pas retaper les mêmes valeurs (§ 3.2). Si *:reprise* vaut *nil,* le mode "reprise" est annulé.
:tirets contient 72 tirets en vue d'impressions.

C3 La boucle principale

Grâce au concept de *tag* en Le_Lisp, il est possible à tout moment de sortir de DOMINO par l'instruction *(exit boucle_principale)*. C'est ce qui est réalisé notamment par les commandes ?a et ?q (§ 3.3).

Le moteur d'inférence proprement dit, qui tourne jusqu'à ce qu'aucune règle ne soit applicable, est décrit ci-dessous.

C4 Le moteur d'inférence

La logique de ce moteur est présentée au § 4.2 (chaînage avant). Sa programmation en Lisp utilise une boucle *tant que* :

```
(while (:inference) ())
```

où **:inference** est une fonction qui parcourt les règles une à une. Si une règle est sélectionnée, la partie droite est évaluée et une valeur différente de *nil* est retournée. Cette fonction est appelée tant qu'une règle est sélectionnée, puis le moteur s'arrête.

Il est difficile de préciser l'**ordre** de ce moteur (*), puisque les variables ont ici une signification différente de celle des moteurs d'inférences classiques. Au sens **Prolog,** c'est un moteur d'ordre 0, mais des variables **Lisp** sont utilisées.

(*) Rappel : l'ordre (0, 0+ ou 1) situe la capacité d'un moteur à gérer des variables :
- un moteur d'ordre 0 ne peut manipuler que des faits constants,
- un moteur d'ordre 0+ gère des faits contenant des variables dont les valeurs sont définitivement fixées avant expertise,
- un moteur d'ordre 1 peut exploiter des règles dotées de variables modifiables dynamiquement durant l'expertise.

C5 Les reprises

Nous décrivons ici le mécanisme des reprises, qui économisent à la fois du temps-utilisateur et du temps-machine (§ 3.2) :

C5.1 Ce qui est sauvegardé

Au cours d'une session, différentes informations sont empilées (fonction *newl*) dans la variable **:historique :**
- les noms des bases de connaissances chargées,
- les règles appliquées,
- les faits modifiés,
- les valeurs lues par le SE.

C5.2 Utilitaires de sauvegarde et de reprise

La fonction **:ecrire_reprise** permet de sauvegarder la liste *:historique* sur un fichier. Plutôt que de demander à l'utilisateur le nom de ce fichier, nous avons préféré procéder de la manière suivante :
- le SE demande un commentaire décrivant la session en cours,
- il recherche un nouveau nom de fichier : *reprise.0, reprise.1, reprise.2,* etc,
- il écrit *:historique* dans ce nouveau fichier,
- il ajoute dans le fichier *reprise.txt* le numéro du fichier de reprise et le commentaire associé.

Au début de la session suivante, la fonction **:lire_reprise** est appelée : si le fichier *reprise.txt* est non vide, il est imprimé et on demande à l'utilisateur s'il veut faire une reprise. Si oui, le fichier correspondant est lu dans la variable **:reprise.**

Remarque : actuellement, le système permet seulement d'*ajouter* de nouvelles reprises. Pour en *supprimer* une, il faut détruire le fichier *reprise.i* et manipuler à l'éditeur de textes le fichier *reprise.txt.*

C5.3 Exploitation durant le fonctionnement du SE DOMINO

Lorsque la partie droite d'une règle doit être exécutée, on appelle la fonction **:execution**, qui se comporte différemment selon la valeur retournée par **:comment_executer**. Cette dernière fonction s'écrirait en "pseudo-Lisp" (la version réelle est optimisée) :

```
(de :comment_executer ()
    (when (la_regle_n_est_plus_la_même*)
          (setq :reprise nil)) ; annule le mode "reprise"
    (cond
       ((null :reprise) 1)
       ((les_faits_sont_les_mêmes**)
          (if (une_valeur_va_être_lue***) 2 3))
       (t 2)))
```

* Une comparaison du sommet de :reprise avec celui de :historique est effectuée.

** Pour comparer les bases de faits, on utilise **:bf_identique** qui compare les faits actuels (contenus dans le package **bf**) avec ceux obtenus à la session précédente (continuellement mis à jour dans le backage **bf_anc**, grâce à la fonction **:suivant_reprise**).

*** Pour savoir si on va lire une valeur, on appelle (**:je_vais_faire** ':lire_ligne).

Comportement de :execution selon la valeur retournée :

1 : exécution comme au premier appel de DOMINO, sans reprise.
2 : exécution comme en 1, mais à chaque lecture l'utilisateur voit apparaître entre crochets la valeur qu'il avait tapée auparavant.
3 : exécution simulée, en se contentant d'affecter les faits comme ils l'avaient été auparavant.

Pour synchroniser les reprises contenues dans *:reprise* avec la session en cours, il est parfois fait appel à **:vider_jusque** (notamment, *:execution* et *:lire_reprise* appellent *(:vider_jusque 'regle))* : cette dernière fonction est telle que *(:vider_jusque x)* parcourt *:reprise* en appelant **:suivant_reprise** jusqu'à ce qu'une action commençant par *x* soit trouvée.

C6 La génération d'un job

Rappelons que la fonction **:exec** permet de générer un job et de l'exécuter (§ 4.2, fin). La génération est effectuée par *:generer_job*. Si le fichier commence par "(" ou ";" on considère que c'est un fichier Lisp et on l'exécute avec *loadfile*. Sinon, on appelle le compilateur Fortran et le chargeur par l'intermédiaire de *comline* (qui est simulé par *#:startup:shell* sur Apollo).

:generer_job génère les fichiers de texte source (Fortran ou Lisp) et les données à partir des fichiers élémentaires D, E, F, I fournis par *:imp_forme* (sur Apollo, fichiers d.txt, e.txt, f.txt, i.txt).

:imp_forme édite sur les fichiers D, E, F, I l'évaluation des éléments 1, 2, 3, 4 (car, cadr, caddr, cadddr) de la liste retournée par *:lire_forme*.

(:lire_forme <job>) retourne :
- Si la ligne *-j <job>* n'est pas trouvée dans le fichier contenant la forme des jobs (§ 4.3), alors *nil*.
- Sinon, une liste (d e f i) :
  - d : liste des instructions de déclaration,
  - e : liste des instructions exécutables,
  - f : liste représentant les "fonctions utilisateur",
  - i : liste des données.

Chaque bloc d, e, f, i est lu au moyen de la fonction **:lire_bloc**, qui regroupe les lignes de texte contenues dans le fichier *forme*. Ces lignes sont lues par *:lire_instruction*.

Le fonctionnement de **:lire_instruction** est illustré par les exemples suivants, où le caractère d'échappement standard "?" est employé :

```
      I = 0          ---->   "      I = 0"
?(+ 1 2 3)           ---->   (+ 1 2 3)
?x                   ---->   (:val_fait x)
      J = ?y         ---->   (catenate "      J = " (:val_fait y))
      K = ?z + 3     ---->   (catenate "      K = "
                               (catenate (:val_fait z) " + 3"))
```

Ainsi, *:imp_forme* n'a plus qu'à appeler *eval* et à éditer le résultat obtenu au moyen de *:imp_can*.

:imp_can dirige le canal de sortie vers un des fichiers D, E, F, I ouverts par *:generer_job,* puis appelle *:imp_ins* pour imprimer une instruction. Cette manière de programmer, en dirigeant le canal de sortie au dernier moment, permet d'afficher sur l'écran toutes sortes d'informations (questions à l'utilisateur, traces, etc) sans risque de voir ces impressions déviées sur un fichier !

La définition de **:imp_ins** s'écrit récursivement :

```
(de :imp_ins (x)
    (cond ((null x) (print "C --- NIL"))
          ((atom x) (print x))
          (t (mapc ':imp_ins x))))
```

Ainsi, si x est une liste, on obtient plusieurs lignes, comme indiqué § 4.3.2.

C7 Fonctions de lecture

La fonction de base est **:donnee_suivante**, qui lit une ligne à la manière de *readline* (liste de codes ascii), mais :
- supprime les blancs à gauche et à droite,
- si la ligne commence par un caractère d'échappement ("?" en général), effectue la commande associée et effectue une relecture.

La fonction *:donnee_suivante* est appelée par **:lire_ligne,** qui imprime l'ancienne valeur fournie s'il y a reprise, et empile le résultat de la lecture dans *:historique* (§C5).

:lire_s transforme le résultat de :lire_ligne en un objet Lisp au moyen de la fonction *implode.* Cette fonction :lire_s est appelée par tous les utilitaires de lecture :

:lire_chaine	retourne (string (:lire_s)),
:lire_commandes	demande à l'utilisateur de taper éventuellement des commandes, puis d'appuyer sur la touche Retour,
:lire_entier	lit un entier,
:lire_flottant	lit un flottant,
:lire_logique	retourne *t* pour *(oui o yes y), nil* pour *(non n no),*
:lire_logique_oui	est identique, mais *oui* est pris par défaut,
:lire_logique_non	est identique, mais *non* est pris par défaut.

Si l'utilisateur donne une valeur qui ne correspond pas au type attendu, il est interrogé une nouvelle fois. On peut aussi imposer des contraintes sur les valeurs lues par composition de fonctions, ce qui permet de vérifier certaines données. Par exemple :
(:gez ':lire_flottant) impose que le flottant lu soit ≥ 0,
(:gz ':lire_entier) impose que l'entier lu soit > 0.

Enfin, la fonction **:menu** permet d'afficher très simplement des menus pour l'utilisateur. Par exemple, on désire afficher :

```
0 finir
1 continuer
2 attendre
```

et obtenir l'atome *f* si l'utilisateur tape 0, *c* pour 1, et *a* pour 2. Ceci est réalisable par l'appel suivant :

```
(:menu '((f "finir") (c "continuer") (a "attendre")))
```

Si le texte d'une option comprend plusieurs lignes, il suffit de fournir autant de chaînes de caractères.

C8 Fonctions d'impression

:banniere imprime la bannière de DOMINO (début de session).

:explications imprime des explications générales sur DOMINO (commande ??).

:imprimer_bf imprime la base de faits (?f).

:imprimer_br imprime la base de règles (?r).

:imprimer_historique imprime l'historique de la session en cours (?h).

:imprimer_regle imprime une règle en clair.

Enfin, deux fonctions élémentaires allègent la programmation dans le cas fréquent où un saut de ligne est désiré avant l'impression de valeurs : *:tprinflush* et *tprint.*

:tprinflush saute une ligne, puis imprime la valeur de ses arguments sans passer à la ligne suivante.

Programmation en LISP (terpri suivi de prinflush) :

```
(de :tprinflush l (terpri) (apply 'prinflush l))
```

Exemple :

```
? (progn (:tprinflush "Quelle année ") (read))

Quelle année ? 1954
= 1954
```

:tprint est analogue à *:tprinflush,* mais passe ensuite à la ligne suivante.

Programmation en LISP (terpri suivi de print) :

```
(de :tprint l (terpri) (apply 'print l))
```

Exemple :

```
? (:tprint "6 fois 4 font " (* 6 4))

6 fois 4 font 24
= 24
```

C9 Manipulation de la base de connaissance

(:aff_fait f v) affecte la valeur *v* au fait *f.*

Exemple :

```
(:aff_fait taille_probleme
           (if (< x y) 'petit 'grand))
```

(:charger_bc x) charge la nouvelle base de connaissance *x* (§ 4.2).

(:expansion br) réalise l'expansion de la base de règles *br* : toute règle comportant plusieurs actions en partie droite est décomposée en autant de règles. Ceci permet de simplifier le moteur d'inférence et de mieux gérer les reprises et les explications du raisonnement.

Ex :

```
SI a ALORS b, c
```

devient

```
SI a ALORS b
SI b vient d'être exécuté ALORS c
```

(:val_fait f) retourne la valeur du fait *f*, ou *nil* si le fait n'a pas été affecté.
Exemple : `(:val_fait taille_probleme)`

(:vdf a) est un prédicat vrai si et seulement si la dernière action exécutée est *a*. Cette information étant contenue dans le fait de nom *vient_de_faire*, *:vdf* s'écrit :

```
(df :vdf nev_lispar                    ;;; en FEXPR
    (:vdf_expr nev_lispar))
(de :vdf_expr (lispar)                 ;;; en EXPR
    (equal (:val_fait vient_de_faire) lispar))
```

C10 Utilitaires pour les ensembles

Ces utilitaires effectuent un certain nombre d'opérations sur les ensembles, qui sont représentés par des listes. Ils se sont montrés très utiles pour notre application à Modulef, mais restent très généraux.

(:e_ensemble l) élimine tous les éléments en double dans la liste l.
Ex : `(:e_ensemble '(a b c b a d a)) -> (a b c d)`

(:e_inter l) : l est une liste d'ensembles, et on obtient l'intersection de tous ces ensembles.
Ex : `(:e_inter '((1 3) (1 2 3) (1 3 4 5))) -> (1 3)`

(:e_inter2 x y) retourne l'intersection des ensembles x et y.
Ex : `(:e_inter2 '(bleu blanc rouge) '(vert orange rouge))`
`-> (rouge)`

(:e_select f l) retourne tous les éléments de l tels que f est vrai.
Ex : `(:e_select 'plusp '(-1 7 0 -2)) -> (7 0)`

(:e_telque f l) retourne le premier élément de l tel que f est vrai.
Ex : `(:e_telque 'plusp '(-1 7 0 -2)) -> 7`

C11 Utilitaires pour les vecteurs

Ces utilitaires ont été conçus pour passer facilement des objets LISP (listes ou vecteurs) aux tableaux Fortran.

(:v_aff type tableau objet) traduit un objet Lisp en une suite d'affectations Fortran. La déclaration du tableau Fortran peut ensuite être obtenue par :v_dcl.

Ex :
```
? (setq :declarations nil)
= ()
? (:v_aff "INTEGER" "M" '(36 54 21))
= ("      M(1) = 36" "      M(2) = 54" "      M(3) = 21")
? (:v_aff "REAL" "T" '(#[1. 2. 3.] #[4. 5. 6.]))
= ("      T(1,1) = 1."   "      T(1,2) = 2."
   "      T(1,3) = 3."   "      T(2,1) = 4."
   "      T(2,2) = 5."   "      T(2,3) = 6.")
? :declarations
= (("REAL" "T" 2 3) ("INTEGER" "M" 3))
? (:v_dcl)
= ("      INTEGER M(3)"   "      REAL T(2,3)")
? (:v_dim "M")
= #[3]
? (:v_dim "T")
= #[2 3]
```

(:v_data objet) a un but analogue à :v_aff, mais est utilisé cette fois pour générer un jeu de données, formé de :
- nombre de dimensions de l'objet,
- dimensions de l'objet,
- valeurs des éléments de l'objet, "ligne par ligne".

Ex : `(:v_data #[#[1. 2.] #[3. 4.] #[5. 6.]])`
`-> (2 3 2 1. 2. 3. 4. 5. 6.)`

(:v_dcl) retourne les déclarations des tableaux créés par *:v_aff* (voir ci-dessus).

(:v_dim tableau) retourne le vecteur des dimensions d'un tableau créé par *:v_aff* (voir ci-dessus).

(:v_filtre filtre vecteur) retourne *vecteur* sans les éléments *i* tels que *filtre(i) = nil.* *(:v_filtre v v)* équivaut à *(:v_nnil v).*

Ex : `(:v_filtre #[() t () t t] #[0 1 2 () 4]) -> #[1 () 4]`

(:v_ind_nnil vecteur) retourne le vecteur formé des indices différents de *nil*. Attention : les indices commencent à 1 comme en Fortran !

Ex : `(:v_ind_nnil #[a () b c ()]) -> #[1 3 4]`

(:v_iter initial inc final) retourne un vecteur formé par une itération de *initial* à *final*, avec l'incrément *inc*.

Ex : `(:v_iter 10 5 25) -> #[10 15 20 25]`

(:v_lmax l) donne la plus grande longueur des éléments de la liste l.

Ex : `(:v_lmax '((a b) c (d e f g) (h))) -> 4`

(:v_nb_nnil vecteur) retourne le nombre d'éléments différents de nil.

Ex : `(:v_nb_nnil #[a () b c ()]) -> 3`

(:v_nil0 vecteur) remplace les éléments valant nil par 0.

Ex : `(:v_nil0 #[a () b c ()]) -> #[a 0 b c 0]`

(:v_nnil vecteur) retourne un vecteur formé des éléments différents de nil.

Ex : `(:v_nnil #[a () b c ()]) -> #[a b c]`

(:v_op op v w) effectue l'opération binaire *op* entre chacun des éléments des vecteurs *v* et *w*.

Ex : `(:v_op '+ #[15 40 39] #[3 10 5]) -> #[18 50 44]`

(:v_vecteur objet) convertit un objet en vecteur.

Ex :
```
(:v_vecteur #[a b])  -> #[a b]
(:v_vecteur '(x y z) -> #[x y z]
(:v_vecteur 'atome)  -> imprime un message d'erreur
                        et retourne #[]
```

C12 Utilitaires divers

Ce paragraphe recense quelques utilitaires qui n'ont pas pu être classés dans les rubriques précédentes.

(:cat p_1 ... p_n) permet de gérer des fichiers à la manière de la commande Unix *cat (catenate and print).* Les paramètres p_1 ... p_n sont en général des noms de fichiers qui sont lus dans l'ordre où ils apparaissent et écrits sur un canal de sortie. Si un nom p_i commence par ">", par exemple ">xxx", alors il s'agit en fait d'un fichier de sortie et les écritures se font sur le fichier "xxx". Sinon, les écritures se font sur le canal de sortie standard (l'écran en général). Enfin, un paramètre commençant par "-" est recopié tel quel.

Ex : `(:cat "-debut" "f1" "f2" "-fin" ">f")`

donne dans le fichier f :

```
debut
... fichier f1
... fichier f2
fin
```

(:continue) empêche l'arrêt du moteur d'inférences en positionnant la variable :continuer (§ C2).

(:domino_core) sauve l'image mémoire de la version compilée de DOMINO. A quelques détails près, la programmation en LISP est :

```
(de :domino_core ()
    (compile-all-in-core)
    (save-core "domino.core")
    (:domino))
```

Ainsi, lorsque l'image mémoire est restaurée, l'utilisateur se trouve directement dans la boucle principale de DOMINO.

(:dump exp) imprime l'expression *exp,* puis son évaluation. Il est parfois commode d'insérer des appels à :dump pour vérifier le comportement d'un programme.

Ex : `(:dump (+ 5 4))`

imprime

```
? (+ 5 4)
= 9
```

et retourne la valeur 9.

(:guill x) indique si des guillemets doivent être imprimés ou non autour des chaînes de caractères. De façon plus formelle, :guill réalise l'affectation suivante :

```
(de :guill (x) (setq #:system:print-for-read x))
```

(:majuscule x) traduit une chaîne de caractères en majuscules.
Ex : `(:majuscule "Exemple bateau") -> "EXEMPLE BATEAU"`

(:question q e) pose la question q, et n'évalue e que si l'utilisateur répond oui à la question. Cette fonction n'est utilisable que dans le contexte de DOMINO, car elle utilise des variables internes (*:reprise, :historique, :escape,* etc, § C2).

(:supp_diese x) ne retourne que la dernière partie des noms de variables contenues dans un package, afin d'améliorer les impressions (notamment dans :imprimer_bf et :imprimer_br).
Ex : `(:supp_diese '#:user:vdf) -> ":vdf"`

C13 Index

de : fonction de type EXPR

:aff_fait	df	C9
:banniere	de	C8
:bf_identique	de	C5.3
:cat	de	C12
:charger_bc	de	C9
:comment_executer	de	C5.3
:continue	de	C12
:domino	de	C1
:domino_core	de	C12
:donnee_suivante	de	C7
:dump	df	C12
:e_ensemble	de	C10
:e_inter	de	C10
:e_inter2	de	C10
:e_select	de	C10
:e_telque	de	C10
:ecrire_reprise	de	C5.2
:exec	df	C6
:execution	de	C5.3
:expansion	de	C9
:explications	de	C8
:generer_job	de	C6
:gez	de	C7
:guill	de	C12
:gz	de	C7
:imp_can	de	C6
:imp_forme	de	C6
:imp_ins	de	C6
:imprimer_bf	de	C8
:imprimer_br	de	C8
:imprimer_historique	de	C8
:imprimer_regle	de	C8
:inference	de	C4
:je_vais_faire	de	C5.3
:lire_bloc	de	C6

df : fonction de type FEXPR

:lire_chaine	de	C7
:lire_commandes	de	C7
:lire_entier	de	C7
:lire_flottant	de	C7
:lire_forme	de	C6
:lire_instruction	de	C6
:lire_ligne	de	C7
:lire_logique	de	C7
:lire_logique_non	de	C7
:lire_logique_oui	de	C7
:lire_reprise	de	C5.2
:lire_s	de	C7
:majuscule	de	C12
:menu	de	C7
:question	df	C12
:suivant_reprise	de	C5.3
:supp_diese	de	C12
:tprinflush	de	C8
:tprint	de	C8
:v_aff	df	C11
:v_data	de	C11
:v_dcl	de	C11
:v_dim	de	C11
:v_filtre	de	C11
:v_ind_nnil	de	C11
:v_iter	de	C11
:v_lmax	de	C11
:v_nb_nnil	de	C11
:v_nil0	de	C11
:v_nnil	de	C11
:v_op	de	C11
:v_vecteur	de	C11
:val_fait	df	C9
:vdf	df	C9
:vider_jusque	de	C5.3

ANNEXE D : INSTALLATION SUR APOLLO

DOMINO est écrit en Le_Lisp [Chailloux 85] et fonctionne actuellement sur matériel APOLLO DN320 et DN600, système DOMAIN/IX (qui englobe AEGIS). Le portage de Le_Lisp sur ce matériel étant encore en phase expérimentale, nous avons provisoirement développé notre propre environnement. Cependant, le langage choisi assure une bonne **portabilité** sur de nombreuses machines.

Au processus habituel d'**installation**, nous avons ajouté l'étape (2) ci-dessous :

```
$ wd /udd/lisp/apollo
$ makefile                                    (1)
  ...
$ lispgo
******* Le_Lisp (by INRIA) version 15 (31/Decembre/84) [apollo]
  ...
? (load-std () t t t t)
Je charge /udd/lisp/llib/virtty.ll
  ...
  (llcp-std '<nom>)  pour compiler l'environnement standard
= ()
? (load startup.ll)                           (2)
= startup.ll
? (llcp-std 'lelispa)
Attendez, je sauve :
Systeme standard compile avec editeur avec environnement
  avec compilateur
******* Le_Lisp (by INRIA) version 15 (31/Decembre/84) [apollo]
= Systeme standard compile avec editeur avec environnement
  avec compilateur
? (end)
```

Ensuite, le lancement de LISP s'effectue par la commande :

```
/udd/lisp/apollo/lispgo 6 -r /udd/lisp/llcore/lelispa.core
```

(sans mettre vt100 en début de ligne, afin de profiter de toutes les fonctionnalités du poste de travail Apollo).

En **(1)**, des programmes écrits en assembleur ou en C sont compilés (ne pas tenir compte des messages d'avertissement). Il est important de bien ajuster les différentes zones mémoire [Devin 85] *(pile, code, tas, entiers, flottants, vecteurs, chaînes, symboles, cellules de liste)* : avec des zones trop petites, le gc (*garbage collector,* ou ramasse-miettes) est appelé très souvent, et une erreur fatale peut se produire si aucun espace mémoire n'est récupérable ; avec des zones trop grandes, le gc devient très lent (surtout en mémoire virtuelle). Les paramètres finalement retenus sont :

```
SSTACK=6    SCODE=96    SHEAP=200    SNUMB=1    SFLOAT=1
SVECT=1     SSTRG=15    SSYMB=5      SCONS=5
```

N.B. La taille des cellules de liste (SCONS) est ajustable au moment du lancement de LISP (6 * 8 Kcellules dans la commande ci-dessus).

En **(2)**, un fichier définissant le macro-caractère *s* est chargé :

```
(defsharp s ()
   '((loadfile "/udd/modulef/laug/user_data/startup.ll" nil)))
```

Ainsi, lorsque Le_Lisp entre dans sa boucle principale, l'utilisateur peut taper #s afin de charger le fichier indiqué. Ce moyen est plus souple que l'option *-r fichier* de la commande Lisp, puisqu'une modification du fichier *.../user_data/startup.ll* n'oblige pas à recréer une image mémoire.

Ce dernier fichier contient des définitions de fonctions et de #-macros :

(probefile f) est défini comme dans le manuel [Chailloux 85] (N.B. En mode compilé, le message "No such file or directory" apparaît si le fichier est absent).

(#:startup:rogner posit cars) élimine les blancs de *cars* selon la valeur de *posit* (*g* gauche, *d* droite, *gd* gauche et droite). Le paramètre cars est une liste de codes ascii ou une chaîne de caractères. Le résultat obtenu a le même type.

Ex : `(#:startup:rogner 'gd "  a bc   ") -> "a bc"`

(#:startup:shell lignes) est un palliatif qui remplace la fonction *comline* : un fichier de nom *temporaire.sh* est généré en recopiant le paramètre *lignes,* puis ce fichier est exécuté en appuyant sur la touche F1 spécialement reprogrammée. (*)

#c *(comline)* remplace !

#e *(edit)* remplace ^E

#f *(find)* remplace ^F

#l *(load)* remplace ^L

#p *(pretty)* remplace ^P

#r recharge le fichier contenant une fonction donnée.

(*) Au moment du lancement de LISP, deux *process* sont créés : *le_lisp* qui n'exécute que des instructions LISP, et *comline* qui exécute des commandes, en particulier l'appel d'un programme FORTRAN. Pour se mettre sous *comline,* la touche F1 est reprogrammée :

```
xdmc "kd f1 wp comline -t;es 'temporaire.sh';en ke"
```

Pour revenir sous LISP, le fichier de commandes *temporaire.sh* se termine par :

```
xdmc "es '/udd/.../revenir_sous_lisp.sh';en"
```

et /udd/.../revenir_sous_lisp.sh contient :

```
dlf temporaire.sh
xdmc "wp le_lisp -t;es 'continue';en"
```

Le programme LISP redémarre dès qu'il a lu la ligne "continue".

N.B. Cette manière de procéder avec deux *process* ne permet pas de changer de catalogue *(change_directory).*

D2 Installation de DOMINO

L'organisation générale de DOMINO, avec des fichiers suffixés par ll *et* forme, *est décrite en première partie de ce rapport (§ 4). Ce paragraphe ne décrit que les détails d'implémentation spécifiques à Apollo.*

La commande **/udd/modulef/com/domino** affecte un nom de catalogue dans la variable **^domino**, crée les *process* "comline" et "le_lisp", et reprogramme certaines touches du clavier (§ D1.2).

Le catalogue repéré par ^domino contient tous les fichiers nécessaires à l'exécution de DOMINO, en particulier :

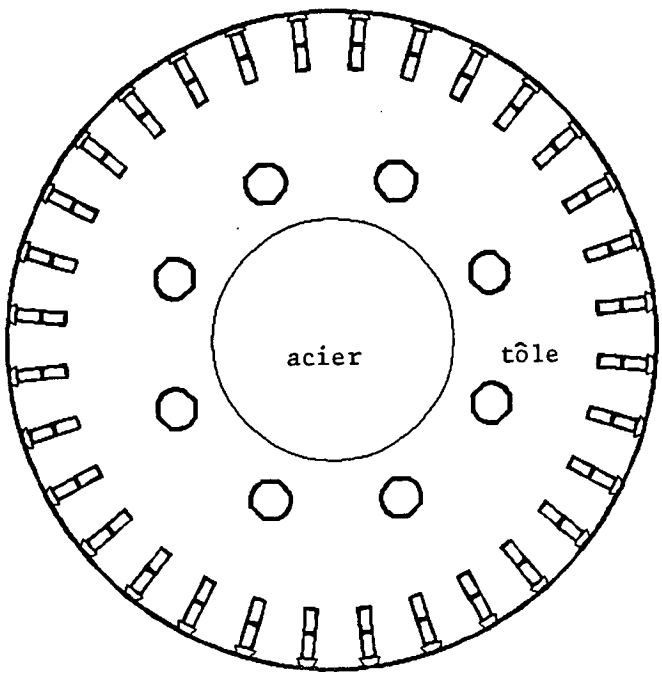
domino.core	image mémoire de DOMINO,
?*.forme	fichiers décrivant la forme informatique des jobs,
?*.ll	fichiers décrivant les règles de stratégie,
start_comline.sh	commandes exécutées au début du *process* "comline" (inlib Modulef),
start_le_lisp.sh	commandes exécutées au début du *process* "le_lisp" (appel de l'interprète et pause avant la destruction du *process*).

ANNEXE E : EXEMPLE D'UTILISATION

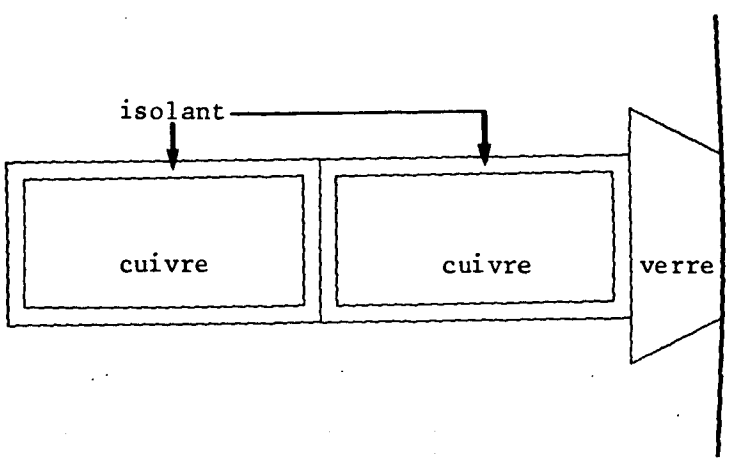
Le problème ci-dessous correspond à un cas réel plus complexe que celui décrit § 3.1.

E1 Le problème

Le rotor d'un moteur électrique est formé principalement d'un axe en acier entouré de tôle :



Le circuit électrique du moteur est formé de 32 éléments, chacun étant constitué de deux fils de section rectangulaire (cuivre entouré d'isolant) maintenus par un bouchon en verre :



Par effet Joule, les conducteurs dégagent de la chaleur, qui se dissipe dans l'air à la périphérie du rotor et par les 8 trous de refroidissement percés dans la tôle. On désire obtenir les courbes isothermes.

Valeurs numériques :

- Conductivités thermiques (W/m°C) :

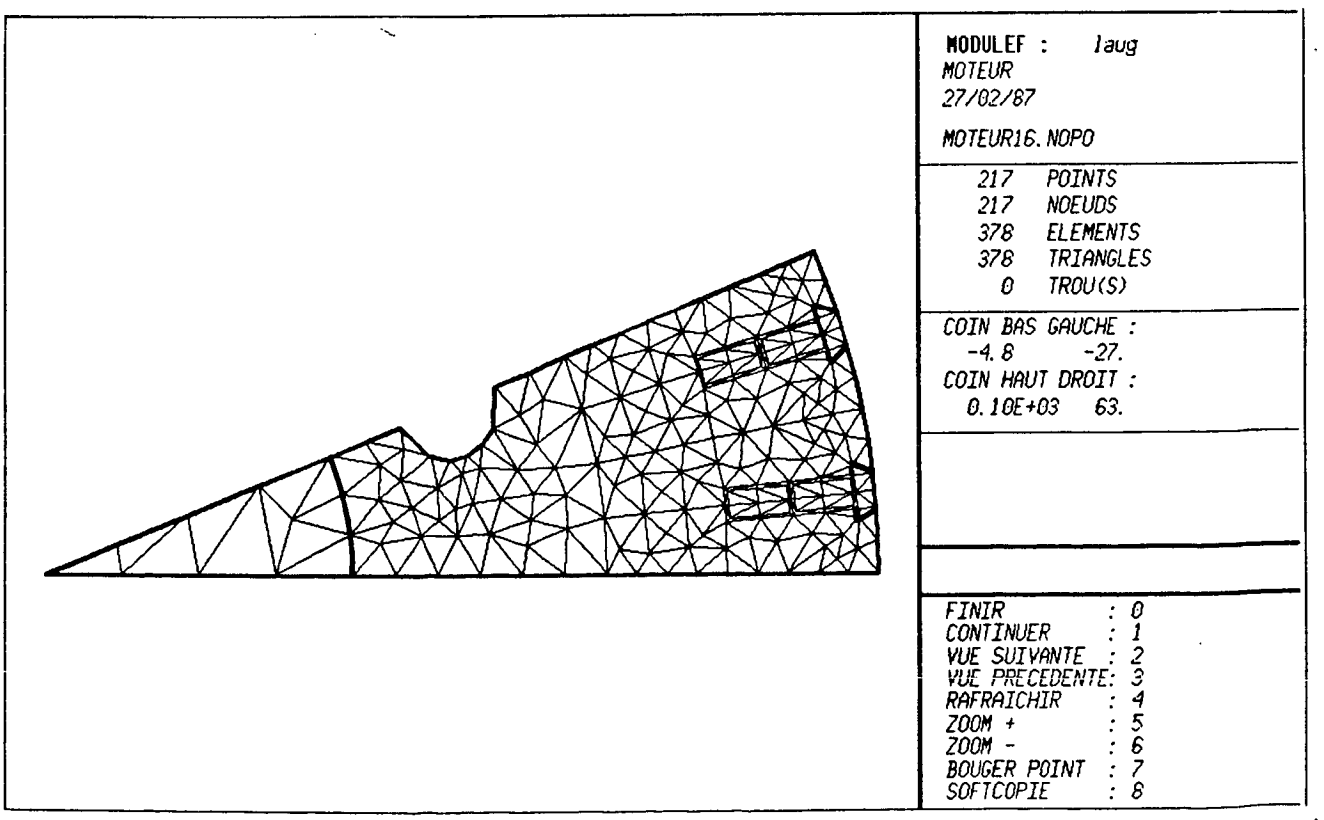
isolant	0.3	verre	0.8	acier	46
tôle	55	cuivre	385		

- Chaleur dégagée par les conducteurs en cuivre : 45 W/m^2

- Air ambiant : température 20 °C, coefficient d'échange 50 $W/m^2°C$

E2 Résolution avec DOMINO :

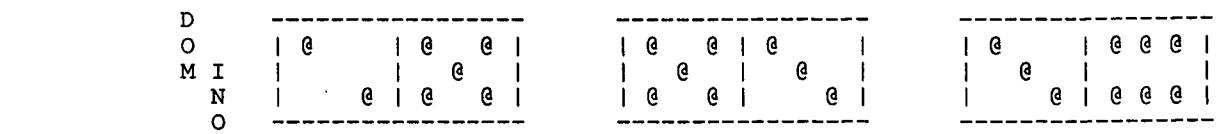
Compte tenu des symétries, il suffit de mailler $1/16^e$ du moteur. Ce maillage est stocké dans la Structure de Données *moteur16.nopo,* qui est visualisée ci-dessous :



N.B. Le maillage est ici de degré 1 (points = nœuds). Le système expert est capable d'ajouter des nœuds si nécessaire.

L'appel de DOMINO donne lieu au dialogue suivant (les réponses sont précédées d'un point d'interrogation et écrites en **gras souligné**) :

? (:domino)




```
Version 3

Tapez ? si vous voulez obtenir la liste des commandes disponibles


Vous devez avoir cree un fichier de maillage (nopo) avec noeuds et
points confondus, a l'aide du mailleur APNOXX ou APN3XX, EMC2...
Si ce n'est pas le cas, tapez ?a pour abandonner.

Nom du fichier de maillage ? moteur16.nopo

Je genere et execute le job crebf

TITRE : MOTEUR

Type du probleme ?
  0 thermique  stationnaire lineaire
  1 thermique  transitoire  lineaire
  2 thermique  transitoire  non lineaire
  3 elasticite stationnaire lineaire
  4 modes propres
  5 mecanique des fluides
?
==> thermique  stationnaire lineaire

Est-ce un probleme axisymetrique ? n

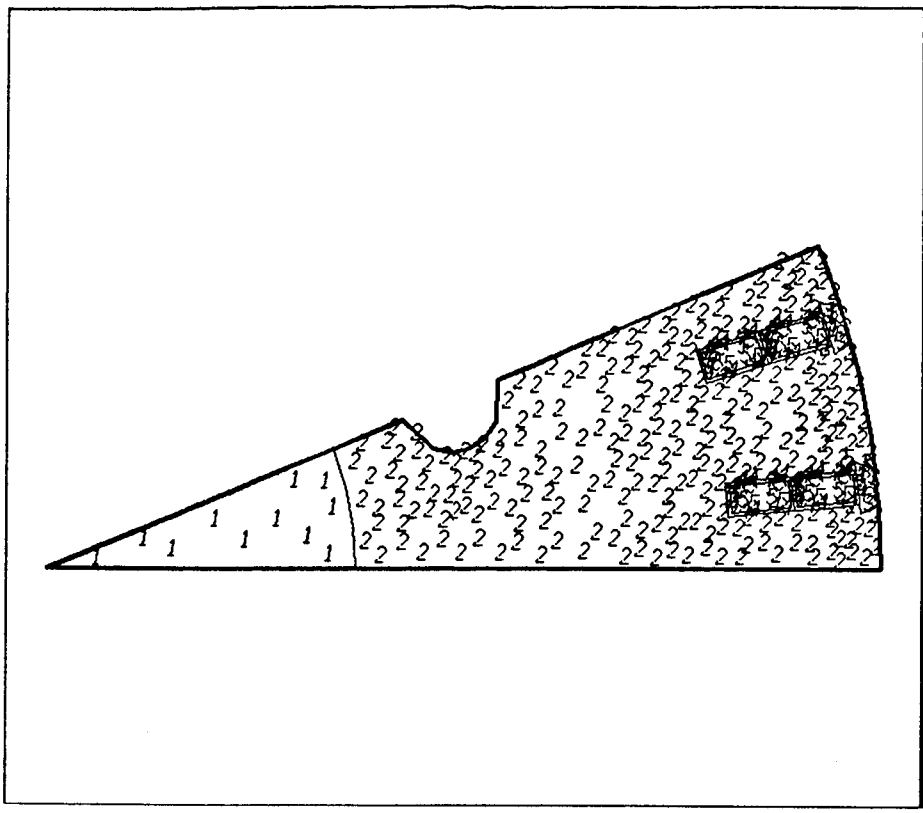
Quels resultats voulez-vous privilegier :
  0 temperatures    (elements de Lagrange)
  1 flux de chaleur (elements mixtes)
?
==> temperatures    (elements de Lagrange)

Choisissez le degre d'interpolation parmi (1 2)
Plus le degre est grand, plus le temps-calcul est grand, mais meilleure
est la precision.   Le degre le plus eleve est recommande : 2
? 2

Je genere et execute le job comaco

Voulez-vous voir votre maillage apres interpolation ? o

Je genere et execute le job trnoxx
```



MODULEF : *laug*
MOTEUR
26/02/87

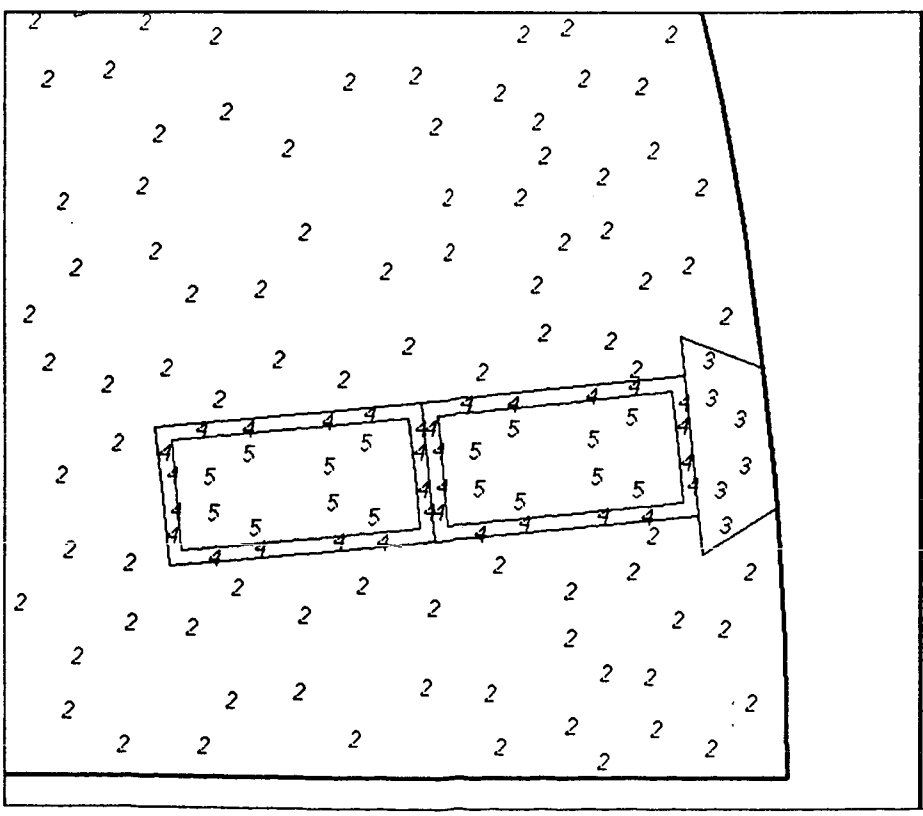
sd. nopo. 1

217	*POINTS*
611	*NOEUDS*
378	*ELEMENTS*
378	*TRIANGLES*
0	*TROU(S)*

COIN BAS GAUCHE :
-4.8 -27.
COIN HAUT DROIT :
0.10E+03 63.

NUMERO SOUS DOMAINE

FINIR	*: 0*
CONTINUER	*: 1*
VUE SUIVANTE	*: 2*
VUE PRECEDENTE	*: 3*
RAFRAICHIR	*: 4*
ZOOM +	*: 5*
ZOOM -	*: 6*
BOUGER POINT	*: 7*
SOFTCOPIE	*: 8*



MODULEF : *laug*
MOTEUR
26/02/87

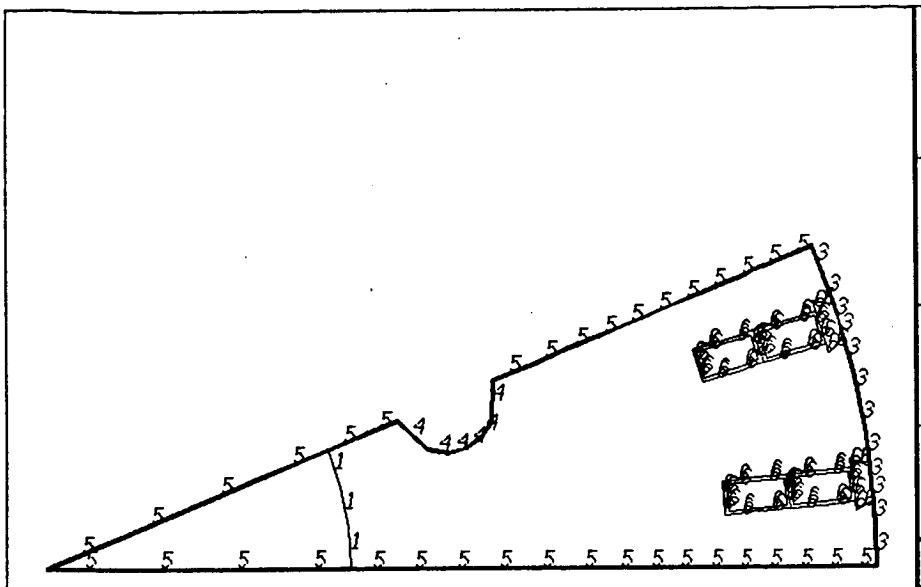
sd. nopo. 1

217	*POINTS*
611	*NOEUDS*
378	*ELEMENTS*
378	*TRIANGLES*
0	*TROU(S)*

COIN BAS GAUCHE :
74. -0.89
COIN HAUT DROIT :
99. 21.

NUMERO SOUS DOMAINE

FINIR	*: 0*
CONTINUER	*: 1*
VUE SUIVANTE	*: 2*
VUE PRECEDENTE	*: 3*
RAFRAICHIR	*: 4*
ZOOM +	*: 5*
ZOOM -	*: 6*
BOUGER POINT	*: 7*
SOFTCOPIE	*: 8*



MODULEF : *laug*
MOTEUR
26/02/87

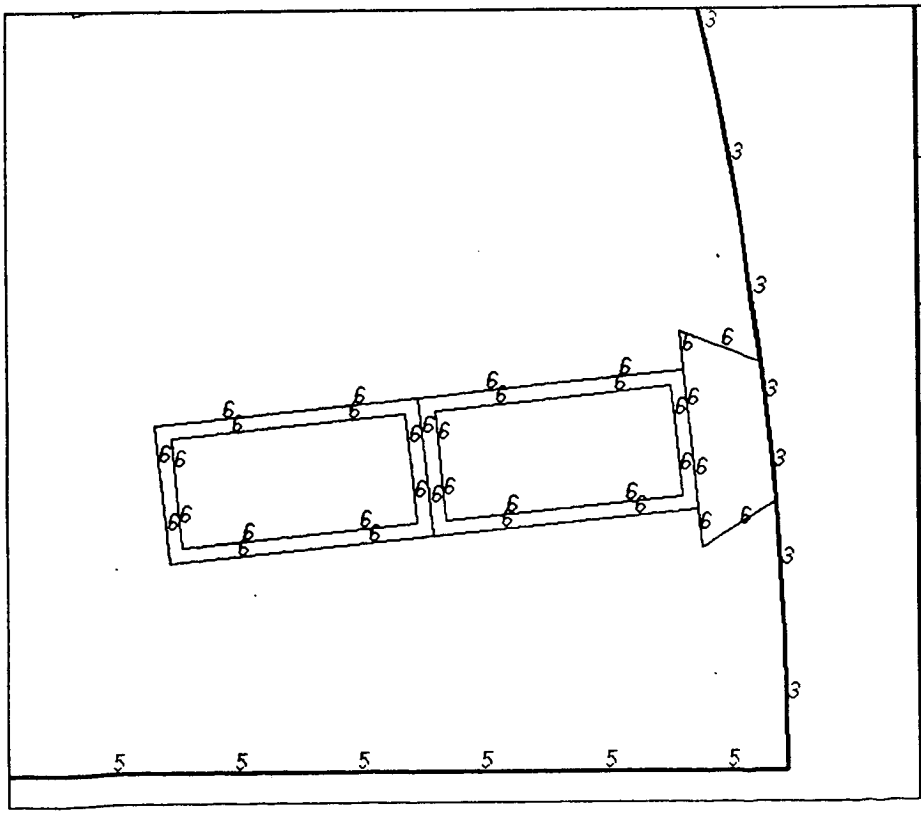
sd. nopo. 1

217	POINTS
811	NOEUDS
378	ELEMENTS
378	TRIANGLES
0	TROU(S)

COIN BAS GAUCHE :
-4.8 -27.
COIN HAUT DROIT :
0.10E+03 63.

REFERENCE (ARETE)

FINIR	: 0
CONTINUER	: 1
VUE SUIVANTE	: 2
VUE PRECEDENTE	: 3
RAFRAICHIR	: 4
ZOOM +	: 5
ZOOM -	: 6
BOUGER POINT	: 7
SOFTCOPIE	: 8



MODULEF : *laug*
MOTEUR
26/02/87

sd. nopo. 1

217	POINTS
811	NOEUDS
378	ELEMENTS
378	TRIANGLES
0	TROU(S)

COIN BAS GAUCHE :
74. -0.89
COIN HAUT DROIT :
99. 21.

REFERENCE (ARETE)

FINIR	: 0
CONTINUER	: 1
VUE SUIVANTE	: 2
VUE PRECEDENTE	: 3
RAFRAICHIR	: 4
ZOOM +	: 5
ZOOM -	: 6
BOUGER POINT	: 7
SOFTCOPIE	: 8

Y a-t-il des conditions aux limites en relation lineaire ? n

VEUILLEZ DECRIRE LES 5 SOUS-DOMAINES DU MAILLAGE :

SOUS-DOMAINE NUMERO 1 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples : 1 (3 -2 5)
? 46

Valeur de la source de chaleur (W / m2) ? 0

SOUS-DOMAINE NUMERO 2 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples : 1 (3 -2 5)
? 55

Valeur de la source de chaleur (W / m2) ? 0

SOUS-DOMAINE NUMERO 3 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples : 1 (3 -2 5)
? 0.8

Valeur de la source de chaleur (W / m2) ? 0

SOUS-DOMAINE NUMERO 4 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples : 1 (3 -2 5)
? 0.3

Valeur de la source de chaleur (W / m2) ? 0

```
SOUS-DOMAINE NUMERO 5 :

Conductivite (W / m degre) :
- si le materiau est isotrope, donnez une valeur,
- sinon, donnez la liste des conductivites (k11 k12 k22).
Exemples :      1      (3 -2 5)
? 385

Valeur de la source de chaleur (W / m2) ? 45

-------------------------------------------------------------------------

VEUILLEZ DECRIRE LES CONDITIONS AUX LIMITES SUR LES 6 REFERENCES :

-------------------------------------------------------------------------

REFERENCE NUMERO 1 ;

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 3
==> Aucune condition thermique (reference purement geometrique)

-------------------------------------------------------------------------

REFERENCE NUMERO 2 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 3
==> Aucune condition thermique (reference purement geometrique)

-------------------------------------------------------------------------

REFERENCE NUMERO 3 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 1
==> Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
Valeur du coefficient de transfert (W / m2 degre) ? 50

Valeur de la temperature du fluide exterieur (degre) ? 20
```

```
---------------------------------------------------------------------

REFERENCE NUMERO 4 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 1
==> Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
Valeur du coefficient de transfert (W / m2 degre) ? 50

Valeur de la temperature du fluide exterieur (degre) ? 20

---------------------------------------------------------------------

REFERENCE NUMERO 5 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 2
==> Neumann   : flux normal de chaleur (f_gamma)

Valeur du flux normal de chaleur (W / m2) ? 0

---------------------------------------------------------------------

REFERENCE NUMERO 6 :

Type de condition ?
  0 Dirichlet : temperature imposee (u_barre)
  1 Fourier   : coefficient de transfert (g)
              + temperature du fluide exterieur (u0 = f_gamma/g)
  2 Neumann   : flux normal de chaleur (f_gamma)
  3 Aucune condition thermique (reference purement geometrique)
? 3
==> Aucune condition thermique (reference purement geometrique)

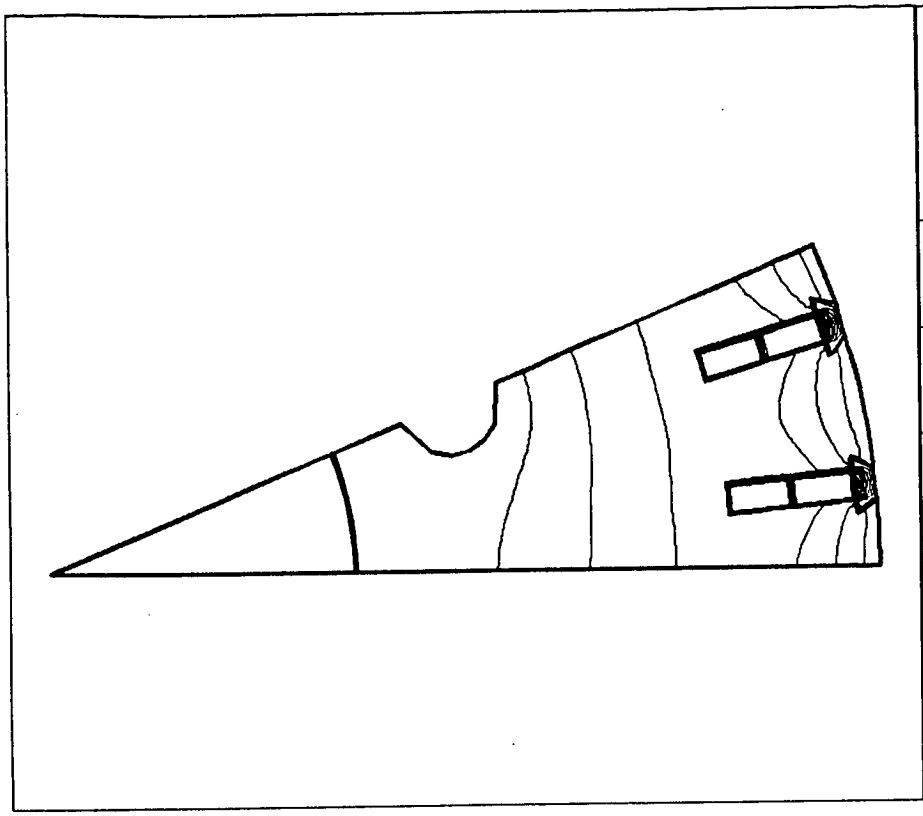
Tapez eventuellement des commandes, puis RETURN  ?

Je genere et execute le job cobdcl

Je genere et execute le job valpro

Je genere et execute le job cholxx

Je genere et execute le job trmcxx
```



MODULEF : laug
MOTEUR
27/02/87
sd.mail.0
sd.coor.0
sd.b.0

289 POINTS
811 NOEUDS
378 ELEMENTS
378 TRIANGLES

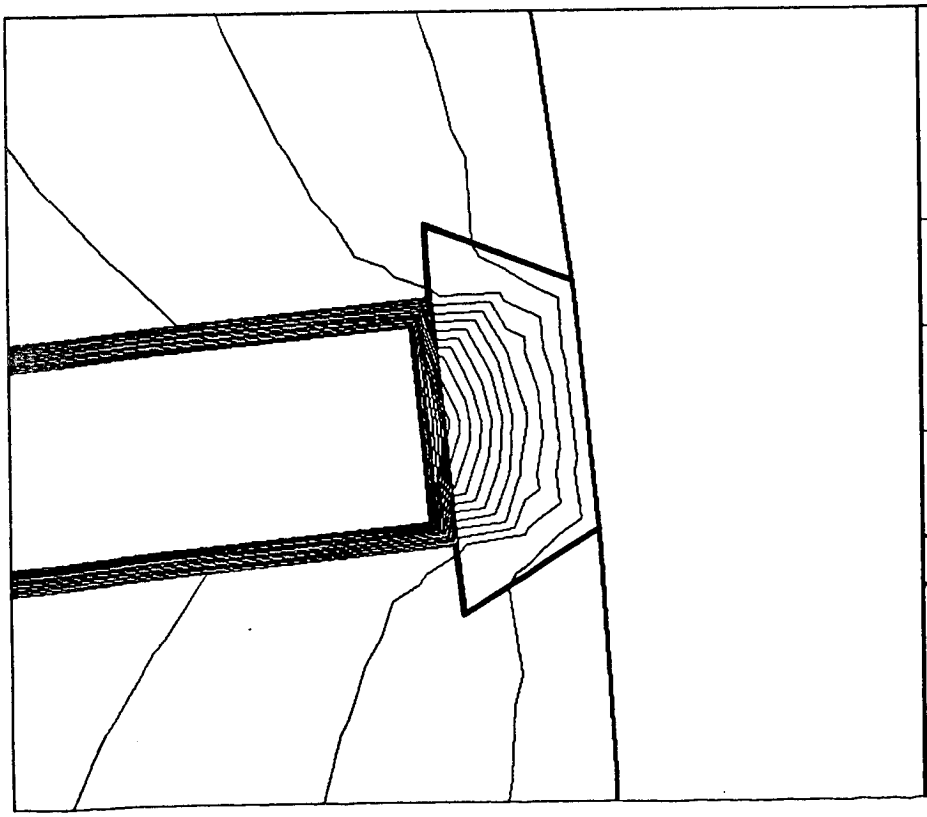
COIN BAS GAUCHE :
-4.8 -27.
COIN HAUT DROIT :
0.10E+03 63.

EXTREMA DU CHAMP B :
20 0.11E+03
EXTREMA DES ISOVALEURS
20. 0.11E+03

20 ISOVALEURS

FINIR : 0
CONTINUER : 1
VUE SUIVANTE : 2
VUE PRECEDENTE: 3
RAFRAICHIR : 4
ZOOM + : 5
ZOOM - : 6

SOFTCOPIE : 8



MODULEF : laug
MOTEUR
27/02/87
sd.mail.0
sd.coor.0
sd.b.0

289 POINTS
811 NOEUDS
378 ELEMENTS
378 TRIANGLES

COIN BAS GAUCHE :
86. 3.3
COIN HAUT DROIT :
0.10E+03 15.

EXTREMA DU CHAMP B :
20 0.11E+03
EXTREMA DES ISOVALEURS
20. 0.11E+03

20 ISOVALEURS

FINIR : 0
CONTINUER : 1
VUE SUIVANTE : 2
VUE PRECEDENTE: 3
RAFRAICHIR : 4
ZOOM + : 5
ZOOM - : 6

SOFTCOPIE : 8

```
Aucune regle n'est applicable. Arret.

Si vous voulez reprendre plus tard cette session, tapez un tiret (-)
suivi d'une ligne de commentaires. Sinon tapez un tiret tout seul :
? -moteur electrique
==> Je sauve cette session dans reprise.0

AU REVOIR !
Pour relancer DOMINO, tapez :    (:domino)
Pour sortir de LISP,  tapez :    (end)
```

REFERENCES

[AVIGNON 86] Sixièmes Journées Internationales "Les Systèmes Experts et leurs Applications", Avignon, 28-30 avril 1986.

[Bennett 79] J.S. Bennett, R.S. Engelmore, "SACON : a knowledge-based consultant for structural analysis" in "Proceedings of the Sixth International Joint Conference on Artificial Intelligence" (6th IJCAI), Tokyo, August 20-23 1979, pp 47-49.

[Bonnet 81] A. Bonnet, "Applications de l'Intelligence Artificielle : les Systèmes Experts", RAIRO Informatique / Computer Science, vol. 15, n° 4, 1981.

[Bourgault 84] S. Bourgault, M. Dincbas, J.P. Lepape, "Manuel LISLOG", Note Technique NT/LAA/SLC/159, CNET, janvier 1984.

[Chailloux 85] J. Chailloux, "LeLisp version 15, manuel de référence", Rapport INRIA, février 1985.

[Clark 82] K.L. Clark, F.G. McCabe, "PROLOG : a language for implementing expert systems" in Machine Intelligence 10 / Intelligent Systems : Practice and Perspective, published by Ellis Horwood, 1982.

[Colmerauer 83] A. Colmerauer, "PROLOG : bases théoriques et développements actuels", TSI, vol. 2, n° 4, 1983.

[Devin 85] M. Devin, "Le portage du système Le_Lisp, mode d'emploi", Rapport INRIA RT50, mars 1985.

[Dincbas 83] M. Dincbas, "Contribution à l'étude des systèmes experts", Thèse DI, Toulouse, janvier 1983.

[Fouet 83a] J.M. Fouet, "Computer aided design of mechanical parts", article soumis à l'IFIP 1983.

[Fouet 83b] J.M. Fouet, "Désapprendre à programmer", Publication Interne GR22CNRS, février 1983.

[George 86] P.L. George, "MODULEF : génération automatique de maillages", Collection Didactique, éd. INRIA, 1986.

[GRECO/GIS 87] Livre publié pour la réunion du GRECO/GIS "Calcul des Structures", Giens, 19-22 mai 1987.

[IMA 86] "Automatic study in stochastic control", Proc. Conf. Stochastic Control, Minneapolis, IMA, June 1986.

[INRIA 84] INRIA, Bulletin de Liaison de la Recherche en Informatique et Automatique n° 98, décembre 1984.

[INRIA 85] J.P. Chancelier, C. Gomez, J.P. Quadrat, A. Sulem, "Vers un Système Expert pour l'Optimisation de Systèmes Dynamiques", Septième Colloque International sur les Méthodes de Calcul Scientifique et Technique, INRIA, Versailles, éd. North Holland, 9-13 décembre 1985.

[Lagache 84] J.C. Lagache, "Présentation d'un système expert pour le code de calcul TITUS", Avignon, 1984.

[Lassalle Balier 86] G. Lassalle Balier, "Un système expert en mathématiques appliquées. Le démonstrateur MATHEXPERT", Rapport CNES CT/DTI/MS/MN, octobre 1986.

[Laurière 82] J.L. Laurière, "Représentation et utilisation des connaissances", TSI, vol. 1, n°1 et 2, 1982.

[Massé 83] Ph. Massé, "Analyse méthodologique de la modélisation numérique des équations de la physique des milieux continus à l'aide de la méthode des éléments finis", Thèse d'Etat, Grenoble, 21 juin 1983.

[Minsky 75] M. Minsky, "A framework for representing knowledge", in "The psychology of computer vision", P.H. Winston, ed. McGraw-Hill, 1975.

[MIT 83] The Mathlab Group, Laboratory for Computer Science, MIT, "MACSYMA : Reference Manual", Version 10, December 1983.

[MODULEF 86] M. Bernadou, P.L. George, A. Hassim, P. Joly, P. Laug, A. Perronnet, E. Saltel, D. Steer, G. Vanderborck, M. Vidrascu, "MODULEF : Une bibliothèque modulaire d'éléments finis", éd. INRIA, 1986.

[Mulet-Marquis 86] D. Mulet-Marquis, P. Benhamou, "ALOUETTE : un environnement logiciel pour l'utilisation des bases de connaissances", Rapport EDF HI/5302-02, janvier 1986.

[NAFEMS 86] John Barlow, G.A.O. Davies, "Selected FE Benchmarks in Structural and Thermal Analysis", National Agency for Finite Element Methods & Standards, August 1986.

[Rechenmann 85] F. Rechenmann, Ph. Vignard, "Outils de développement des systèmes experts et bases de connaissances centrées objets", Avignon, 1985.

[Robinson 82] J.A. Robinson, E.E. Sibert, "LOGLISP : an alternative to PROLOG" in Machine Intelligence 10 / Intelligent Systems : Practice and Perspective, published by Ellis Horwood, 1982.

[Roussel 75] P. Roussel, "PROLOG : manuel de référence et d'utilisation", Université d'Aix-Marseille 2, Groupe d'Intelligence Artificielle, 1975.

[Saurel 85] C. Saurel, "Système expert d'assistance aux réalisateurs de logiciels scientifiques. Présentation de la maquette PELAGIE", Rapport ONERA/CERT/DERI, octobre 1985.

[Trau 85] P. Trau, "SYMATRAU : système expert de maillage tridimensionnel automatique, Avignon, 1985, pp 1165-1176.

[Vignard 85a] Ph. Vignard, "CRIQUET version 2 : un outil de base pour construire des systèmes experts", Rapport de Recherche INRIA n°380, mars 1985.

[Vignard 85b] Ph. Vignard, "CRIQUET : un outil de base pour construire des systèmes experts, version 5, manuel d'utilisateur", Rapport Technique INRIA n°64, décembre 1985.

[Winston 84] P.H. Winston, B.K.P. Horn, "LISP", ed. Addison-Wesley, 1984.

[Zanon 84] G. Zanon, "PROLISP : manuel d'utilisation", rapport n° 1/3617/DERI, ONERA/CERT, 1984.