Outils pour le calcul scientifique

B. Rousselet [1] [2]

septembre 1999

1. *Laboratoire de Mathématiques, Parc Valrose, F 06108 Nice, Cédex 2, email :* br@math.unice.fr http://math.unice.fr/membres/br.html

2. *réalisé par l'auteur avec l'aide technique de A. Omrane et le support financier du P.E.C. Tempus No 07 228-94 "Shape optimal design, application to structures and environment"*

Table des matières

Chapitre 1

Introduction

1.1 Introduction générale

1.1.1 Calcul scientifique

Le calcul scientifique a recemment émergé comme une discipline en plein essort tant du point de vue de la recherche que son utilisation industrielle. Il s'agit de recherches et développement d'algorithmes pour la résolution de problèmes d'origine scientifique et ceci en utilisant les moyens des ordinateurs du moment.
La mécanique est historiquement une des sciences qui a le plus utilisé les possibilités des ordinateurs; c'est ainsi que la mécanique numérique est devenue une discipline assez autonome. Mais depuis les années 60 avec le développement des méthodes d'éléments finis pour résoudre des problèmes de mécanique des milieux continus, de nombreuses autres sciences se sont apperçues des possibilités de résolution numérique de modèles mathématiques adaptés à leur discipline: la physique et la chimie bien sûr mais aussi la biologie, la médecine même si dans ces deux derniers cas les problèmes sont tellement compliqués qu'une utilisation fiable est encore hors de portée (pensons par exemple à la mécanique du coeur: interaction fluide (le sang), solide (les muscles) mais avec des lois de comportement complexes et la necéssité de calculs tridimensionnels en l'abscence de symétries parfaites et en présence de phénomènes dynamiques !).

1.1.2 Fortran

Le langage Fortran a été le premier langage standardisé et ceci en 1966: le fortran 66.
Cette ancienneté permet à des informaticiens non avertis du calcul scientifique de propager l'idée que le langage Fortran est un vieux langage inadapté aux matériels et problèmes actuels.
Cela serait en partie exact si le fortran était resté standard 66 ; une modification modeste a été réalisé par la norme fortran 77 (en 1977-78); ce nouveau standard a été vraiment utilisé au début des années 80 alors que dans les comités de standardisation (T50...) on préparait un nouveau standard fortran $8x$; cependant les comités de normalisation ont eu beaucoup de mal à se mettre d'accord sur les évolutions souhaitées du langage et ce n'est finalemet qu'en 90 qu'un accord sur le contenu technique s'est dégagé pour être publié en 1991.
Ce nouveau standard 90 est entièrement compatible avec le standard 77 tout en intégrant de nouvelles fonctionalités que l'on peut rencontrer dans d'autres langages avec en plus des fonctionalités adaptés au traitements numériques. Et l'évolution continue avec fortran 95 et HPF (High Performance Fortran) conçu pour le parallélisme.

1.1.3 Historique matériel et logiciel

De façon très aproximative:

- Années 50-60: ordinateurs de taille gigantesque, salles climatisées, cartes perforées, mono tâche, peu de mémoire (quelques kilo octets) et de puissance; plan calcul en France. IBM.
- Années 70: écrans alphanumériques, multi tâches; augmentation mémoire, puissance; IBM, Deq, Bull.
- Années 80: mini-informatique, micro informatique,pc, macintosh; stations de

travail apollo, sun, hp; terminaux X, X-window; UNIX; reśeaux, courier électronique.

– Annés 90: les prix baissent et les prestations montent.

1.1.4 Bibliothèques, graphiques

Dans le cadre de la formation de jeunes scientifiques, l'outil numérique est devenu incontournable mais l'outil du langage du fortran est loin d'être le seul utile.

Citons d'abord l'utilisation de bibliothèques de sous-programmes: elles sont très nombreuses sur le marché d'une part et d'autre part on en trouve en accés libre à travers le réseau.
C'est la présence de cet immense gisement de sous-programmes qui contribue au développement et à la pérénité du fortran.
Dans les bibliothèques de programmes, une place à part doit être faite aux outils graphiques qui sont devenus des partenaires indispensables des résultats numériques même s'ils doivent être maniés avec précaution dans leur interprétation.
Pour illustrer ces deux points nous avons choisi la bibliothèque H.S.L. 12 de A.E.A. Technology qui propose des sous-programmes dans toutes les questions de base de l'analyse numérique jusqu'à la résolution de systèmes linéaires creux du type éléments finis et de l'optimisation avec contraintes non linéaires.
Pour le graphique il est commode dans le cadre universitaire d'utiliser les programmes de *Gnuplot* et *xfig* du domaine public.
Mais le traitement graphique de résultats numériques peut être aussi réalisé avec la partie graphique de logiciels de calcul formel comme MAPLE.
Mais un logiciel de calcul formel peut aussi être utilisé pour aider à la réalisation de programmes fortran de modélisation de mise en équation et d'approximation numérique; on peut ensuite engendrer directement les formules utiles en langage fortran; toutefois je regrette pour l'instant qu'il n'engendre que des instructions au standard 77 !
Le logiciel de calcul numérique scilab (logiciel domaine public de l'INRIA): à la fois langage interpréte de programmation, bibliothèque de calcul numérique et outils

graphiques. Dans ce document à vocation pédagogique nous essayons de donner des points d'entrée à ces différents outils en mettant l'accent sur scilab et fortran 90.

1.2 Nécessaire de survie

1.2.1 Unix

Les Fichiers

cat

cd

cp

df

du

ls

mkdir

more

mv

pwd

rmdir

rm

exercice 1 *Utiliser ces commandes*

Pour Imprimer

lpq

lpr

lprm

exercice 2 *Utiliser ces commandes*

Diverses commandes

env

find exemple: find br -name "uran*" -type f -print

cherche dans tous mes répertoire les fichiers qui commencent par 'uran' et imprime leurs noms; on peut chercher des répertoires avec '-type d'

kill

man

ps

who

exercice 3 *Utiliser ces commandes*

La Compression

gunzip

gzip

tar

exercice 4 *Utiliser ces commandes*

Le Réseau

exmh

ftp

finger

mail

netscape

rlogin

exercice 5 *Utiliser ces commandes*

Le système X et le graphique

xcalc

xclock

xdvi

xfig

xterm

a2ps

gnuplot

gv

exercice 6 *Utiliser ces commandes*

Saisie de texte

emacs

latex

1.2.2 Démarer avec emacs

Le menu de emacs contient un help avec un tutorial, mais voici les commandes de base.

entrer: taper 'emacs'

sortir C-x C-c

C-x C-f

C-x C-s

C-x C-w

C-x d Edition de répertoire.

En edition de repertoire :

- 'f' pour prendre un fichier sous editeur ou pour aller dans un sous-directory ou pour remonter avec '..';
- compresser 'Z';

- copier 'C';
- marquer à effacer 'd'; ' 'pour marquer à effacer les fichiers ' '
- pour exécuter effacement 'x';
- marquer 'm';
- changer 'owner' 'O'
- deplacer 'R'
- uuencoder un fichier: 'U'(pour envoyer par 'mail')

C-g

C-s

C-r

C-b et C-f

C-p et C-n

C-a et C-e

C-d

C-k

C-w

Esc %

C-x 2

C-x 1

C-x C-b

ESC g

exercice 7

1. *Utiliser emacs; pour vous former, utiliser le menu 'help' et dedans, le menu 'tutorial'.*
2. *Réaliser un petit fichier en latex pour les principales commandes de macs.*
3. *Réaliser un petit fichier en latex pour charte de l'étudiant en mim avec: Respecter les camarades et enseignants: arriver à l'heure, éviter de blaguer avec les voisins; Rendre les devoirs aux dates voulues; Apprendre les cours et td au fur et à mesure; ne pas attendre l'époque des partiels; apporter son dictionnaire d'anglais pour les TD de programmation; renforcer son anglais chez soi! Decider rapidement si je souhaitre faire un stage industriel. Travailler en groupe. Respecter la charte du bon usage informatique.*

Suggestions:utiliser itemize,

Chapitre 2

L'ordre make

Notes rédigées par R. Grin, département d'Informatique, UNSA

Chapitre 3

Introduction à Latex

Notes rédigées par R. Grin, département d'Informatique, UNSA

Chapitre 4

Début en fortran

4.1 Introduction à l'utilisation du *fortran*

Nous avons dit dans l'introduction que le standard 90 était compatible avec le standard 77; avec toutefois deux exceptions:
- le standard 90 n'est pas compatible avec les <u>extensions</u> diverses proposées par les compilateurs f 77 du marché ! même si certains compilateurs f 90 peuvent proposer certaines extensions "classiques " de f 77 cela n'est pas standard.
-la présentation des instructions; ce dernier point très simple mérite des détails car les étudiants sont souvent perplexes voire perdus dans cette affaire.

4.2 La présentation des instructions

Un peu d'histoire d'abord ! Ceux qui ont programmé dans les années 60 et au début des années 70 ont connu l'usage des cartes perforées; c'était l'époque du fortran 66; on utilisait non pas un terminal (encore moins un terminal X) mais une perforatrice de cartes et un lecteur de de cartes ! Dans une salle ou un couloir on se mettait dans une file d'attente derrière les perforatrices munies de clavier (analogue à celui d'une machine à écrire.... mécanique); avec ce clavier on perforait des cartes: c'est à dire que l'on mettait une instruction par carte à l'aide de petits rectangles perforés qui seront ensuite lues de manière optique dans le lecteur de cartes.

Les jeunes lecteurs (ou les moins jeunes qui à l'époque ne fréquentaient pas ces lieux...) doivent imaginer le bruit qui régnait dans ces lieux et qui serait insupportable maintenant; car à côté des perforatrices et des lecteurs de cartes se trouvait l'imprimante; celle ci n'était pas de type laser ni à jet d'encre mais des chaines de caractères métaliques venaient frapper la feuille de papier qui se présentait sur un rouleau.
Epoque héro¨ique où tout débutant un jour a fait tomber par terre dans le désordre son paquet de cartes avant de les avoir numérotées... les remettre dans l'ordre pouvait prendre des heures; mais il faut aussi rappeler qu'un calcul qui nous fait attendre 8 à 9 secondes devant notre terminal (attente insupportable) prenait alors des heures et que les résultats numériques ne pouvaient se voir que sur une feuille de papier; quand aux sorties graphiques c'etait souvent pour le lendemain!
A l'époque en France, on voyait rouler des "4 chevaux " et des "dauphines "; celui qui voulait faire du calcul numérique perforait ses instructions sur les 80 colonnes de ses cartes! cette petite introduction humoristique, nostalgique est destinée à vous faire sentir cette contrainte devenue insupportable aux jeunes programmeurs: taper ses instructions sur 80 colonnes! Car autrefois on ne pouvait réaliser que 80 perforations sur une carte!
Mais pourquoi parler de cela puisque c'est du passé! Tout simplement à cause des bibliothèques de programme écrites en 77 et aussi parceque de nombreuses équipes, en particulier industrielles, ne sont pas encore passées "au 90"! Il faut donc se retrouver au milieu
d'instructions en format fixe.
Tordons le cou à une mauvaise interprétation: un programme écrit avec toutes les nouvelles fonctionnalités $f90$ peut être écrit avec des instructions en format fixe de même que l'on peut utliser les seules instructions du $fortran$ 77 avec un format libre et dans ce dernier cas donner le tout à un compilateur f 90; ces deux possibilités ne sont bien sur pas conseillées sauf raisons particulières!
Assez de discours voici comment se présente une instuction en format fixe!

$$\text{colonnes : } \underbrace{\text{1 à 5}}_{\text{étiquettes}} \quad 6 \quad \underbrace{\text{7 à 72}}_{\text{corps de l'instruction}} \quad \underbrace{\text{73 à 80}}_{\text{numéro de l'instruction}}$$

en colonne 6 on met un caractère de ligne suite.

Le champ 73 à 80 n'est pas lu par la machine, il sert seulement au programmeur maladroit qui fait tomber son paquet de cartes !

Le champ 1 à 5 contient une étiquette mise devant une instruction; cette étiquette permet de se déplacer dans le source du programme à l'aide d'instructions que nous rappelons ici; il faut savoir les lire mais il est tout à fait déconseillé de les utiliser ! Ces instructions sont désuètes ou périmées (voir §4.4) .

• Indiquons d'abord que C en première colonne indique en *f* 77 que les autres caractères de la ligne sont des commentaires.
• Dans la suite nous encadrons les instructions typiques du standard 77 de deux lignes C 77.

L'instruction aller à une étiquette:

```
C 77
go to 12
C 77
```

lors de l'execution du programme il fallait sauter à la ligne qui contient 12 comme étiquette par exemple:

12 print∗, ceci est maintenant à éviter.

A noter que cette étiquette 12 peut se trouver avant ou après l'instruction `go to 12` ; l'abus de l'usage du `go to` peut rendre un programme inestricable, d'où son abondon.

• L'instruction de test arithmétique; voici un exemple.

```
C 77
   if(a) 7,8,2
7  print*,'a<0'
    go to 5
```

```
8    print*,'a=0'
     go to 5
2    print*,'a>0'
5    print*,'ça continue'
C 77
```

Le programme se passe de commentaires: suivant le signe de la variable a , on saute à l'instruction dont l'étiquette est indiquée sur la ligne du if!
Pour séparer les 3 éventualités il convient de mettre un go to!
l'instruction de test logique (voir §2.4.3.) renvoie aux oubliettes ce schéma mais on le trouve encore de nombreux programmes!

• Un caractère de ligne suite est situé en colonne 6: il permet d'écrire une instruction sur 2 lignes (avec possibilité de continuer sur plusieurs lignes).
• Les identificateurs peuvent avoir plus de 6 caractères, ce qui interdisait parfois des noms clairs pour les variables.

Les instructions en format libre:
rappelons que l'on peut programmer toutes les fonctionalités du *fortran* 90 en utilisant le format fixe décrit ci- dessus; toutefois il faut avoir de sérieuses raisons car:
- il existe des utilitaires qui convertissent du format fixe au format libre (exemple: CONVERT du CERN); ce qui permet de developper un vieux programme avec le format libre.
- on peut utiliser un vieux sous- programme écrit en format fixe avec un programme principal et d'autres sous programmes écrits en format libre.

Dans le format libre:
- les identificateurs peuvent avoir jusqu'à 31 caractères
- les lignes font au plus 132 caractères (au lieu de 80)
- les numéros des colonnes n'ont aucune importance
- les lignes de suite sont indiquées par & situé à la fin de la ligne que l'on souhaite

poursuivre

• on peut mettre ! à la fin d'une instruction: tout ce qui suit est un commentaire; on peut aussi le mettre en début de ligne à la place du C du format fixe

• on peut mettre deux instructions sur la même ligne en les séparant par ;
ex: x=1. ; y=2.
mais les instructions se terminent par la fin de la ligne

• les *blancs* sont significatifs alors qu'ils ne l'étaient pas en *f* 77:
c'est ainsi que DO i=1.10
est interprété en 77 comme la variable DOi initialisé à 1.10 au lieu d'être compris comme une instruction de boucle; ceci est détecté à la compilation *f* 90.

Dans la suite nous présentons les éléments de base de *f* 77 et de *f* 90; insistons: toute la syntaxe *f* 77 est compatible avec *f* 90 mais il est souvent conseillé ou plus commode d'utiliser la syntaxe ou les concepts *f* 90.

exemple: longueur = 3. + y !*on ajoute 3. à y*

ATTENTION: 1) on ne peut pas mélanger le format fixe et le format libre dans une même unité de programme, mais deux unités de programme peuvent être rédigées chacune dans un format différent de l'autre!

2) lors de la compilation il faut s'assurer que le compilateur comprendra l'option choisie pour le format des instructions; cela n'est pas prévu dans le standard: toutefois les mots- *fixed on- free* peuvent être fournis, par exemple, pour le compilateur NAG ou pour le compilateur "dec- α"; de plus pour ces deux compilateurs si le fichier fournit porte le suffixe ".f " le compilateur comprend qu'il est de format fixe et inversement un fichier ".f 90" est interprété en format libre.

3) Si lon n'a pas de pratique du f77 et que l'on doit modifier un programme écrit en f77, il est recommnadé de le convertir au format libre; voir paragraphe **??**

Signalons enfin un détail: on peut donner un nom à un programme:
exemple
program mon_fichier_programme

```
  real longueur_rectangle, largeur_rectangle, &
surface_rectangle ! voici des noms explicites
  largeur_rectangle=15.
  longueur_rectangle=largeur_rectangle +3
  surface_rectangle=longueur_rectangle * largeur_rectangle
    print*,'surface_rectangle =', surface_rectangle
end program mon_fichier_programme
```

Pour finir voici quelques références:
- *Delanoy* [?] un livre très clair en français, beaucoups d'exemples mais malheureusement aucun exemple numérique,
- *Metcalf & Reid* [?] Destcription complète du langage par deux membres de comités de normalisation.
- *Dodson Z.* [?]: un document en accés libre sur le réseau, il est fournit avec de nombreux exemples numériques.

Chapitre 5

Déclarations, tableaux, Entrées-sorties en fortran 77 et 90

5.1 Déclarations des types prédéfinis

Comme language dédié au calcul numérique le *fortran* possède des types prédéfinis: integer, real, logical, complex ainsi que le type character pour les chaînes de caractères.

Type implicite: courant en f77 (mais à éviter en f77 comme en f90): par convention et sauf mention du contraire (en relation avec l'usage mathématique), toutes les variables dont le nom commence par I, J, K, L, M, N sont des variables entières et toutes les autres sont des variables réelles (simple précision).
C'est ainsi que
LONG est implicitement *entier*
ARETE est implicitement *réel*
il était courant d'utiliser
ALONG pour désigner implicitement une variable réelle
en rapport avec la variable mathématique LONG.
de même NARETE pouvait désigner une variable entière par exemple:
nombre d'arête.
mais on pouvait déclarer explicitement

```
integer arete
real long
```

Avantage: pas de déclarations à faire

Inconvéniant majeur: le compilateur ne peut pas détecter les fautes de frappe:

```
NBSOM = 10
IB = NBZOM + 2
```

avec la faute de frappe NBZOM n'est pas initialisé, il contient vraisemblablement zéro et IB vaut donc 2 au lieu de 12 ! Ce genre d'erreur est très difficile à trouver dans un programme de plusieurs centaines de lignes !

Le typage implicite est à proscrire : commencer toutes les unités de programme par:

```
IMPLICIT NONE
! déclarer ensuite les variables utilisées
INTEGER ARETE
REAL LONG
LOGICAL BOUBOULE
CHARACTER (LEN=10) UN_MOT
```

•constantes symboliques : elles sont déclarées et initialisées en même temps et ne peuvent être modifiées dans l'unité de programme.

En f 77:

```
parameter (ncol = 5)
integer ncol
```

En f 90:

```
integer, parameter :: nombre_colonnes = 5
```

L'usage principal des constantes symboliques est de dimensionner les tableaux.

Remarque importante: les instructions d'organisation (dont les instructions de type) doivent être placées avant les instructions exécutables.

5.2 Initialisation et erreurs classiques

En fortran, toutes les variables utilisées doivent être initialisées: c'est à dire leur donner une valeur initiale; par defaut le contenu est aleatoire et très souvent nul.

exercice 8 1. *Ecrire un petit programme qui déclare une variable réelle et l'affiche sans initialisation (dernière instruction end; compiler avec f90 nomfich.f90 et taper a.out pour exécuter). C'est peut être le moment de dire qu'un programme doit se terminer par l'instruction* END*; d'autre part pour votre premiere utilisation:*

   (a) *dans votre editeur de texte, sauver votre fichier qui contient le fortran; le fichier doit avoir le suffixe .f90 ; exemple:* toto.f90.

   (b) *Compiler le programme avec le compilateur f90: taper l'instruction* f90 toto.f90

   (c) *Le compilateur crée un fichier a.out; pour lancer l'exécution, il suffit de taper l'instruction* a.out.

2. *rajouter*

```
y = 1/x
print*,'x et y',x,y
```

   que conseillez vous de rajouter?

3. *rajouter*

```
\verbatim
x=2/3; y=4/3
print*,'x=',x,'  y=',y

interpr{\'e}ter le resultat; comment conseillez vous de modifier le r{\'e}sulta
```

5.3 Les tableaux d'étendue fixe et les opérations globales sur les tableaux

5.3.1 A quoi servent les tableaux?

Les objets mathématiques comme les vecteurs, les matrices sont présents dans de très nombreux algorithmes. Ils sont programmés à l'aide de tableaux. Pour la rapidité de mise au point il faut prendre de petits tableaux (2 à 5 éléments); pour la resolution numérique d'équations aux dérivées partielles on peut avoir besoin de tableaux de plusieurs dizaines de miliers d'éléments. Pour pouvoir changer facilement les tailles de tabeaux on utilise des contantes symboliques; mais cela doit être fixé à la compilation; la nouveauté très attendue du fortan 90 est de pouvoir fixer la taille d'un tableau à l'exécution! Ce sont les tableaux allouables ou dynamiques.

5.3.2 Les tableaux d'étendue fixe

1. en 77: `dimension t(5)`

   `integer t`

   ou

   `integer t(5)`

   itemen 90: `integer, dimension (5) :: t`

   mais on peut avoir des indices négatifs (ce qui était impossible en 77).

   `integer, dimension (-1:3) :: t 1`

   indique que l'indice du tableau va de -1 à 3.

• L'étendue : taille des colonnes d'un tableau,
ex `t` et `t1` ont même étendue.

• Le rang : nombre d'indices d'un tableau (1 à 7 en f90) (attention ceci n'est pas le sens mathématique du rang d'une matrice) ex un vecteur est de rang 1

une matrice est de 2

une suite finie de matrices peut se mettre dans un tableau de rang 3.

• profil (shape en anglais) : liste des étendues

ex `dimension (-1:10 , 0:10) :: t2`

`t2 a pour profil -12, 11`

• taille (size en anglais) : produit des étendues c'est à dire nombre d'éléments du tableau.

• Dans une unité de programme, on peut fixer les étendues de plusieurs tableau avec des constantes symboliques

ex :

```
integer, parameter :: nco = 5
real, dimension (nco) :: t,s,u
real, dimension (nco, 3*nco + 2) :: t2, t3, s2
```

Avantage: les étendues de plusieurs tableaux sont faciles à modifier
Inconvéniant: les étendues sont fixées dès la compilation mais on peut le faire à l'exécution en f90: voir tableaux allouables au §2.2.3.

5.3.3 Opérations globales sur les tableaux (en f90)

En dehors de l'allocation dynamique de la mémoire (voir tableaux allouables), une grande nouvauté de f90 est constituée par ces opérations globales.
C'est ainsi que l'on peut initialiser les tableaux `t1` et `t2`, `u` ci-dessus avec la même valeur par:

```
t2=0. ; t3=1.E+5 ; u=1.E-2
```

donc sans aucune boucle; mieux pour un tableau de rang 1, comme t ci-dessus on peut utiliser:

```
t = (/4. , 5., 0., 0., -1.E+4/)
```

On peut déclarer et initialiser en même temps:

```
real, dimension (5) :: t = (/4. , 5., 0., 0., -1.E+4/)
```

mais on peut aussi écrire:

```
s = u + t      ! cela aditionne terme à terme;
               ! c'est à dire réalise l'addition vectorielle


s2 = t2 + t3     ! mais aussi l'addition matricielle


s2 = t2 * t3     ! Attention: cela réalise le produit
                 ! terme à terme et non le produit matriciel


s2 = 10 * t2       ! réalise bien le produit de la matrice t2 par 10.
```

Sections de tableaux:

```
t=t2(:,2)    ! la 2$^{\mbox{i{\`e}me}}$ colonne
             !  de t2 est mise dans le vecteur t\\
t3=t2(2:4, 3:5) !  sous matrice de t2, mise dans t3\\
t2(1,:)=(/1., 3., 2., 5.7,1.e-3/) ! initialisation de la ligne 1 de t2 \\
```

Exercice: on peut écrire t3 = exp(t2) le programmer et comparer avec $s2 = \sum_{n=1}^{m} \frac{t2^n}{n!}$; conclusion?

(s'inspirer du programme allocatable matrices de Dodson [?]; voir aussi ex. fin §2.3.3.).

Mieux on peut écrire:

```
where (t>0)
u = sqrt (t)    ! quand t(i)>0 on fait u(i)=√t(i)
elsewhere
```

```
u = 13. 24     ! quand t(i)<0 on fait u(i)= 13.24
endwhere
```

Nombreuses fonctions intrinsèques pour les tableaux, parmis lesquelles

```
x=DOT_PRODUCT (t,u)     ! produit scalaire de deux vecteurs
S2=MATMUL(TRANSPOSE(T2),T3)          ! le produit matriciel au sens mathé-
matique
! TRANSPOSE forme la transposée de T2; noter que les dimensions des
! matrices doivent être compatibles pour exécuter le produit.
x=MAXVAL(t)                  ! calcul de la valeur maximum d'un tableau
y=MINVAL (t)                 ! idem minimum
z=SUM(t)                     ! nomme des éléments d'un tableau
```

Remarque : si c'est utile pour d'autres développements, on peut surcharger * pour qu'il comprenne le produit matriciel; voir §3.4.2.
Remarque : Voir et utiliser les procédures intrinsèques du §4.3 .
Remarque : La comodité des opérations globales doit inciter à ne pas oublier d'initialiser les tableaux comme toutes les variables utilisées par un programme; SANS INITIALISATION UNE VARIABLE A UN CONTENU INDÉFINI; c'est très souvent zero.

5.3.4 Tableaux allouables, dynamiques (f90 seulement)

En f90 on peut réserver la place de tableaux à l'exécution et la libérer en cours d'exécution.

Exemple:

```
real, allocatable :: t2(:,:)
integer n
  read*, n
  allocate(t2(n,n))
     ! lire et imprimer t2
```

```
    ! réaliser les opérations souhaitées;
  deallocate (t2) ! quand on n'en a plus besoin
end
```

Exercice 1) prendre le programme Fahrenheit-Celsius de Dodson [?] ; mettre les températures dans un tableau allouable.

2) programme allocatableMatrices; rajouter des entrées-sorties de matrices, addition et multiplication de matrices.

5.4 Instructions d'entrées- sorties et boucles

5.4.1 Entrées sorties de variables

Simple et rapide: format libre (ou sans format)

```
read*, t      ! tableau (en f90 seulement) ou simple variable
print*, t     ! tableau (en f90 seulement) ou simple variable
```

inconvénient: inutilisable pour les données volumineuses, en particulier en sortie; pour une matrice, cela affiche la transposée !

Utilisation de format: plus précis mais un peu rébarbatif.

Principe: chaque type prédéfini peut être prévu en entrées-sorties avec plus ou moins de chiffres significatifs; voici des exemples.

exemples: I3: entier avec 1 signe et 2 chiffres;

I5: entier avec 1 signe et 4 chiffres;

F5.2: réel codé en "virgule " fixe: 2 chiffres après la virgule;

E12.5: le signe, E (remplace la base 10), signe de l'exposant et 2 chiffres pour l'exposant;

ex: + 0.15270E+02 devant la virgule on met toujours zéro

$\boxed{\text{Attention: tout résultat de calcul doit être sorti en virgule flottante.}}$
Car les sorties en format fixe peuvent être tronquées à gauche !

ex: 987. en format F5.2 peut sortir sous la forme +87. qui n'a plus aucun rapport ! le résultat peut dépendre du compilateur !

Variantes en f90 pour la virgule flottante:

ES12.5 comme E12.5 mais devant la virgule, 1 chiffre non nul (le S du format ES12.5 signifie scientifique);

ex: + 1.52700E+01

Les blancs sont désignés par X: ex: 3X désigne 3 blancs

Comment associer le format à un ordre d'entrée-sortie?
Deux façons de procéder suivant que les instructions sont entrées en mode fixe ou libre (le mode se référe à l'organisation des lignes de fortran, voir §1.2.).

Instructions en mode fixe.

ex. pour une variable réelle a:

```
      read(*,144) a
144   format(F5.2)
      write(*,145), a
145   format(1X, E12.5) a
```

Commentaires:

- on utilise des étiquettes (voir §1.2.)
- * signifie lecture au clavier, écriture à l'écran; pour des fichier, il convient d'avoir associé un numéro d'unité logique à un fichier physique; voir §2.3.3.
- noter la virgule flottante en sortie; pourquoi ?
- 1X désigne un caratère blanc.

- les formats peuvent se mettre à n'importe quel endroit de l'unité de programme mais il est conseillé de ne pas le faire dans la pagaille.

Instructions en format libre.

on ne peut utiliser d'étiquette, on utilise le mot clef fmt pour désigner le format:

ex: write(*,fmt='(F5.2,I3)') a,k

remarque : pour les formats compliqués à remettre plusieurs fois: utiliser une variable chaîne de caractères:

ex:
```
CHARACTER*7, forme    ! déclaration
forme='F(5.2, I3)'     ! affectation
read(*,fmt=forme)a,h   ! utilisation
```

5.4.2 Entrées-sorties de tableaux et boucles

Nous avons vu que l'entrée-sortie d'un tableau en f90 est enfantine:

```
print*, t, t2
```

Attention pour une matrice cette instruction fait apparaître la matrice transposée ! ça peut être déroutant pour les grandes matrices.

Voici comment procéder en f90; donnons un exemple élémentaire de boucle !

```
 do i=1,m
  do j=1,n-1
   write(*, fmt='(E12.5)', advance='no') t2(i,j)
  end do
  write(*, fmt='(E12.5)') t2(i,n)
end do
```

Commentaire : si l'on ne met pas advance='no' , après impression de t2(i,j) on passe à la ligne : c'est à dire que l'on écrit un élément par ligne, ce qui est dommage pour une matrice !

A noter : les bibliothèques scientifiques proposent des sous-programmes pour les sorties de tableaux volumineux. C'est le cas de Harwell.

5.4.3 Entrées-Sorties avec fichiers

Pour les petits programmes, il suffit d'utiliser le clavier pour entrer les données et on lit ensuite quelques résultats à l'écran. Dès que l'on doit entrer et modifier plus d'une dizaine de données, il vaut mieux les saisir dans un fichier à l'aide de son éditeur de textes (par exemple emacs); de même pour les sorties, les écrire dans un fichier permet de les regarder tranquillement avec un éditeur de textes, les imprimer mais surtout en donner une présentation graphique avec gnuplot, maple, scilab.

Une particularité du fortran consiste à associer un numéro dit d'unité logique à un fichier; ceci est réalisé par l'ordre OPEN; on effectue les entrées-sorties quand on n'a plus besoin du fichier on utilise l'ordre CLOSE.
Ces deux ordres contiennent de nombreux paramètres qui ont des valeurs par défaut; voici donc un ordre avec le minimum de paramètres:
ex: `OPEN(unit=10,File='a_tracer', status='new' )`
Pour un fichier deja existant mettre old au lieu de new; unknown si l'on ne sait pas: c'est la valeur par defaut. `write(unit=10, fmt='(E12.5)', iostat=ios) (X(i), i=1,n)`
`close(unit=10, status='keep')`

Commentaires 1) choix du numéro d'unité logique: éviter les entrees ≤ 9, ils sont souvent réservés par le compilateur; c'est aussi que unit=5 est souvent réservé à l'entrée clavier et unit=6 à la sortie écran; par convention l'usage de $*$ remplace les numéros associés aux entrées-sorties clavier et écran (voir §2.3.1.).

2) nom du fichier 'a_tracer': nom à votre guise mais ne pas oublier les apostrophes car c'est une chaîne de caractères; si nécessaire, on peut utiliser une variable chaîne de caractères.

3) l'état du fichier est indiqué avec le mot clef status; dans l'odre open, on met new pour un fichier qui n'existe pas encore et old pour un fichier existant on peut aussi mettre unknown si l'on ne sait pas et c'est même le défaut, si bien que l'on peut omettre le mot clef status à l'open. Dnas l'odre close, on peut décider de le conserver avec keep ou de l'effacer avec delete.

4) dans l'ordre d'écriture, la variable ios associée au mot clef iostat permet de savoir si l'on est arrivé en fin de fichier; dans ce cas elle est négative.

5) Avec cette forme simple d'open, le fichier est d'accés séquentiel par défaut: les enregistrements sont les uns derrière les autres; il est possible d'utiliser un fichier d'accés direct. De même les enregistrements sont formatés, ce qui permet de voir le fichier à l'éditeur de texte; il est possible d'écrire dans des fichier non formatés; ces fichiers ne sont pas lisibles par un être humain mais les entrées sorties sont plus rapides et le mode non formaté est donc à conseiller pour des volumes importants de données; voir Delanoy [?].

5.5 Chaînes et expressions logiques, if logique

5.5.1 Opérations sur les chaînes de caractères

Nous avons vu au §2.1., comment déclarer une variable chaîne de caractères de nom un_mot en précisant sa longueur avec le mot clef CHARATER .
Pour désigner une sous-chaîne, la syntaxe est analogue aux sections de tableaux.

ex
```
un_mot(1:3)='abc'
un_mot(4:6)=un_mot(1:3)
un_mot(6:10)='defgh'
```

Pour accoler les chaînes de caractères, on utilise //:
si deux_mots est une variable de type caractères de longueur 20, on peut écrire:
```
deux_mots=un_mot// un_mot
```
deux_mots contient alors deux fois le contenu de un_mot

Pour les entrées-sorties, on utilise le format A:

ex read(unit=10,'(A10)') un_mot

write(*,'(A)', ADVANCE='NO') 'entrer X='

dans l'ordre write, on voit qu'il est inutile de mettre la longueur de la chaîne; advance='no' permet de laisser le curseur après le =; si l'on lit ensuite read*,X on pourra taper la valeur de X, sur la même ligne juste après le signe =.

On peu utiliser des tableaux de chaîne de caractère en les déclarant ainsi:

CHARACTER(LEN=20), DIMENSION(5,20) :: DES_MOTS

5.5.2 Expressions logiques

Le plus courant, au cours de calculs numériques est d'utiliser des valeurs logiques qui proviennent de comparaisons de valeurs numériques.

C'est ainsi que f77 disposait de

.GE.	pour désigner	$\geq$
.GT.	————	$>$
.LE.	————	$\leq$
.LT.	————	$<$
.EQ.	————	$=$
.NE.	————	$\neq$

on peut maintenant comparer deux membres avec

>=	pour désigner	$\geq$
>	————	$>$
<=	————	$\leq$
<	————	$<$
==	————	$=$
/=	————	$\neq$

ex : c'est ainsi que si bouboule est une variable logique, bouboule= x >=3.1 donne à bouboule la valeur "vraie " si x est bien supérieur ou égal à 3.1 .
Pour former des expressions logiques, on disposait déja en f77 des opérateurs sin-changés en f90:
.NOT. .OR. .AND.
qui correspondent à la négation, au ou non exclusif et au et logique.
On peut utiliser des tableaux de variables logiques et réaliser des opérations globales sur des tableaux pour faire des comparaisons.

ex real, dimension(14,3):: x,y
logical, dimension(14,3):: compare
compare = x < =y

l'élément compare (i, j) est vrai si x(i,j) est bien $\leq$ à y(i,j).

5.5.3 Tests sur des expressions logiques

La construction du if logique était déja en fonction en f77:

Exemple 1
```
logical bouboule, babale\\
if (bouboule) then\\
print*,'bouboule est', bouboule\\
elseif(babale) then\\
print*,'babale est', babale\\
else\\
print*,'bouboule et babale sont',
le , babale\\
endif\\
```

Cette construction peut se simplifier si l'on n'a qu'un cas à considérer:

Exemple 2
```
logical bouboule, babale\\
if (bouboule) then
```

```
    print*,'bouboule est', bouboule
  endif
```

ou même si l'on n'a comme ici qu'une instruction à effectuer:

Exemple 3 `if (bouboule) print*,'bouboule est', bouboule`

Mais l'on peut aussi distinger différents cas à l'aide d'une variable entière

ex
```
integer num
      select case (num)
      case(-4:0)
        print*,'num est compris entre -4 et zéro'
      case(3)
        print*,'num est égal à 3'
      case default
        print*,'num n'est pas égal à 3 et pas compris entre -4 et zéro'
      endselect
```

Le bloc select case est ainsi une présentation structurée qui évite l'emploi de goto et du goto calculés qui sont périmés voir §4.4.
Mais le goto, était très utile pour sortir d'une boucle avant que le nombre de tours de boucle soit effectué; les instructions CYCLE et EXIT permettent de rendre cela plus lisible, on association avec un nom donné à la boucle: ex:

```
  extérieure: do   i=1,m
  intérieure: do   j=1,n
        if (bouboule)   CYCLE
        if (babale)   EXIT exterieure
! les instructions à executer      end do   intérieure
      end do   extérieure
```

dans le premier cas quand bouboule est vrai CYCLE fait recommencer la boucle intérieure en abondonnant la fin des instructions, quand bouboule est fausse on continue et si babale est vraie on sort de la boucle extérieure, c'est à dire qu'on abondonne les deux boucles !

Chapitre 6

Les Sous-programmes en fortran 77 et 90

6.1 Introduction

En fortran 77, on ne disposait que de sous-programmes EXTERNES, écrits et compilés independemment du programme d'appel; en fortran 90, on dispose comme dans de nombreux langages de sous-programmes INTERNES un peu plus liés au programme d'appel; ils ont en particulier accès aux variables du programme d'appel sans passer par les paramètres; toutefois cette possibilité est dangereuse et à éviter. Les sous programmes peuvent être du type FUNCTION, dans ce cas ils s'utilisent à peu près comme des fonctions mathématiques (voir exemple dans le programme d'Euler plus loin); ils peuvent être des SUBROUTINE; dans ce cas les entrées et les sorties se font par paramètres qui peuvent être variés et nombreux (exemple au paragraphe suivant). Le passage des paramètres peut se faire avec des arguments à l'appel: le passage se fait par ADRESSE et non par valeur comme dans certains langages.
Pour transmettre un paramètre d'un sous-programme à un autre sous-programme, il faut donc le faire remonter jusqu'au programme principal ce qui est mal commode qund on utilise plusieurs couches de sous-programmes.
En f77, on utilise un COMMON pour transmettre des données d'un sous-programme à un autre; pour éviter toute confusion, il est conseillé de ne pas modifier les valeurs

des variables d'un COMMON.
Le common est une instruction périmée (§4.4.) en f90, il vaut mieux utiliser un module pour partager des paramètres voir §3.4.3.

Pour un sous-programme externe (le seul type en f77) les variables qui ne sont ni en paramètres, ni en common, ni en commun à l'aide d'un module sont des variables locales; donc pas de variable globale en fortran; en revanche un sous-programme interne peut utiliser des variables de l'unité qui le contient (dangeureux).
A noter qu'un sous-programme externe peut être situé dans le même fichier que le programme appelant ou un autre fichier; cette dernière solution est la seule commode avec un grand nombre de sous-programmes.

6.2 Interfaces

En f77, de très nombreuses erreurs se font lors de passages de paramètres; cette question est reglée en f90 par des interfaces explicites.

6.2.1 Interfaces implicites

Voici un exemple au standard f77:

```
subroutine toto(n,p)
implicit none
integer n,p
n=5
p=3*n
end
```

et un programme principal d'appel

```
implicit none
integer nn,pp
real x
nn=4
```

```
call toto(x,pp)
x=pp+1.2
end
```

L'utilisation de sous-programme comporte 2 erreurs:
- on appelle toto avec comme premier paramètre x qui est réel alors que dans toto le premier paramètre n est entier.
- on donne à nn la valeur 4 dans le programme et n=5 dans le sous-programme: ces deux variables n'ont rien à voir.
En f77 cela ne donne aucune erreur à la compilation mais ne fait peut être pas ce que l'on souhaite !
Avec ce mode d'utilisation, on dit que l'interface entre le programme principal et le sous-programme est implicite; ce qui signifie
- compilation séparée des unités de programme;
- pas de vérification des types de l'ordre ni du nombre des paramètres;
- pas de vérification des tailles de tableau.

C'est l'une des sources principales d'erreur en f77.
Ceci est toujours possible mais il est vivement conseillé d'utiliser un interface explicite en f90; ceci est possible de trois façons:
a) utiliser un bloc d'interface pour des sous-programmes externes
b) utiliser un sous-programme interne: ceci n'existait pas en f77; pour la lisibilité il est conseillé de n'utiliser en interne que de petits sous-programmes
c) utiliser un module voir §3.4. .

6.2.2 Bloc d'interface

Il suffit de mettre dans l'unité de programme d'appel d'un bloc:

```
interface
 !  mettre ici l'entete du subroutine utilise,
 !  suivi de declarations de type
 ! Aucune instructin executable
 !  suivi de l'ordre  END
endinterface
```

Répétez cela pour tous les subroutines appelés. Ce bloc d'interface doit se trouver avant les instructions exécutables.

Il est conseillé de complèter les déclarations de variables par l'instruction intent(in) ou intent(out) qui précise si la variable est une variable d'entrée (non modifiée par le sous-programme) ou une variable de sortie (sans valeur à l'entrée du sous-programme); par défaut les variables sont intent(inout): dans ce cas elles ont une valeur en entrée et sont modifiées dans le sous-programme.

Exemple 4 *(nouvelle version du programme du* §3.2.1. *avec le même sous-programme)*

```
program exemple_interface
 implicit none
 integer nn, pp
 real x
 interface
    subroutine toto(n,p)
    integer, intent(in) :: n
    integer, intent(out):: p
    end subroutine toto
 end interface
   nn=4
   call toto(x,pp)
   x = pp + 1.2
end program_interface
```

Commentaires: 1) on utilise le même sous-programme `toto`, cela doit faire une erreur, car les déclarations ne sont pas les mêmes: précaution, mettre dans l'interface le "coupé-collé" de l'entête du sous-programme et modifier éventuellement aux deux endroits.

2) le compilateur doit détecter maintenant que `x` n'a pas le bon type dans l'appel de `toto`; de même dans `toto` on modifie la valeur de `n` qui est une variable d'entrée, cela doit donner une erreur.

Remarque: Le corps d'un sousprogramme ne doit pas se trouver à l'intérieur de l'unité d'appel; c'est l'interface qui ne contient que des instructions d'organisation et qui se trouve dans l'unité d'appel.

6.2.3 Sous-programme interne

Ils ne peuvent être utilisés que par le programme d'appel et sont séparés par `CONTAINS`; un sous-programme interne a automatiquement un interface explicite (bloc d'interface inutile) et se trouve dans le même fichier que l'unité de programme qui le contient; un sous-programme externe peut avoir lui même des sous-programmes internes, mais un sous-programme interne ne peut pas avoir d'autre sous-programme interne. Un sous-programme interne a accés à toutes les variables de l'unité hôte mais cela ewst à éviter!

Exemple 5 *voir program Triangle de Dodson [**?**].*

Exemple 6

1. *Exécuter le programme ci dessous (copier le programme chez vous avec l'ordre unix* `cp`.
2. *pour le cas jx=1 ou 2 ou 3, résoudre $x' = Ax$ et comparer avec la solution exacte calculée avec exp[onentielle (voir exercice du chapitre; cas où A est diagonale avec de petits termes hors diagonaux. 2); dans le cas où A est antisymétrique quelle est la propriétede $\exp A$? Lien avec la dynamique du corps rigide.*
3. *Résoudre dans le cas où A a une valeur propre grande négative et une proche de zéro; comment devez vous choisir le pas?*

4. *Résoudre avec* $f = A * x + \varepsilon\phi$

```
program euler_explicite
! programme elementaire de resolution numerique d'equation differentielle par
!la methode d' Euler explicite
implicit none
integer, parameter:: jx=1
integer nmax, n
real:: t,tau
real, dimension(jx):: x
real, dimension(jx,jx):: a
open(unit=10, file='tracer')
t = 0.; x =1.
print*, 't= ',t, 'jx= ', jx, 'x= ', x
print*,  'tau= ', 'nmax= ','matrice a= '
read*,  tau, nmax, a

do n =1,nmax
x = x + tau*f(t,x,a)
t = t + tau
print*, t,x , exp(1.)
write(10,fmt='(2e12.5)') t,x(1)
enddo

contains
function f(t,x,a)
implicit none

real:: t
real, dimension(:):: x
real, dimension(:,:):: a
real, dimension(size(x)):: f
f= matmul(a , x)
```

```
end function

end program euler_explicite
```

6.2.4 Fonctions en paramètre

Pour utiliser une fonction externe en paramètre, il suffit de mettre un interface dans le sous-programme qui l'utilise; voici quelques lignes extraites du module RootFinders de Dodson

```
function findRoot(a, b, f, df, tolerance)}
implicit none}
real findroot}
real, intent(in) :: a, b }
real, optional, intent(in) :: tolerance}
interface}
function f(x)}
real f}
real, intent(in) :: x}
end function f}
function df(x)}
real df}
real, intent(in) :: x}
end function df}
end interface}
.
.
.
! instructions ex{\'e}cutables}
```

Les functions f et df peuvent bien sûr être écrites avec d'autres noms, par exemple f1, df1, f2, df2; à part ce changement de nom les instructions exécutables doivent être identiques à l'interface; c'est d'ailleurs le seul cas où l'interface n'est pas rigoureusement identique aux instructions non exécutables du sous-programme appelé. Il peut être plus commode de mettre ces fonctions dans un module: voir §3.4.3.

6.2.5 paramètres optionnels

En f90 les paramètres peuvent être omis lors de l'appel d'un sous-programme si ils ont étés déclarés avec l'attribut

`optional;`

voir le module

`rootfinders`

de Dodson au §2.9; voir aussi section sur les partages de données.

6.2.6 recursivité

Les fonctions fortran peuvent être récursives, c'est à dire s'appeler elles mêmes. Voici un exemple très simple proposé par Dodson [?]; noter que l'attribut <u>récursive</u> doit être utilisé ainsi que l'option result pour mettre le résultat; <u>le nom de la fonction ne doit pas apparaître à gauche du signe d'affectation =</u>;

```
recursive function factorial(n) result(f)
 integer f
 integer, intent(in) :: n
  if( n <= 0) then
    f = 1
  else
    f = n * factorial(n-1)
  end if
end function factorial
```

Une fonction récursive peut faire partie d'un processus itératif, voir par exemple newton dans rootfinders de Dodson.

6.3 Les tableaux en paramètres

6.3.1 Dimension fixe dans le sous-programme

Exemple 7

```
subroutine toto(t)
integer, dimension(5) :: t,s
t = 0.
do i=1, 5
t(i) = t(i) + i
 s=t+3.
 t=s*s
end do
end subroutine toto
```

Dans le programme d'appel :

call toto (tt)

! Attention le tableau tt doit avoir une taille supérieure ou égale à celle de t du sous-programme (ils peuvent porter le même nom); là encore aucun diagnostic à la compilation pour un interface implicite, si la taille de tt est strictement inférieure à celle de t mais des débordements d'indice à l'exécution; l'utilisation d'un interface explicite est encore plus recommandée.

Nouvauté f90 : Un sous-programme function peut avoir comme résultat un tableau; exemple: programme MatrixVector de Dodson

le résultat peut être mis:

- directement dans le nom de la fonction (voir program Triangle de Dodson)
- dans une variable de résultats:

Exemple 8 result(c) *voir dans program MatrixVector de Dodson)*

6.3.2 Dimension variable dans un sous-programme

Les tableaux en paramètres

En f77: nous ne considérons, pour mémoire, que le cas simple d'un vecteur; on passe en paramètre la taille d'un vecteur, cette taille doit être initialisée dans le programme d'appel.

Exemple 9
```
subroutine tataf(t,n)
  dimension t(n)    ! autres possibilités t(1) ou t(*) dans ce cas il prend
  t = 0.   ! la taille du tableau du programme d'appel.
  do i=1, n
  t(i) = t(i) + i
  end do
 end subroutine tataf
```

voici par exemple un morceau du programme d'appel:
```
  integer, parameter :: m = 5
  dimension tt(m)
  call tataf(tt, nn)
  print*, tt
```
En f90 ceci continue bien sûr à marcher mais il vaut mieux utiliser des tableaux de profil supposé (celle du sous-programme d'appel) ("assumed shape array ").
L'interface doit être explicite

Exemple 10
```
subroutine tatas (t2)
 real, intent(out), dimension (:,:) :: t2
 integer m, n
```

```
m = size(t2,1)
n = size(t2,2)
do i=1, m
  do j=1, n
    t2(i,j)= i**2 + j
  end do
end do
end subroutine tatas
```

Dans le programme d'appel:

```
integer, parameter :: nn = 5, mm = 3
dimension tt2 (nn, mm)
  interface
  subroutine tatas (t2)
  real, intent(out), dimension(:,:) :: t2
  end subroutine tatas
  end interface
call tatas(tt2)
print*, tt2
end
```

Les tableaux locaux

On peut utiliser un tableau automatique dans un sous-programme:
dans `tatas` on peut rajouter

```
real, dimension(size(t2,1),size(t2,2)) :: travail
```

ce tableau est local à `tatas` et disparait à la sortie de tatas: utile pour des calculs locaux au sous-programme considéré.

6.4 Structures et Modules

6.4.1 Structures ou types dérivés

En f77 il n'existe que des types de base (integer, real, complex, logical, character) et les tableaux construits à partir de ces types.
En f90, il est possible de définir de nouveaux types dits dérivés, encore appelés structures.
En les plaçant dans un module, on peut définir facilement des opérations sur ces structures.

Ex : ! on définit d'abord les diverses composantes de la structure

```
type point      ! exemple classique: le nom
                ! du point et ses deux coordonées.
character (len = 2) :: nom
real, dimension(2) :: xx
end type point
type (point) :: a, b, c , d           ! 4 variables de type point
type (point), dimension(50) :: courbe     ! tableau de structures
a % nom ='a1'       ! on initialise les composantes de la structure a
a % xx(1) = 1. ; a % xx(2) = 3.5
d = point('a4', 7.2, 8.4)          ! autre façon d'initialiser
real*, b          ! lecture: donner les trois valeurs que doit contenir b
c % nom ='a3'
c % xx(1) = - a % xx(1) ; c % xx(2) = - a % xx(2)
do i=1, 50
courbe(i) = point('i', sin (i/10.), cos (i/10.))
end do
end
```

6.4.2 Modules et Structures

Les modules peuvent être utilisés pour partager les données (ce qui remplace les common), éviter de dupliquer des interfaces, réaliser des procédures génériques ou surcharger des opérations.
Inutile de mettre les modules dans un bloc d'interface, leur interface est automatiquement explicite.

Les modules permettent de définir commodément des opérations un des structures (comme dans les langages orientés objets).

Exemple 11 `module symetrie_de_point`

```
 implicit none
 type pointn}  ! type different de point,
!                on met un num{\'e}ro au lieu d'un nom}\\
 integer numero
 real x, y
 end type point
 contains  ! m{\^e}me instruction que pour un sous-programme interne
!            d'une autre unit{\'e} de programme
 function symetrie (p,n)}
 type(point) :: p
 type(point) :: symetrie
 integer n
 symetrie = point(n, -p\%x, -p\%y)
 end function symetrie
 end module symetrie\_de\_point

 program utilisation_symetrie
 use symetrie_de_point
 implicit none
 ! le mot use doit {\^e}tre mis pour l'utilisation de tout module
 type (point) :: a = point(3, 2.0., 3.5), b
 ! noter qu'avec l'ordre use, on a acc{\`e}s {\`a} tous
 ! les types d{\'e}finis dans le module symetrie_de_point
```

```
  b = symetrie(a, 10)
  ! et aux fonctions et sous-programmes d{\'e}finis dans le module
 print*, a, b
 end program utilisation_symetrie
```

La définition d'une structure et d'opérations sur des variables du type de cette structure est donc aisée avec un module.
Il faut noter que pour un exemple aussi simple il aurait été commode de définir le type "pointn" dans le programme principal "utilisation_symetrie " et d'utiliser une fonction externe "symetrie "; toutefois le type pointn aurait dû être redéfini dans la fonction "symetrie ". Dans la présentation avec module, dans toute unité de programme où l'on utilise des variables du type "pointn" et des opérations sur ces variables il suffit de mettre "use symetrie_de_point" pour pouvoir déclarer ces variables et leur appliquer les opérations définies dans le module (ici seulement la symétrie à titre d'exmple).

Utilisation pratique d'un module, et édition de lien avec le programme principal:

- Les modules doivent être compilés séparément avant, l'édition de liens avec le programme principal; f90 -c symétrie_de_point.f90 la compilation du module symétrie_de_point crée un fichier symétrie_de_point.mod pour compilateur NAG ou *dec* α, et symétrie_de_point.hmo pour le compilateur de silicon; ces fichiers sont utilisés dans la phase d'éditions de liens.
- on réalise ensuite l'édition de liens: f90 pp.f90 symétrie_de_point.o
  attention le fichier .mod (ou .hmo) n'est utilisé dans aucune commande!

6.4.3 Module et partage de données et d'interfaces

On peut partager entre des sous-programmes des constantes symboliques, des variables ou tableaux ainsi que des fonctions; pour les variables et tableaux, il convient de déclarer au niveau du module (et non simplement dans des sous-programmes internes) avec l'attribut SAVE.

ex :

```
module partage

implicit none
integer, parameter :: n=5
real, dimension(n,n), save :: a
end module partage
```

On pourra alors partager a entre ces deux sous-programmes !

```
subroutine lecture
use partage
print*, n,'matrice a
read*, a
end subroutine lecture
function carre
use partage
real, dimension(n,n) :: carre
carre = matmul(a,a)
end function carre
```

Attention la fonction carre n'a pas de parametres; ce qui est inhabituel; pour l'appeler

```
d= carre()  !parentheses indispensables
```

Ceci est particulièrement valable pour l'utilisation de fonctions; on peut mettre la fonction et le programme d'appel dans un module mais l'interface reste utile (ce qui est un peur en desaccord avec le principe des modules!

```
module findroot_et_fonctions
 function f(x)
 real f
 real, intent(in) :: x
 ! instructions ex{\'e}cutables
```

```
 end function f
 function df(x)
 real df
 real, intent(in) :: x
 ! instructions ex{\'e}cutables
 end function df


 function findRoot(a, b, f, df, tolerance)
 implicit none
 real findroot
 real, intent(in) :: a, b
 real, optional, intent(in) :: tolerance
interface
 function f(x)
 real f
 real, intent(in) :: x

 end function f
 function df(x)
 real df
 real, intent(in) :: x

 end function df
end interface
 .
 .
 .
 ! instructions ex{\'e}cutables
end  function findRoot
```

on peut aussi remplacer commodément le passage d'une fonction en paramètre décrit au §3.2.4. ; voici comment procèder pour transformer cet exemple;
(a) mettre les deux fonctions `f` et `df` dans un module:

```
module functions_and_derivatives
  implicit none
  contains
  function f(x)
  .
  .
  .
  end function f
  function df(x)
  .
  .
  end function df
end module function_and_derivatives
```

(b) enlever *f* et *df* des paramètres de findroot; attention: comme df n'est plus en paramètre, il ne peut plus être optionnel et on ne peut plus utiliser l'operateur

`present`

(c) enlever l'interface de *f* et *df* dans findroot;
ce qui donne

```
function findRoot(a, b, tolerance)
  use functions_and_derivatives
  implicit none


  real findroot
  real, intent(in) :: a, b
  real, optional, intent(in) :: tolerance
  ! sauter interface
  ! suite du sous-programme inchangé
```

Dans le même esprit lorsque un sous-programme est appelé par plusieurs sous-programmes il est commode d'écrire un seul interface que l'on place dans les modules; tous les sous-programmes qui en ont besoin y ont accés par l'ordre use.

ex. reprenons l'exemple du §3.2.2 avec le subroutine toto;

```
module voici_interface
  implicit none
 interface subroutine toto (n,p)
  integer, intent(in) :: n
  integer, intent(out) :: p
  end subroutine toto
 end interface
end module voici_interface
```

et le programme d'appel devient:

```
programme exemple_interface_module
  use voici_interface
  integer nn, pp
  real x
  nn = 4
  call toto (nn, pp)
  x = pp + 1.2
end program exemple_interface_module
```

Naturellement ceci n'est vraiment intéressant que si toto est utilisé par plusieurs sous-programmes.

6.4.4 Modules et procédures génériques, surcharge d'opérateurs

Reprenons l'exemple du §3.3.1. avec 2 versions: entiers ou réels.

```
module tata_generique
  implicit none
  interface tata
  module procedure tataentier, tatareel
  end interface
  contains
```

```
    subroutine tataentier(t)
    integer, dimension(5) :: t
    t=0
    do i = 1, 5
    t(i) = t(i) + i
    end do
    end subroutine tataentier
    subroutine tatareel(t)
    real, dimension(5) :: t
    t=0.
    do i=1, 5
    t(i) = t(i) + float(i)
    end do
    end subroutine tatareel
  end module tata_generique
```

On peut alors s'en servir ainsi:

```
  program tata_et_tata
    use tat_generique
    integer, dimension(5) :: an
    real, dimension(5) :: ar
    call tata(an)
    call tata(ar)
    print*, an, ar
  end program tata_et_tata
```

On n'a pas à se soucier du type de la variable pour savoir quelle version de la fonction utiliser cela se fait automatiquement: c'est la généricité.
N.B. Ce principe de nom générique s'applique à toutes les **fonctions prédéfinies du fortran**: c'est ainsi qu'il n'est plus utile d'utiliser la fonstion DCOS pour le cosinus en double précision COS(X) fait bien le calcul en double précision dès que X est déclaré en double précision.

Surcharge d'opérateurs : elle consiste à pouvoir utiliser le symbole opératoire évident comme + ou * pour le faite agir sur des variables de type différent ou définir de nouveaux opérateurs, que l'on désigne ainsi:

.addition. pour un opérateur qui serait sensé réaliser une addition.

ex.
```
module operations
  type f_df
  real f, df
  end type
  interface operator (*)
  module procedure multiplier
  end interface
  contains
  function multiplier  (u,v)
  type f_df multiplier, u, v
  multiplier % f= u % f * v % f
  multiplier % df= u % df * v % f + u % f * v % df  ! (uv)' = u'v + uv'
  end function multiplier
  end module operations
```

Pour multiplier 2 fonctions et avoir leur dérivées on peut alors utiliser le program multiplier :

```
program multiplier
  use les fonctions; use operations
  type(f_df) :: phi, psi, produit
  ! fonc1 et fonc2 deux sous-programmes fonctions ainsi que les
  ! sous-programmes dérivées: dfonc1 et dfonc2
  phi = f_df(fonc1(x), dfonc1(x))
  psi = f_df(fonc2(x), dfonc2(x))
  produit = phi * psi
  print*, produit     ! l'opérateur * est ici unchangé
```

```
end program multiplier


module les_fonctions
 implicit none
 contains
 function fonc1(x)
 real x, fonc1
 fonc1= x**2
 end function fonc2
 function fonc2(x)
 real x, fonc2
 fonc2= sin x
 end function fonc2
 function dfonc1(x)
 real x, dfonc1
 dfonc1 = 2 * x
 end function dfonc1
 function dfonc2(x)
 real x, dfonc2
 dfonc2 = cos x
 end function dfonc2
end module les_fonctions
```

6.4.5 Erreurs possibles

Attention: entre l'instruction module et contains on peut faire des declarations sur les types de variables: ces variables deviennent globales au module; mais on ne peut mettre aucune instruction executable entre module et contains: suivant le cas mettre ces instructions dans la programme principal ou dans un sous programme situe dans le contains. De plus on ne peut mettre apres le contains que des instructions situées dans des sous-programmes ou fonctions!

6.5 Exemple d'adaptation de programme

Nous considérons quelques adaptations du module rootfinders proposé par Dodson [?] au §2.9..
Nous avons déja vu au §3.4.3. comment passer les fonctions externes f et df en les glissant dans un module; nous voulons considérer ici les transformations à effectuer pour résoudre des systèmes non linéaires

$$f(x) = 0 \qquad \text{avec} \qquad x, f \in I\!R^n$$

alors que le module rootfinders est prévu pour $n = 1$.
a) Première remarque: la méthode de Newton passe facilement en n variables mais ce n'est pas le cas de la méthode de la sécante; dans ce dernier cas il faudrait programmer une méthode quasi-Newton (Broyden, B.F.G.S. etc ...) ce qui sort du cadre de cette présentation du langage; nous supprimons donc la sécante; dans ces conditions il est inutile de demander à l'utilisateur 2 points a et b pour données initiales, un seul peut être utilisé pour Newton.
b) *dimension des tableaux*; nous avons choisi d'utiliser une constante symbolique nx définie dans le module functions_and_derivatives fournit par l'utilisateur; cela n'est pas très sûr si le module est écrit par un utilisateur non averti; une valeur par défaut pouvaît être prévue; on pourrait ainsi utiliser les tableaux allouables (§2.2.3) .
c) la fonction programmée est très simple

$$f\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \varepsilon \begin{pmatrix} x_1^3 \\ x_2^3 \end{pmatrix} - \begin{pmatrix} g_1 \\ g_2 \end{pmatrix}$$

d) le module root_newton est seulement modifié avec use functions_and_derivatives ce qui donne accèes à la constante symbolique nx pour dimensionner les tableaux; on rajoute 2 fonctions pour calculer déterminant et inverse de matrice; limité à la taille 2; nous avons rajouté une constante symbolique entière à ictmax.
e) dans findroot on dimensionne xk, le point de départ et la fonction *findroot*; on supprime l'appel de la sécante.
f) dans *Newton*, on remplace les variables par des tableaux à 1 ou 2 indices suivant le cas; on a choisi ici de se limiter à de petites dimensions ($nx \leq 4$) et d'utiliser l'algorithme de Newton en inversant la matrice dfguess;pour nx de 3 ou 4 à 9 ou

10, on peut utiliser un logiciel de calcul formel comme maple pour resoudre le système linéaire, voir [?] §5.2.1 et engendrer les instructions fortran voir [?] §12.2.1;et pour des tailles supérieures, il conviendrait d'appeler un programme de résolution numérique de système linéaire; nous testons le nombre d'itérations de Newton avec le paramètre *ictmax*, constante symbolique définie dans *root_newton*.

Remarque . à la suite de la suppression de la sécante le module rootfinders renommé en *root_newton* n'a plus beaucoups d'utilité: il gère seulement le paramètre de tolérance : tol et évite à l'utilisateur de s'occuper du compteur ict.

Chapitre 7

Utilisation de scilab

Scilab est un logiciel intégré dédié au calcul numérique; il comprend à la fois un langage de programmation et des bibliothèques de sous programmes; à noter qu'à la différence du fortran c'est un langage interprété; l'utilisation est donc assez proche d'un système de calcul formel comme maple. Scilab est développé par l'INRIA et est du domaine public: il peut eṫre trouve sur le site: http://www-rocq.inria.fr/scilab; on peut y trouver documentation et renseignements. Citons une référence récente [?].

7.1 Mise en train

Pour appeler scilab, tapez

```
scilab&
```

Dans la fenetre scilab, on peut utiliser les commandes de base de EMACS; en particulier:

```
ctrl b
ctrl p  (ou touche fleche vers haut)
```

Les commandes sont séparées par la touche 'return' ou par ';'si elles sont tapées sur la même ligne; une commande terminée par ';'ne donne aucune sortie écran à l'exécution.

exercice 9 *taper quelques instructions simples*

Les contantes

```
%i; %pi; %e; %eps; %t; %f
```

exercice 10 *manipuler quelques constantes reelles et complexes*

L'utilisation de matrices est très souple.

```
n=10
x=[1:n]
//multiplication terme a terme
y=x.*x
//produit scalaire
px=x*x
//transposition
x'
//saisie
v=[1 +3]; w=[1,3]; ww=[1+ 3];
v=5:-.25:3;
v=ones(1:4); v=zeros(1:6)
a=[1 2; 6 -3]; b=ones(2,2); d=eye(3,4)
//Attention '*' designe le produit matriciel
a*b
```

Attention '*' designe le produit matriciel mais pas en f95!

exercice 11 *Faire quelques manipulations de matrices*

7.2 Programmation

Attention, il est vivement déconseillé de taper un grand nombre d'instructions directement dans la fenêtre scilab!

- taper les instructions sous emacs
- pour exécuter le fichier

  ```
  'exec('nom_fichier')'
  ```

- Réaliser des fonctions! voir plus bas.

Pour nettoyer la cession: 'clear'; si le comportement de scilab est bizare, quiter et relancer scilab!

7.2.1 Fonctions

Les fonctions se mettent dans un fichier et se tapent sous emacs

```
function ydot=f(t,y)
yp(1)=y(3)
yp(2)=y(4)
r=sqrt(y(1)**2 + y(2)**2)
k=1; m=1;
yp(3)=-(k*(m/r)/r)*(y(1)/r)
yp(4)=-(k*(m/r)/r)*(y(2)/r)
ydot=yp
//
function [mu,sig2]=calcmoy(nnb)
xx=rand(nnb,1,'uniform');
mu=mean(xx)
sig2=st_deviation(xx)
```

Pour utiliser une fonction: on va chercher le fichier par:

```
getf('fonc-rand')
```

Noter: pas de end en fin de fonction; différent du fortran.

exercice 12 *Utiliser ces fonctions; faire de petites modifications.*

Réaliser une fonction pour calculer la fonction caractéritique du complémentaire d'un cercle dans un carré; calculer espérance et écart type.

On peut aussi définir des fonctions simples en ligne:

```
deff('[y]=fsol2(x)','y=a*x+b+1.e-3*(x.*x)');
```

Mise au point, utiliser

```
//dans la fonction
pause
// a l'execution
return
//ou
resume
```

7.2.2 Boucles

```
ninterv=22
for i=1:ninterv ,
 l(i)=(i)/ninterv - 1/(2*ninterv)
end
//
while x<14, x=3*x, end
//
 for i=1:ninterv ,
  if ( ((i-1)/ninterv <= xx(n) ) & (xx(n) <= (i)/ninterv) ) then,
   nfreq(i)=nfreq(i) +1;
// noter break
```

```
  break,
 end
end
```

7.2.3 Conditions

```
//noter & logique
 if ( ((i-1)/ninterv <= xx(n) ) & (xx(n) <= (i)/ninterv) ) then,
  nfreq(i)=nfreq(i) +1;
  break,
 end
//
select lecas,
case 1, y=.3,
case 2, y=x^2,
case 3, y=x*sin(x)
end
```

exercice 13 *histogrammes, moyennes, écart-type, loi des grands nombres.*

7.3 Les bibliothèques

En cliquant sur bouton 'help', on a accès à un aide de toutes les fonctions prédefinies; pour des raisons d'efficacité, elles sont le plus souvent codées en fortran! Elles sont souvent issues de la bibliothèque netlib.

Pour obtenir le temps d'exécution:

```
timer(); [m,s]=calcmoy(nnb); timer()
```

exercice 14 *A l'aide de l'aide en ligne, réaliser des calculs: sommes des termes d'un vecteur, racines carrés, sinus; utiliser des boutons de dialogue 'x'; tracer une courbe simple*

7.4 Equations differentielles

Scilab fournit différents algorithmes pour résoudre des équations différentielles. *voir sous-prog sur hivaoa*

exercice 15 *Résoudre la dynamique de la terre avec les algorithmes proposés par scilab; coder aussi la méthode d'Euler; comparer.*

7.5 EDP

On peut considèrer la mise en oeuvre de schémas de différences finie pour l'équation des ondes.

7.5.1 Lax

exercice 16 *Résolution (voir sur math3)*

7.5.2 Modes propres

exercice 17 *(voir sur Hivaoa)*
Résolution avec différences finies pour espace et solution explite après calcul des vecteurs propres.

1.
$$u_{tt} = u_{xx} \quad u_t(0,x) = 0 \quad u(0,x) = u_0(x) \tag{7.1}$$

   discrétiser par différences finies; préciser K.

$$\underline{\ddot{u}} = K\underline{u} \tag{7.2}$$

2. *Diagonaliser K avec Scilab : vecteurs propres v_i*
3. *Soit q^i les composantes de u dans la base de vecteurs propres:*

$$u = \Sigma q^i v_i \tag{7.3}$$

montrer que q^i est solution de:

$$\ddot{q^i} = -\lambda_i q^i \quad \dot{q^i}(0) = 0 \quad q^i(0) = q_0^i \tag{7.4}$$

avec

$$\Sigma q_0^i v_i = \underline{u_0} \tag{7.5}$$

solution explicite des équations différentielles.

4. *La solution approchée:*

$$u = \Sigma q_0^i (cos\sqrt(\lambda_i)t)v_i \tag{7.6}$$

tracer u pour différentes valeurs de t

Fin du chapitre.

Chapitre 8

Utilisation d'une bibliothèque scientifique

Il me parait indispensable tout en apprenant la programmation en f90 et l'analyse numerique d'utiliser une bibliothèque de calcul scientifique; Nous avons choisi la bibliothèque Harwell [?] qui founit des sous-programmes de bonne qualité pour un tarif universitaire abordable; un inconvenient aucun programme principal n'est fourni.

8.1 Sytèmes linéares pleins

Voici par exemple un programme principal en f90 pouur la resolution de systèmes linéaires avec la méthode de Gauss dans la mise en oeuvre de ma21a de [?]:

```
program ppma21
implicit none
common/ma21e/ lp,jscale,ea,eb
integer::lp,jscale
real:: ea,eb
integer, parameter::n=6, ia=n  !n ordre de a; ia 1ere dim de a
real:: e   ! en entree: >0 pour avoir raffinement iteratif;
          ! en sortie  -2: erreur a execution, arret;
          !            -1: idem mais execution terminee;
```

```
          !             0: bon sans rafinement
          !            >=0:bon avec rafinement: estimation de erreur
real, dimension(n,n):: a !matrice du syteme lineaire
real,dimension(n)::b(n)  ! second membre
real,dimension(n*(n+5))::w  !tableau de travail
integer::i
!
interface
      SUBROUTINE MA21A(A,IA,N,B,W,E)
      REAL ZERO,ONE,TWO,P5
      PARAMETER (ZERO=0.0,ONE=1.0,TWO=2.0,P5=0.5)
      REAL E
      INTEGER IA,N
      REAL A(IA,*),B(*),W(*)
      REAL DET,ENORM,ENORMA,EPSN
      INTEGER I,IENT,IP,IT
      EXTERNAL MA21F,MA21I,MA21J
      COMMON /MA21E/LP,JSCALE,EA,EB
      REAL EA,EB
      INTEGER JSCALE,LP

end
endinterface
!
e=1.

a=0.
a(1,1)=1.
do i=2,n-1
a(i-1,i)=-1.
a(i,i)=2.
a(i+1,i)=-1.
enddo
```

```
a(n-1,n)=-1.
a(n,n)=2.
print*,a

b=1.111

call ma21a(a,ia,n,b,w,e)
print*,'solution'
print*,b
print*,'qualite',e
end program ppma21
```

8.2 Systèmes linéaires creux

Dans le cas de systèmes linéaires qui proviennent de calculs par eléments finis, la matrice est creuse, c'est à dire avec beaucoup de zéros; il existe différents modes de stockage: profil, morse, frontal.

Voici l'idée de la méthode frontale qui est une mise en oeuvre particulière de l'algorithme de Gauss: une étape de cet algorithme, consiste à effectuer:

$$A_{mn} \leftarrow A_{mn} - \frac{A_{ml}A_{ln}}{A_{ll}} \tag{8.1}$$

où l est l'indice de la ligne du pivot; dans la méthode de Gauss , on effectue ces calculs dans des boucles après avoir assemblé la matrice A. Dans la méthode frontale on remarque que ce calcul peut être effectué dès que tous les éléments qui doivent contribuer à la formation des coefficients utilisés ont étés assemblés: ces éléments sont appelés frontaux, d'où le nom de la méthode.

La bibliothèque Harwell propose une mise en oeuvre en f90 d'une méthode frontale: MA42. Voici un programme principal pour résoudre un problème très simple:

```
program ppma42
use hsl_ma42_double
```

```
!janvier 1998 rousselet
! un exemple tres simple d'utilisation du module hsl_ma42 de Harwell:
! resoudre  -u''=f par elements finis de degre 1 avec
!u(0)=0 u'(1)=0
!Naturellement dans ce cas il vaudrait mieux utiliser un autre
!algorithme.
! Les impressions sont realisees avec oa02 et oa03 de Harwell
implicit none
type(ma42_data) data
integer deb_elt, ielt, jstop, jstrt, kstrt, nelt, nvar, nz, rhscrd, &
ndf, maxe
integer i,iu
integer, parameter::nc=20,nsf=4  ! parametres d'impressions
real(wp), allocatable:: reals(:,:), rhs(:,:), x(:,:), b(:,:), u(:)
real(wp) :: h
integer, allocatable :: eltnum(:), eltvar(:)
!
call ma42_initialize(data)
!la sructure data est initialise avec des valeurs par defaut
nelt=10                   ! nombre d'elements
allocate(eltnum(nelt))    ! nombre de variables dans chaque element
eltnum=2 !  2 variables par elements ( blocage a gauche penalise)
nz=sum(eltnum)
maxe = maxval(eltnum)
 !ne sert que si le nombre de variables  est non constant
allocate (eltvar(nz))  !liste des variables des elements
                       !(dans l'ordre des elements)
do i = 1, nelt
eltvar(2*i-1 ) = i     !eltvar doit etre un tableau a 1 indice et
!                        contient la
eltvar(2*i) = i+1      !liste des variables des elements
enddo
!
```

```
jstrt=1
!jstrt et
do ielt = 1,nelt       ! boucle sur les elements
 jstop = jstrt + eltnum(ielt) -1
 call ma42_analyze(eltvar(jstrt:jstop), data)
 if (data%error < 0 ) then
  print*,'erreur 0', data%error
  stop
 end if
 jstrt=jstop +1
end do
!
!ma42_analyze permet de calculer le plus grand indice utilise
ndf = data%largest_index
print*, 'ndf=', ndf,' nelt=',nelt
allocate( x(ndf,2))
allocate(u(ndf))
rhscrd = 2*nelt
h=1./real(nelt)   ! le pas de discretisation
! on discretise -u''=f par elements finis de degre 1
!u(0)=0 u'(1)=0; si f=1 on a la solution exacte -(x**2)/2. + x
do iu=1,ndf
u(iu)= -((real(iu-1)*h)**2)/2. + real(iu-1)*h
enddo
allocate( rhs(rhscrd, 2))  ! 2 seconds membres sous forme elementaire
rhs(1:rhscrd,1) = 0.5*h ; rhs(1,1) = 0. !; rhs(rhscrd,1) = 1.*h
rhs(1:rhscrd,2) = 4.*h  ; rhs(1,2) = 0. !; rhs(rhscrd,2) = 8.*h
allocate(reals(1:maxe,1:maxe))  !raideur elementaire
!
!
jstrt=1; kstrt=1
!
! construction de la raideur elementaire
```

```
do ielt = 1, nelt
 nvar = eltnum(ielt)
 jstop = jstrt + eltnum(ielt) -1
 if( (nvar == 1) .and. (ielt == 1) ) then
  ! blocage a gauche
 reals(1,1) = 10.
 elseif ( (nvar == 2) .and. (ielt == 1) ) then
  reals(1,1:2) = (/1.E+8, -1./)/h !blocage penalise
  reals(2,1:2) = (/-1.,1./)/h
 else
  reals(1,1:2) = (/1.,-1./)/h
  reals(2,1:2) = (/-1.,1./)/h
 endif

! factorisation et resolution (rhs et x presents)
 call ma42_factorize(eltvar(jstrt: jstop), reals(1:nvar,1:nvar),&
     data, rhs=rhs(jstrt: jstop,1:2),x=x)
!
 if (data%error < 0 ) then
  print*,'erreur 2', data%error, 'ielt=', ielt
  stop
 end if
 jstrt = jstop + 1
enddo
print*, 'solutions'

call oa02ad(x(:,1),ndf,nc,nsf)
print*, '    '
call oa02ad(x(:,2),ndf,nc,nsf)
call oa03ad(x(:,1),u,ndf,nc,nsf,nsf,'ee',' uexact ')
print*, 'data%det ' ,data%det, ' data%sign_det=',data%sign_det
!
! entree de second membre asemble
```

```
!
allocate(b(ndf,2))
b(1:ndf,1)=1.*h  ;  b(1,1)=0.; b(ndf,1)=0.5*h
b(1:ndf,2) = 8.*h ; b(1,2) = 0. ;b(ndf,2)=4.*h
call ma42_solve(b,x,data)     !resolution avec second membre assemble
 if (data%error < 0 ) then
  print*,'erreur 3', data%error, 'ielt=', ielt
  stop
 end if
!
 call ma42_final(data)

print*, 'solution 2'
call oa02ad(x(:,1),ndf,nc,nsf)

 print*, '    '
call oa02ad(x(:,2),ndf,nc,nsf)
 print*, 'fin analyse'
 end program ppma42
```

exercice 18 *En s'inspirant de pp3ma42, réaliser une mise en oeuvre d'un programme bidimensionnel d'éléments finis; par exemple prendre celui du chapitre 4.*

8.3 Programmation quadratique

Il s'agit du problème présenté dans le cours d'optimisation au§2.2 : $v \in I\!R^n \quad J(v) = \frac{1}{2}{}^t v\, A\, v \; - \; {}^t v\, f$ avec la contrainte

${}^t B\, v \; - \; c = 0$ où $\underset{m}{\boxed{\mathrm{B}}}\; n$ est une matrice à n lignes et m colonnes avec $m < n$.

La bibliothèque Harwell propose une mise en oeuvre d'un algorithme pour résoudrer ce type de problème$>$

Chapitre 9

Utilisation de Maple.

9.1 Calcul d'une matrice de rigidité élémentaire

En suivant les notations de **??** voici un petit programme maple que l'on peut réaliser pour faire un programme fortran de la matrice de rigidité élémentaire avec des polynomes de degré 1; naturellement dans ce cas très simple maple est superflu; il devient rès utile pour des éléments plus compliqués. Pour l'aide au maintien des sou-programmes, je conseille d'insérer le souce maple en commentaires dans le sous-programme fortran.

```
module matrices
contains
function rigid(x,y,k)
implicit none
integer, parameter::nel=3
double precision, dimension(nel,nel):: rigid,r
double precision x(nel), y(nel), b(nel,2),det, k(2,2)
!---------------------------------
! le determinant de la matrice A et la matrice B
!(voir notes de cours d'elements finis, chapitre 1) sont calcules
! ici avec l'aide de maple; voici le programme maple utilise:
!with(linalg):
```

```
!#avec x,y tabeaux
!x:=vector(3); y:=vector(3);
!a:=matrix( [ [1., x[1], y[1] ],[1., x[2], y[2]], [1., x[3], y[3] ] ]);
!ainv:= inverse(a);
!d:=det(a);
!# on multilplie par le det(A), pour simplifier presentation
!ainvd:=scalarmul(ainv,d);
!#on reordonne par rapport a x et y
!ainvdn:=map(normal,ainvd);

!phi:=matrix( [[1., xx, yy] ]);
!# H contient les 3 fonctions de base
!ht:=multiply(phi,ainvdn);
! hc:=map(collect,ht,[xx,yy]);

!b1:= map(diff,ht,xx); b2:=map(diff,ht,yy);
!bt:=stack(b1,b2);
!b:=transpose(bt);
!# bd:=evalm(d * b);
!# attention dans le fortran, il faudra diviser par det(A)
!fortran([det=d],filename=matelem.f90);
!fortran(b,filename=matelem.f90);

!-----------------------------------
    det = 1.E0*x(2)*y(3)-y(2)*x(3)-x(1)*y(3)+1.E0*y(1)*x(3)+1.E0*x(1)*y(2)-y(1)*x(2)
      b(2,1) = 1.E0*y(3)-y(1)
      b(2,2) = -x(3)+1.E0*x(1)
      b(3,1) = -y(2)+1.E0*y(1)
      b(1,1) = -y(3)+1.E0*y(2)
      b(3,2) = 1.E0*x(2)-x(1)
      b(1,2) = 1.E0*x(3)-x(2)
b=b/det
r=matmul(b,matmul(k,transpose(b)))
```

```
!r=matmul(b,transpose(b))
rigid = r*abs(det)/2.
end function rigid
end module matrices

program elem
use matrices
implicit none
integer, parameter::nel=3

double precision x(nel), y(nel),h,k(2,2)
double precision, dimension(nel,nel):: rel
x=0.; y=0.
h=0.1
x(2)=h
y(3)=h
k=0.; k(1,1)=1.;k(2,2)=1.
rel=rigid(x,y,k)
print*, rel
end program elem
```

exercice 19 *Intégrer le module matrices dans le programme d'éléments finis décrit à la section* **??**

exercice 20 *En utilisant maple, et en voyant [*?*], réaliser un sous-programme f90 qui calcule les matrices élémentaires pour un élément*

1. *triangulaire à 6 noeuds;*
2. *quadrangulaire à 4 noeuds en utilisant un carré de référence.*

Chapitre 10

Euler implicite

10.1 Introduction

L'objet de ce chapitre est d'étudier une mise en oeuvre de l'algorithme d'Euler implicite (ou tangente améliorée) pour la résolution numérique de systèmes d'équations différentielles ordinaires. L'objectif est de résoudre un système classique: la dynamique du point matériel dans un champs de forces centrales, et pour fixer les idées on va essayer de calculer une portion de la trajectoire $t \longrightarrow \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ de la Terre autour du Soleil.

Les algorithmes sont donc:

- Euler implicite.
- Algorithme de Newton- Raphson.

L'algorithme de Newton sert à résoudre à chaque pas de temps, le système non linéaire fourni pas l'algorithme d'Euler implicite.
Les outils logiciels sont:

- Fortran 90 pour le programme lui même (on rapellera au passage quelques particularités du langage fortran 90).
- Le logiciel de calcul symbolique Maple, pour engender les équations approchées du schéma d'Euler et leur dérivée, utiles pour l'algorithme de Newton- Raphson.
- Adaptation pour un système, d'un programme en fortran 90 pour l'algorithme de Newton proposé par Dodson [?] pour le cas scalaire.

10.1.1 Le schéma d'Euler Implicite

Pour approcher dans $[t_0, T]$ la solution de:

$$\begin{cases} \dfrac{d\vec{x}}{dt} = \vec{f}(t, x) \\ \\ \vec{x}(t_0) = x_0 \end{cases} \tag{10.1}$$

On peut découper (réf. [?], [?]) l'intervalle $[t_0, T]$ en intervalles $[t_n, t_{n+1}]$ et dans ces intervalles on approche (1) par le schéma:

$$\begin{cases} \dfrac{\vec{x}_{n+1} - \vec{x}_n}{\tau} = \vec{f}(t_{n+1}, \vec{x}_{n+1}) \\ \\ avec\ x_0\ condition\ initiale\ de\ (1) \end{cases} \tag{10.2}$$

Ce schéma est dit implicite car (2) est une équation implicite pour l'inconnue x_{n+1} . L'origine géométrique de l'algorithme est très simple: le second membre est le vecteur vitesse de la solution de (1) passant par le point inconnu; le premier membre est le taux de variation du point de la trajectoire passant de t_n à $t_n + \tau$

Dessin.

Par rapport au schéma d'Euler explicite:

$$\frac{\vec{x}_{n+1} - \vec{x}_n}{\tau} = \vec{f}(t_n, \vec{x}_n) \tag{10.3}$$

cet algorithme présente de meilleurs propriétés de stabilité. Toutefois il reste rudimentaire et de meilleurs algorithmes peuvent être utilisés (Runge- Kutta, algorithme de Gear [?]: où on pourra par exemple utiliser dans Harwell Subroutine Library [?], les subroutines DA01, DA02, DC03).

Pour un système de second ordre

$$\frac{d^2\vec{x}}{dt^2} = \vec{f}(t, \vec{x}) \tag{10.4}$$

(comme celui de la dynamique du point matériel), il est habituel de transformer en système du premier ordre:

$$\begin{cases} \dfrac{d\vec{v}}{dt} = \vec{f}(t, \vec{x}) \\ \\ \dfrac{d\vec{x}}{dt} = \vec{v} \end{cases} \tag{10.5}$$

Le schéma d'Euler implicite est alors:

$$\begin{cases} \dfrac{\vec{v}_{n+1} - \vec{v}_n}{\tau} = \vec{f}(t_{n+1}, \vec{x}_{n+1}) \\ \\ \dfrac{\vec{x}_{n+1} - \vec{x}_n}{\tau} = \vec{v}_{n+1} \end{cases} \tag{10.6}$$

On peut alors l'appliquer sous la forme:

$$\frac{\vec{X}_{n+1} - \vec{X}_n}{\tau} = \vec{\mathcal{F}}(t_{n+1}, \vec{X}_{n+1}) \tag{10.7}$$

où $X_n = \begin{pmatrix} v_n \\ x_n \end{pmatrix}$, et $\vec{\mathcal{F}} = \begin{pmatrix} f(t_{n+1}, \vec{x}_{n+1}) \\ v \end{pmatrix}$.

Nous pouvons avoir une troisième formulation en reportant la deuxième équation de (.) écrite au temps t_{n+1}, au temps t_n:

$$v_{n+1} = \frac{x_{n+1} - x_n}{\tau} \quad v_n = \frac{x_n - x_{n-1}}{\tau}, \tag{10.8}$$

ce qui donne:

$$\frac{x_{n-1} - 2x_n + x_{n+1}}{\tau^2} = f(t_{n+1}, x_{n+1}). \tag{10.9}$$

Ce dernier schéma est à deux pas (x_{n-1} et x_n sont necéssaires pour calculer x_{n+1}), et implicite; il peut être obtenu directement à partir de (4) en approchant $\frac{d^2x}{dt^2}$ par le schéma à trois points:

$$\frac{x(t-\tau)-2x(t)+x(t+\tau)}{\tau^2}.$$

10.1.2 Les équations à résoudre.

Un exemple simple.

Pour la mise au point du programme on pourra utiliser le système:

$$\begin{cases} \dfrac{d\vec{x}}{dt} = A\vec{x} \\ \\ \vec{x}(t_0) = \vec{x}_0 \end{cases} \tag{10.10}$$

où A est une matrice de petite taille, puis le systeme "non- linéaire ":

$$\begin{cases} \dfrac{d\vec{x}}{dt} = A\vec{x} + \varepsilon\vec{\varphi}(\vec{x}) \\ \\ \vec{x}(t_0) = \vec{x}_0 \end{cases} . \tag{10.11}$$

Ici $\varphi(x)$ est la partie non -linéaire. Pour les calculs, on peut choisir $\varphi(x) = x^3$ avec $x^3 = (x_1^3, x_2^3, ..., x_N^3)$.

La dynamique du point matériel dans un champs de forces centrales.

Selon Newton (ref. Berkeley [?], Le Bellac[?]), le soleil exerce sur la Terre une force d'attraction:

$$\vec{F} = -\frac{Km_T m_S}{r^2}.\frac{\vec{x}_T - \vec{x}_S}{r} \tag{10.12}$$

où le Soleil est situé en $\vec{x}_S$, et la Terre en $\vec{x}_T$; r est la distance de la Terre au Soleil:

$$r = \|\vec{x}_T - \vec{x}_S\|_2$$

m_T = masse de la Terre,
m_S = masse du Soleil.

En supposant en première approximation que le soleil est fixe, posons $\vec{x} = \vec{x}_T - \vec{x}_S$. L'équation de la dynamique de la Terre s'écrit:

$$m_T \frac{d^2x}{dt^2} = -K\frac{m_T m_S}{r^2}.\frac{\vec{x}}{r} \tag{10.13}$$

soit:

$$\frac{d^2x}{dt^2} = -K\frac{m_S}{r^3}\vec{x}. \tag{10.14}$$

Nous sommes donc dans le cas (4) avec $f(t,\vec{x}) = -K\frac{m_S}{r^3}\vec{x}$; f ne dépend pas de t; le système est dit autonome.

Quelques propriètés classiques utiles pour la résolution numérique.

<u>**Energie Totale.**</u>

$$E = \frac{1}{2}m_T(\frac{dx}{dt})^2 - \frac{Km_T m_S}{r} \tag{10.15}$$

Comme $\frac{\partial r}{\partial x}.\delta x = \frac{x.\delta x}{r}$, on vérifie facilement que

$$\frac{dE}{dt} = \left(m_T\frac{d^2x}{dt^2} + K\frac{m_T m_S}{r^2}\frac{\vec{x}}{r}\right)\frac{dx}{dt}.$$

D'où comme x est solution de (13), $\frac{dE}{dt} = 0$ c'est à dire que $E = c$ est constante sur une trajectoire.

Moment Cinétique. Le moment cinétique est défini par le produit vectoriel $\mathcal{J} = \vec{x} \wedge m_T \vec{v}$. En effectuant le produit vectoriel de (13) par x, il vient:

$$m_T \frac{d^2 x}{dt^2} \wedge x = 0$$

et donc ,

$$\frac{d}{dt}\left(m_T \frac{dx}{dt} \wedge x \right) = 0$$

d'où $\mathcal{J} = \vec{v} \wedge m_T \vec{x}$ est un vecteur constant. En fait, son module a une valeur connue:

$$\|\vec{\mathcal{J}}\| =$$

Cela permet de déduire la vitesse à partir de la position; en pratique cela est très utile pour poser les conditions initiales en vitesse.

10.2 L'algorithme de Newton- raphson

10.2.1 Présentation.

Pour résoudre un système d'équations non- linéaires (voir P. G. Ciarlet [?]), on utilise souvent la méthode de Newton- Raphson. Elle est très efficace et d'ordre deux, comparée à d'autres méthodes qui résolvent les mêmes problèmes:

$$\vec{\mathcal{F}}(\vec{X}) = 0. \tag{10.16}$$

L'idée est très simple: si X_k est une approximation d'une solution de (16) on cherche $X_{k+1} = X_k + \delta X_k$, solution de

$$\vec{\mathcal{F}}(X_k + \delta X_k) = 0. \tag{10.17}$$

Cette equation est aussi difficile à résoudre que (16), mais on peut l'approcher par une l'équation linéaire:

$$\vec{\mathcal{F}}'(X_k)\delta X_k = -\vec{\mathcal{F}}(X_k). \tag{10.18}$$

10.2.2 Utilisation pour résoudre l'équation fournie par l'algorithme d'Euler implicite

Pour résoudre l'équation implicite (2) on écrit:

$$\mathcal{F}(x) = \frac{x - x_n}{\tau} - f(t, x) \tag{10.19}$$

et donc

$$\mathcal{F}'(x)\ \delta x\ =\ \frac{\delta x}{\tau} - \frac{\partial f}{\partial x}\delta x$$

et donc (18) s'écrit:

$$\frac{\delta x}{\tau} - \frac{\partial f}{\partial x}\,\delta x = -\frac{x - x_n}{\tau} + f(t, x). \tag{10.20}$$

Pour l'équation (9) on a:

$$\mathcal{F}(x) = \frac{x_{n-1} - 2x_n + x_{n+1}}{\tau^2} - f(t_{n+1}, x) \tag{10.21}$$

qui implique:

$$\mathcal{F}'(x)\delta x = \frac{\delta x}{\tau^2} - \frac{\partial f}{\partial x}\delta x,$$

et (18) s'écrit:

$$\frac{\delta x}{\tau^2} - \frac{\partial f}{\partial x}\,\delta x\ =\ -\frac{x_{n-1} - 2x_n + x_{n+1}}{\tau^2} + f(t_{n+1}, x). \tag{10.22}$$

Pour l'équation (7), et en posant $X = \begin{pmatrix} v \\ x \end{pmatrix}$, on a:

$$\mathcal{F}(X)\ =\ \begin{bmatrix} \dfrac{v - v_n}{\tau} & - f(t_{n+1}, x) \\ & \\ \dfrac{x - x_n}{\tau} & - v \end{bmatrix}, \tag{10.23}$$

$$\frac{\partial \mathcal{F}}{\partial X}\ \delta X\ =\ \begin{bmatrix} \dfrac{\delta v}{\tau} & -\dfrac{\partial f(t, x)}{\partial x}\delta x \\ & \\ -\delta v & \dfrac{\delta x}{\tau} \end{bmatrix}$$

et (18) s'écrit:

$$\left\{\begin{array}{l} \dfrac{\delta v}{\tau} - \dfrac{\partial f(t,x)}{\partial x}\delta x = \dfrac{v - v_n}{\tau} + f(t_{n+1}, x) \\ \\ -\delta v + \dfrac{\delta x}{\tau} = -\dfrac{x - x_n}{\tau} + v \end{array}\right. \tag{10.24}$$

Ce système linéaire 4×4 peut être résolu directement par un algorithme de type Gauss; la matrice $\begin{bmatrix} I/\tau & -\frac{\partial f(t,x)}{\partial x} \\ -I & I/\tau \end{bmatrix}$ peut aussi être inversée à l'aide de Maple; toutefois elle se présente sous la forme de blocs 2×2 et il est plus efficace de tirer δv à partir de δx de la 2^{ieme} équation (voir §3.2).

Remarque

Comme $v_n = \dfrac{x_n - x_{n-1}}{\tau}$ (voir (8)), l'équation (25) est identique à (22).

Un détail très important pour le bon déroulement de l'algorithme de Newton est de disposer d'une approximation convenable de la solution; dans le cas contraire il est bien connu que l'algorithme peut diverger.

Toutefois, quand on utilise l'algorithme de Newton pour résoudre l'équation d'Euler implicite, on dispose d'un bon point de départ: c'est la solution x_n au pas de temps précédent, ou la solution d'Euler explicite (voir §3.2).

10.3 Mise en oeuvre des schémas d'Euler explicite et implicite

10.3.1 Euler explicite

L'algorithme est banal, on utilise deux tableaux x_0 et x_1 pour représenter 2 itérés succéssifs. Les tableaux x^0 et x^1 sont déclarés à la dimension du système différentiel considéré et il suffit de réaliser la condition initiale (t_0, x^0), de donner un pas de temps τ, et le nombre de pas de temps $nmax$:

$$\left|\begin{array}{l} faire\ pour\ i=1,...,nmax \\ x^1 = x^0 + \tau f(t_0, x^0) \\ ecrire\ \ x^1\ \ dans\ \ un\ \ fichier \\ x^0 = x^1 \\ fin,\ \ faire. \end{array}\right. \tag{10.25}$$

Il reste à fournir la fonction $f(t_0, x^0)$ qui est le second membre de l'équation différentielle; dans le cas de la dynamique du point matériel:

$$\vec{f} = -K\,\frac{m_S}{r^3}\,\vec{x} \qquad avec \quad r^2 = (x_1)^2 + (x_2)^2 = \|x\|^2. \tag{10.26}$$

mais comme il s'agit d'une équation du 2^{nd} ordre, il convient de le transformer en système du premier ordre:

10.3.2 Euler implicite

Comme dans le cas explicite, on utlilise les tableaux x^0 et x^1, les données initiales (t_0, x^0), τ pas de temps, et $nmax$ le nombre maximum de pas de temps, suivant l'algorithme:

$$\left|\begin{array}{l} faire\ \ pour\ \ n\ de\ \ 1\ \ a\ \ nmax: \\ x^1 - x^0 - \tau f(t^1, x^1) = 0 \\ ecrire\ n\tau\ et\ \ x^1\ \ dans\ \ \ un\ \ fichier \\ x^0 = x^1 \\ fin,\ \ faire. \end{array}\right. \tag{10.27}$$

L'équation pourra être résolue à l'aide de l'algorithme de Newton du §3.

Avec $\mathcal{F}(X) = X - X^0 - \tau f(t^1, X)$, τ, X^0, et t^1 étant des paramètres; comme point initial pour l'algorithme de Newton on pourra prendre:

$$X = X^0 \quad dans\ \ ce\ \ cas \qquad \mathcal{F}(X^0) = -\tau f(t^1, X^0) \tag{10.28}$$

ou mieux $X = X^1_{explicite}$ fournit par l'algorithme d'Euler explicite:

$$X = X^1_{explicite} = X^0 + \tau f(t^0, X^0), \tag{10.29}$$

il vient alors:

$$\mathcal{F}(X^1_{explicite}) = \tau f(t^0, X^0) - \tau f(t^1, X^0 + \tau f(t^0, X^0)) = \tau^2 \left(\frac{\partial f}{\partial t^0} + \frac{\partial f}{\partial X^0} f^0 \right),$$

qui est donc en $O(\tau^2)$ et fournit en principe un bon point de départ.
Ce dernier choix est un cas particulier de la famille de méthodes: prédiction- correction; prediction par une méthode explicite, et correction par une méthode itérative (par exemple Newton), pour satisfaire une implicite.

Dans le cas d'un système de second ordre, l'algorithme s'écrit en suivant (6)

$$\left\{ \begin{array}{l} v^1 = v^0 + \tau f(t^1, X^1) \\ \\ X^1 = X^0 + \tau v^1 \end{array} \right. \tag{10.30}$$

La fonction à fournir à Newton est suivant (23):

$$J(X) = \begin{pmatrix} \dfrac{v^1 - v^0}{\tau} & - f(t^1, X^1) \\ \\ \dfrac{X^1 - X^0}{\tau} & - v^1 \end{pmatrix} \tag{10.31}$$

avec $\begin{pmatrix} v \\ x \end{pmatrix}$ et $\frac{\partial \mathcal{F}}{\partial X} \delta X = \begin{bmatrix} \partial v / \tau & - \frac{\partial f^1}{\partial x} \delta x \\ -\delta v & \delta x / \tau \end{bmatrix}$.

Cette matrice 4×4 peut être inversée simplement. On trouve que le système

$$\left\{ \begin{array}{l} \dfrac{\delta v}{\tau} - \dfrac{\partial f^1}{\partial x} \delta x = a \\ \\ -\delta v + \dfrac{\delta x}{\tau} = b \end{array} \right. \tag{10.32}$$

se résout en

$$\left\{ \begin{array}{l} \dfrac{\partial f^1}{\partial x} \delta x = -a + \dfrac{\delta v}{\tau} \\ \\ \delta v = \dfrac{\delta x}{\tau} - b \end{array} \right. \tag{10.33}$$

puis:

$$\left\{\begin{array}{l}\left(\dfrac{\partial f^1}{\partial x}-\dfrac{1}{\tau^2}\right)\delta x\;=\;-a\;-\;\dfrac{b}{\tau}\\[4ex]\delta v\;=\;\dfrac{\delta x}{\tau}\;-\;b\end{array}\right. \tag{10.34}$$

et le système de Newton (18) se résout donc ici à l'aide de (35) avec:

$$\left\{\begin{array}{l}a\;=\;\dfrac{v^1-v^0}{\tau}-f(t^1,X^1)\\[4ex]b\;=\;\dfrac{X^1-X^0}{\tau}-v^1\end{array}\right. \tag{10.35}$$

<u>Réalisation effective</u>: Le seul système à résoudre est donc la première équation de (35) avec une matrice 2×2; on utilisera donc une solution explicite.

<u>Mise en œuvre</u>: Bien que ce ne soit pas indispensable dans ce cas; nous allons à titre pédagogique utiliser Maple pour les instructions fortran pour le calcul de $\mathcal{F}$ et de son inverse.

Chapitre 11

Divers en fortran 77 et 90

11.1 Pointeurs

Les pointeurs sont des variables munies de l'attribut POINTER mais pas un type particulier.
Ils décrivent les attributs des objets; les pointeurs ne peuvent pointer que vers des variables ciblées avec attribut TARGET ou vers d'autres pointeurs.
Voici un exemple où l'on utilise des pointeurs comme alias pour effectuer le produit matriciel p = b c z.

```
program produit
  real, target :: b(10,10), c(10,10), z(10), r(10), p(10)
  real, pointer :: a(:,:), x(:), y(:)
  y ⇒ r    ! on associe le pointeur y à la cible r
  a ⇒ c
  x ⇒ z
y = matmul(a,x)    ! effectue cz et le met dans r
  y ⇒ p
  a ⇒ b
  x ⇒ r
y = matmul(a,x)    ! effectue br et le met dans p qui contient donc bcz
end program produit
```

Mais on peut aussi utiliser des pointeurs pour créer des listes chainées ; ceci se réalise à l'aide d'une récursivité dans la déclaration d'un type:
voici la syntaxe pour la déclaration d'un type qui permet de construire une liste chaînée.

```
type noeud
  real donnée
  type(noeud), pointer :: suivant
end type noeud
```

On trouvera un exemple d'utilisation au §2.8. de Dodson.

11.2 Les variantes des types de base

Les types de base (en particulier reel) possèdent des caractéristiques qui dépendent de l'ordinateur considéré. C'est ainsi que le type simple précision de Cray correspond à un nombre de chiffres significatifs qui correspond à la double précision de la plupart des autres machines.
Pour assurer la portabilité f90 définit des numéros de variante: le paramètre kind; malheureusement le numéro associé à des caractéristiques données dépend de la machine: cela se trouve dans la documentation du compilateur mais peut être aussi fourni par des fonctions intrinsèques:

```
select_real_kind(7,20)
```

fournit le numéro de la variante de nombres réels avec au moins 7 chiffres après la virgule et un exposant inférieur à 20.
Cette valeur peut être fournie et modifiée facilement dans une bibliothèque de sous-programmes à l'aide d'un module:

```
module Precision
  integer, parameter :: pre = select_real_kind(7,20)
end module Precision
```

pour utiliser une variable x avec cette variante on déclare:

```
real(pre) :: x, a=2.76_pre
```

Enfin pour une constante ou variable donnée, kind fournit son numéro de variante: kind(1.do) fournit le numéro de variante de la double précision.

11.3 Procédures intrinsèques de f90

Elles sont très nombreuses: consulter Delanoy (1993) [?] ou Metcalf -Reid (1992) [?] pour listes complètes.

• Voici quelques fonctions numériques:

aint(x,kind) : *troncature de x (partie entière) réel de variante kind; résultat réel;*

anint(x,kind) : *arrondi à l'entier le plus proche;résultat réel;*

int(x,kind) : *troncature de x, mais cela donne ici un entier;*

nint(x,kind) : *entier le plus proche de x;*

ceiling(x) : *entier immédiatement supérieur à x*

floor(x) : *entier immédiatement inférieur à x*

real(x,kind) : *réel de variante kind associé à x*

(si x complexe on obtient la partie réelle);

max(a_1, ..., a_n) : *maximum des* a_i

min(a_1, ..., a_n) : *minimum des* a_i

• fonctions mathématiques.

Acos(x) *donne Arc cos(x)* $\in [0, \pi]$ *avec* $x \in [-1, 1]$

Asin(x) *donne Arc sin(x)* $\in [-\frac{\pi}{2}, \frac{\pi}{2}]$ *avec* $x \in [-1, 1]$

Atan(x) *donne Arc tg(x)* $\in [-\frac{\pi}{2}, \frac{\pi}{2}]$ *avec* $x \in I\!R$

Atanr(x,y) *donne Argument(x+iy)* $\in [-\pi, \pi]$ *avec* $x \in I\!R$, $y \in I\!R$

cos(x) *donne cos x* $x \in I\!R$ ou C

cosh(x) *donne ch x* $x \in I\!R$

exp(x) *donne* e^x $x \in I\!R$ ou C

log(x) *donne ln x* : logarithme néperien pour $x \in I\!R$

(ou $x \in C_*$ avec Imln $x \in [-\pi, \pi]$)

log10(x) *donne log(x) : logarithme décimal de* $x \in I\!R_+$

sin(x) *donne sin x ,* $x \in I\!R$ *ou C*

sinh(x) *donne sh x ,* $x \in I\!R$

sqrt(x) *donne* $\sqrt{x}$, $x \in I\!R_+$ *(ou* $x \in C$*, alors* $Im\sqrt{x} \geq 0$*)*

tan(x) *donne tg x ,* $x \in I\!R$

tanh(x) *donne th x ,* $x \in I\!R$

Attention : ne plus utiliser les anciens noms qui dépendent du type comme dcos(x) quand x est en double précision: en fortran 90 c'est inutile et périmé (voir §4.4.).

• fonctions liées au type des nombres:

digits(x) : *nombre de chiffres significatifs;*
epsilon(x) : *plus grand nombre* ε *tel que en machine* $1+\varepsilon=1$*;*
huge(x) : *plus grande valeur représentable dans le type de x;*
maxexponent(x) : *plus grande valeur possible pour un exposant;*
minexponent(x) : *plus petite valeur possible pour un exposant;*
tiny(x) : *plus petite valeur représentable dans le type de x;*

• Pour les chaines de caractères, nombreuses fonctions; citons seulement

achar(i) : *avec i entier, fournit le caractère correspondant au code ascii de i;*
iachar(c) : *fournit l'entier correspondant au code* ascii *du caractère c.*

• Manipulation de bits;
citons seulement:

bit_size(i) : nombre de bits dans la représentation des entiers du type de i.

• Pour les tableaux;

dot_product(x,y) : *produit scalaire des 2 vecteurs x et y*
matmul(a,b) : *produit matriciel de matrices rectangulaires*
(de dimensions compatibles).
minval(tab) : *plus petite valeur du tableau tab;*
maxval(tab) : *plus grande ————————–;*
sum(tab) : *somme des éléments du tableau tab;*
product(tab) : *produit des éléments du tableau;*
transpose(tab) : *tranposition du tableau tab de rang 2;*
maxloc(tab,masque) : *indices relatifs à la plus grande valeur des éléments de tab;*

masque est un tableau logique (optionnel) de même profil que tab: la recherche se fait seulement dans les éléments de tab associés à des valeurs vraies des éléments analogues de masque.

minloc(tab,masque) : comme maxloc mais avecla plus petite valeur.

• Divers

call date_and_time(date, time, zone, values)

fournit dans les arguments de sortie:

date ='19970315' soit le 15 mars 1997 ;

time ='183427' soit 18h 34mn et 27 secondes;

zone ='+0100' soit 1 heure et zéro minutes de plus que le temps universel;

values : tableau de 8 entiers qui contient les valeurs précédentes sous-forme numérique

call_random_number(x) : fournit un nombre aléatoire dans]0,1[; x peut être un tableau.

11.4 Instructions désuètes et périmées

Les instructions périmées: doivent être signalées par le compilateur; peuvent devenir interdites dans la prochaine version de fortran; il s'agit des instructions suivantes avec en parenthèse la méthode f90.

Equivalence	(Utiliser des pointeurs)
Common	(Utiliser un module)
data	(initialiser en déclarant)
block data	(——————————)
go to calculé	(utiliser case)

anciens noms de fonctions intrinsèques qui dépendent du type comme dsqrt pour la racine carrée en double précision.

la directive: include (utiliser un module)

Les instructions désuètes : emploi déconseillé; peuvent devenir périmées dans la prochaine version

if arithmétique

assign
go to assigné
namelist

11.5 Exemple de mise aux normes f90

Nous considèrons un petit programme d'éléments finis écrit en f77 par F. Bay (CEMEF, Ecole des Mines). Ce programme peut être utilisé tel quel en f90. Nous proposons quelques adaptations: en particulier suppressions d'instructions périmées (voir §4.4.) .
Par commodité, nous le transformons en mode libre avec le programme domaine public CONVERT. Si l'on n'a pas la pratique du f77, il est fortement conseillé de faire cette conversion.
1) Nous remplaçons l'instruction include par 'par_max' présente dans le programme principal et des sous-programmes par un module:

```
 module par\_max
 integer, parameter :: nbnoemax = 500
integer,parameter:: nbeltmax= 100
integer,parameter:: nel =3
integer,parameter :: nbitermax= 2000
 end module par\_max
```

Au passage, nous avons utilisé la nouvelle syntaxe pour les constantes symboliques, mais cela n'était pas indispensable.
Nous mettons use 'par_max' à la place de include 'par_max'

2) Nous remplaçons le common/discrétisation par un module ceci pourrait être mis dans le même module que par_max; nous en faisons un autre; noter que les variables sont également déclarées dans le module.

```
module discretisation
  double precision, save :: hdiser
  integer, save :: nbnoe, nbelt, nbn
```

```
end module discretisation
```

Nous avons mis l'attribut $\boxed{\text{save}}$ pour que les valeurs des variables soient bien transmises d'une unité de programme à l'autre; <u>attention</u> en l'abscence de SAVE: pas d'erreur de compilation mais on n'est pas sûr que les valeurs soient passées (dépend de la machine).
Au passage, nous avons remplacé Real* 8 par double precision; *real* 8* est une <u>extension</u> courante en f77 mais n'a pas été reprise par f90 car il a mis en place la notion de variante de type, voir §4.2. ; *double precision* est une instruction standard de f77.

3) mettre implicit none dans le programme principal et sous-programmes et déclarer les variables qui ne l'étaient pas.
Un point important est de réaliser des interfaces explicites (cf §3.2.) ; cela peut être réalisé de 3 façons:

- des blocs d'interfaces
- des sous-programmes internes
- des modules.

Nous proposons de réaliser 3 modules:un pour le maillage, un pour assemblage et conditions aux limites, un pour la resolution; notons que les modules ont besoin des parametres situes dans par_max; mettre use parmax dans les modules:

```
module maillage
  contains
  subroutine maillage(nbn, hdiser, xglob, none, nbnoe, nbelt)
  .
  .
  .
  .
  end subroutine maillage
end module maillage
```

Le sous-programme assemblage ne peut contenir matloc dans lequel f serait une fonction interne (2 niveaux de sous-programme internes sont exclus par f90) matloc; nous mettons donc assemblage, matloc et impose_di au même niveau dans un module et f fonction interne de matloc.

```
  module assemblage_et_conditions_limites
   use discretisation; use par_max
   contains
   subroutine assemblage(none, xglob, a, b)
    .
    .
    .
    .
   end subroutine assemblage
    subroutine matloc(none, xglob, a, b)
    .
    .
    .
    ! après les instructions exécutables
    contains
function f(x,y)
real*8 x, y,f  ! f rajoute
f = 1.
end function f
    end subroutine matloc
   subroutine impose_di(a, b, di, val_imposée)
    .
    .
   end subroutine impose_di
  end module assemblage_et_conditions_limites


  module résolution
   use par_max; use discretisation
   contains
```

```
 subroutine gauss_seidel(a, b, x, y)
  .
  .
  .
 end subroutine gauss_seidel
end module résolution .
```

Index

Bibliographie

[1] *Delanoy C.*, Programmer en fortran 90; guide complet, Eyrolles, 1993.

[2] *Dodson Z.*, A Fortran 90 Tutorial, Computer Science Department, University of New Mexico. June 20, 1994.

[3] *Cl. Gomez, B. Salvy P. Zimmermann*, Calcul formel: Mode d'emploi; exemples en Maple, Masson, 1995.

[4] *Cl. Gomez* Scilab, 1999.

[5] *AEA Technology, Harwell Laboratory, Oxfordshire, England* Harwell subroutine library, release 12

[6] *Metcalf M. & Reid J.*, Fortran 90 explained, Oxford University Press, 1992.

[7] B. Rousselet, Rudiments de calcul par éléments finis, Notes de cours, 1998.