ISIMA Première Année

COURS DE PROGRAMMATION NUMÉRIQUE

Introduction au FORTRAN 90

Jonas KOKO

Institut Supérieur d'Informatique, de Modélisation et de leurs Applications Campus des Cézeaux
B.P. 1025 - 63173 AUBIERE CEDEX
http://www.isima.fr

Table des matières

Introduction

Le Fortran est LE langage du Calcul Scientifique. Le mot Fortran signifiait, à l'origine, *IBM Mathematical FORmula TRANslation System*, puis a été abrégé en *FORmula TRANslation*. C'est le plus vieux langage de programmation de haut niveau puisqu'il date de 1954. Les sous–programmes (procédures et fonctions) sont apparus en 1958 avec la deuxième version du langage.

Dans les années 60, le langage est devenu tellement populaire que d'autres constructeurs ont commencé à produire leur propre compilateur. Ce qui a conduit à une multiplication de dialectes Fortran. Il a été alors reconnu que cette divergence croissante n'était dans l'intérêt ni des programmeurs, ni des constructeurs. C'est ainsi que Fortran 66 est devenu en ... 1972 le premier langage à être officiellement standardisé. Depuis c'est une tradition Fortran d'adopter comme numéro de version l'année de la première mouture de la norme (qui n'est pas celle de l'officialisation de la norme). En 1980, c'est le Fortran 77 qui a été adopté par l'ISO comme standard international.

Il a fallu ensuite attendre 1991–1992 pour que le Fortran 90 soit officiellement adopté, après des années de conciliabules à cause du sort à réserver au Fortran 77, pour ne pas perdre l'énorme quantité de bibliothèques scientifiques existantes (fiabilisées par de nombreuses années d'utilisation). C'est pour cela que Fortran 90 «comprend» Fortran 77. A noter que les premiers compilateurs Fortran 90 ne sont apparus qu'en 1994. La norme la plus récente est le Fortran 95 dont les plus vieux compilateurs date de 1999.

Les apports de Fortran 90 sont considérables. Voici une liste de quelques nouveautés apportées par le Fortran 90.

- écriture du programme en «format libre», qui remplace avantageusement l'ancien «format fixe» de Fortran 77, hérité des cartes perforées
- objet de types dérivés
- plus de structures de contrôle pour éviter les `goto` «sauvages»
- expressions tableaux
- procédures et fonctions internes (ça n'existe pas en Fortran 77 ! !)
- interfaces, modules pour fiabiliser la communication entre unités de programme
- vocation des arguments, arguments optionnels, passage par mot clé
- nouvelles fonctions intrinsèques

Nous n'avons pas l'intention de faire une présentation complète de toutes les possibilités offertes par le Fortran 90. Nous ne présenterons que les aspects les plus adaptés pour les TP de Programmation Numérique. Bien que Fortran 90 comprenne Fortran 77, nous ne parlerons pas des antiquités du Fortran 77 telles que : *étiquettes, goto, common, data,...* car devenues complètement obsolètes.

Chapitre 1

Généralités, types scalaires

1.1 Généralités

Tout le jeu de caractères usuels est permis en Fortran 90, sauf qu'il n'y a pas de différence entre minuscules et majuscules. En «format libre» (le format par défaut) les lignes peuvent être de longueur quelconque à concurrence de 132 caractères. Le caractère ! rencontré sur une ligne indique que tout ce qui suit est un commentaire. Les chaînes sont délimitées par des apostrophes (') ou des guillemets (").

Il est possible de placer plusieurs instructions sur la même ligne en les séparant par un point–virgule ;

```
x=0                   ! une seule instruction sur ligne
i=i+1; x=(2*i-1)*h    ! deux instructions sur une ligne
Nom='SATAN'           ! une chaine de caracteres delimitee par des '
PNom=''Lucifer''      ! une chaine de caracteres delimitee par des ''
```

La fin de ligne est donc un séparateur d'instructions. Si une instruction est trop longue, on peut l'écrire sur plusieurs lignes grâce au caractère `&`, comme dans

```
print *,'Montant  HT : ',montant_ht,  &
        '        TVA : ',tva       ,  &
        'Montant TTC : ',montant_ttc
```

Lors de la coupure d'une chaîne, la suite de la chaîne doit être précédée de `&`

```
print *,'Entrer un nombre entier &
        &compris entre 10 et 100 '
```

Enfin la structure générale d'un programme Fortran 90 (en format libre) est la suivante

```
program MONPROG      ! debut du programme
implicit none        ! obligatoire pour eviter des problemes
! toutes les declarations
...
! instructions executables
...
end program MONPROG  ! fin du program
```

Le «format fixe» correspond au format du Fortran 77. Il n'est pas recommandé pour écrire des programmes Fortran 90. C'est plus pour permettre la compilation en Fortran 90 de programmes en Fortran 77 grâce à une option de compilation (`-qfixed` sur IBM, `-fixedform` sur SGI). Il est strictement interdit d'utiliser les deux formats dans un même fichier (même dans des sous–programmes distincts) car les commentaires en «format fixe» (en fait ceux de Fortran 77) ne sont pas les mêmes en «format libre» et le compilateur vous le fera savoir !

Remarque 1.1 *Les mots clés du Langage Fortran 90 (tels que* `program`, `end`, `integer`, `real`, `print`, `if`,*...) ne sont pas des mots réservés. Autrement dit, il est possible de les utiliser comme identificateurs sans pour autant qu'ils perdent leur signification prédéfinie. Le choix de la «bonne signification» se fait alors en fonction du contexte avec des (mauvaises) surprises éventuelles.* ◇

1.2 Les types scalaires

En Fortran 90, il existe cinq types de variables scalaires

`character`	pour les chaînes d'un ou plusieurs caractères.
`logical`	pour les variables booléennes, i.e. ne prenant que deux valeurs possibles : `.true.` (vrai) ou `.false.` (faux).
`real`	pour les nombres réels.
`integer`	pour les nombres entiers relatifs.
`complex`	pour les nombres complexes, i.e. de la forme $x + iy$.

La forme générale d'une déclaration de variable est[1]

```
type [,liste_attributs ::] liste_variables
```

La liste d'attributs précise s'il s'agit d'une constante, d'un tableau, d'un pointeur, la visibilité de la variable, etc... Les attributs peuvent être

`parameter`	pour une constante symbolique
`dimension(...)`	pour un tableau
`allocatable`	pour un tableau dynamique
`pointer`	pour un objet accessible par pointeur
`target`	pour un objet cible potentiel d'un pointeur
`save`	pour un objet rémanent
`intent(...)`	pour un paramètre formel
`optional`	pour un paramètre formel
`public, private`	pour une entité définie par un module

Remarque 1.2 *Il existe un type* `double precision` *pour les nombres réels, mais on verra plus tard qu'il n'est qu'un sous–type du type* `real`. ◇

Voici quelques déclarations simples

[1] La notation `[ ]` signifie que le contenu des crochets peut apparaître 0, une ou plusieurs fois.

```
character         :: sexe        ! un caractere
character(len=10) :: nom         ! chaine de 10 caracteres au plus
character(len=*)  :: prenom      ! chaine de longueur inconnue
logical           :: cel         ! celibataire ou pas ?
real              :: taille, poids
integer           :: age
```

Les types scalaires ci–dessus présentent des caractéristiques (encombrement mémoire, domaine couvert, précision, etc.) qui varient d'une machine à une autre. Par exemple, un réel en double précision sur IBM RS/6000 ou NEC SX5 correspond à un réel en simple précision sur CRAY T3E. Même les caractères nécessitent une attention particulière. En effet, si un caractère latin se code sur 1 Octet, 2 Octets seront nécessaires pour coder les idéogrammes des alphabets chinois ou japonais.

Les types scalaires s'adaptent pour permettre à l'utilisateur de spécifier ses besoins précis, en précision ou en encombrement mémoire, sans trop tenir compte de la machine cible. Les types scalaires `integer`, `real`, `logical` et `character` deviennent des noms génériques désignant plusieurs sous–types ou variantes accessibles grâce au paramètre de type `kind` .

Voici quelques déclarations de variables avec variantes

```
integer(4)      :: i,j     ! entier court
integer(kind=8) :: k,l     ! entier long
real(kind=4)    :: x, y    ! reel simple precision
real(8)         :: Tol     ! reel double precision
```

En l'absence du paramètre `kind`, c'est la variante par défaut qui est sélectionnée. A noter que le paramètre `kind` est aussi une fonction intrinsèque qui renvoie la variante de la variable en argument. Mais cette fonction intrinsèque permet surtout de faire des déclarations avec des variantes quelle que soit la machine.

`kind(x)` retourne la valeur associée au sous–type de la variable `x`

`kind(0)` retourne la valeur associée au type entier par défaut

`kind(0.0)` retourne la valeur associée au type réel par défaut

`kind(1.0)` retourne la valeur associée au sous–type réel simple précision

`kind(1.d0)` retourne la valeur associée au sous–type réel double précision

`real (kind=kind(1.d0))` permet de déclarer un réel en double précision quelle que soit la machine.

La notation `1.d0` est la forme exponentielle des réels en double précision (i.e. elle est équivalente à `1.e0`).

1.3 Le typage implicite

Une survivance du Fortran préhistorique est que, en l'absence de toute déclaration, les variables dont les noms commencent par

`i,j,k,l,m,n` sont considérées comme de type `integer` ;

`a,b,...,h,o,p,...,z` sont considérées comme de type `real`.

Il s'ensuit des erreurs difficiles à détecter, par exemple lorsqu'on tape malencontreusement `j` au lieu de `i`. Pour forcer le compilateur à vérifier que chaque variable à été déclarée, il faut placer au début du programme l'instruction

```
implicit none
```

qui inhibe le typage implicite.

1.4 Les constantes littérales

Pour introduire une constante entière dans un programme, on l'écrit simplement sous la forme décimale habituelle, avec ou sans signe, comme

```
-2365   +7665452   467498
```

Pour les constantes réelles, on a le choix entre la forme décimale ou la forme exponentielle, comme dans

```
-0.314159   -.314159
1.e-5   1e-5  3.88e5
```

Pour les réels en double précision, la forme exponentielle s'écrit avec `d` au lieu de `e`

```
1.d0   -.76543d-12
```

Les constantes chaînes de caractères se codent à l'aide d'apostrophes comme

```
'jimmy'  "Shiva"
```

Lors de l'écriture d'une constante, on peut spécifier le sous–type désiré en la suffixant (pour les nombres) ou préfixant (pour les chaînes) par la valeur du sous–type voulu en utilisant le caractère _ comme séparateur :

```
256_4              ! constante  entier court
3.14259_4          ! constante  reel simple
687.9865209876_8   ! constante  reel double
1_'Selena'         ! chaine 1 octet/caractere
```

1.5 Les constantes symboliques

En Fortran 90 les constantes symboliques ne sont plus nécessairement de types scalaires (comme en Fortran 77). L'appel à certaines fonctions élémentaires est possible lors de la déclaration. Pour déclarer une constante symbolique, il suffit de sélectionner l'attribut `parameter` comme dans

```
real, parameter :: pi=3.141516
character(len=*), parameter :: Name='SATAN', Surname="Lucifer"
integer, parameter :: Nmax1=10, Nmax2=30
integer, parameter :: Res=mod(Nmax1,Nmax2)   ! fonction elementaire MOD
integer, parameter :: Nbv=abs(Nmax1-Nmax2)   ! fonction elementaire ABS
```

Les constantes symboliques peuvent être utilisées pour rendre les déclarations de variables avec variantes un peu plus portables.

```
integer, parameter   :: rdouble=8
integer, parameter   :: ishort=4
integer(kind=ishort) :: i, j
real(kind=rdouble)   :: x, y, z
```

En cas de problème de variante, avec une machine, il suffit de modifier les constantes symboliques correspondantes.

1.6 La précision des nombres

Il existe deux fonctions (entières), `select_int_kind` et `select_real_kind` prédéfinies qui permettent de faire le lien entre les besoins de l'utilisateur et le numéro de variante correspondant à un domaine donné.

`select_int_kind(r)` fournit l'entier qui correspond au numéro de variante du type `integer` acceptant au moins `r` chiffres décimaux, c'est–à–dire les entiers dans l'intervalle $[-10^r,\ 10^r]$. La fonction retourne -1 si aucun sous–type ne correspond à la demande.

`select_real_kind(p,r)` fournit l'entier qui correspond au numéro de variante du type `real` susceptible de représenter des nombres réels avec une précision de p et une étendue de r, i.e. des réels autorisant p chiffres significatifs et un intervalle de valeurs positives au moins égal à $[10^{-r},\ 10^r]$. La fonction retourne -1 si la précision demandée n'est pas disponible, -2 si l'étendue desirée n'est pas disponible, et -3 si ni la précision ni l'étendue ne sont disponibles.

Avec les fonctions `select_int_kind` et `select_real_kind`, on peut maintenant écrire des programmes vraiment portables. Voici quelques déclarations avec variantes indépendantes de la machine.

```
integer, parameter :: rprec=select_real_kind(p=9,r=50)
integer, parameter :: iprec=select_int_kind(r=2)
  ...
integer(kind=iprec):: i, j, k
real(kind=rprec)   :: x, y, z
```

Remarque 1.3 *La précision et l'étendue peuvent aussi être évaluées en utilisant les fonctions intrinsèques* `precision` *et* `range`*, voir ch. 7.* ◇

1.7 Initialisation

L'initialisation d'une variable se fait à la déclaration par simple affectation comme dans l'exemple suivant.

```
real, parameter  :: Pi=3.14159, Rayon=6500
character(len=*) :: Jour="Lundi", Mois="Janvier"
integer          :: i=0, nbr=90
real             :: circ=2*Pi*Rayon    ! Initialisation avec une expression
real             :: prec=epsilon(1.0)  ! Avec une fonction intrinseque autorisee
real(8)          :: x=0.d0
```

La fonction `epsilon` renvoie la quantité considérée comme négligeable devant 1, pour le sous–type de l'argument (ici réel simple). En principe les fonctions intrinsèques ne sont pas autorisées lors de l'initialisation sauf quelques-unes comme `abs`, `mod`, `epsilon`,...

Chapitre 2

Expressions, instructions élémentaires

2.1 Les expressions

On distingue 3 types d'expressions en Fortran :
- les expressions arithmétiques
- les expressions logiques
- les expressions de type texte

Chacun des trois types scalaires dispose de ses propres opérateurs intrinsèques. Les expressions sont donc construites à partir de l'un au moins de ces opérateurs et d'au moins un opérande. Dans le cadre de ce cours (Programmation numérique) on ne s'intéressera qu'aux expressions arithmétiques et logiques.

2.1.1 Les expressions arithmétiques

Les expressions arithmétiques sont construites à l'aide d'opérateurs arithmétiques usuels. Ils sont par ordre de priorité décroissante

`**`	élévation à la puissance
`* /`	multiplication et division
`+ -`	addition et soustraction

En cas de priorités identiques, les calculs se font de gauche à droite. Une expression arithmétique peut aussi être construite à partir de fonctions mathématiques intrinsèques ou définies par l'utilisateur. Voici quelques exemples d'expressions arithmétiques

`x-y**3/100.0`	correspond à $x - (y^3/100)$
`a/b/c`	correspond à $a/(bc)$
`-a+c/d`	correspond à $(-a) + (c/d)$
`x**y**z`	correspond à $(x^y)^z$

On peut utiliser les parenthèses pour forcer la priorité et, accessoirement rendre le programme plus lisible. Les expressions précédentes deviennent

```
x-((y**3)/100)
a/(b*c)
```

```
-a+(c/d)
(x**y)**z)
```

Remarque 2.1 (Quotient entre deux entiers) *En Fortran* `3/2` *vaut 1, tandis que* `3.0/2.0` *vaut (approximativement) 1.5. Quand les 2 opérandes sont de type entier le résultat de la division est le quotient entier. Il faut donc bien écrire les constantes littérales de type* `real` *pour éviter les (mauvaises) surprises.* ◇

2.1.2 Les expressions logiques

Les expressions logiques sont construites à partir de comparaison entre expressions numériques et d'opérateurs logiques.

Fortran 90 (comme Fortran 77) dispose de 6 opérateurs de comparaison, présentés dans la Table 2.1.2. Comme Fortran 90 «comprend» Fortran 77, on peut utiliser indifféremment l'ancienne notation (i.e. celle de Fortran 77) ou la nouvelle, plus lisible.

ancienne notation	nouvelle notation	signification
`.lt.`	`<`	inférieur à
`.le.`	`<=`	inférieur ou égal à
`.gt.`	`>`	supérieur à
`.ge.`	`>=`	supérieur ou égal à
`.eq.`	`==`	égal à
`.ne.`	`/=`	différent de

TAB. 2.1 – Opérateurs de comparaison

La priorité des opérateurs de comparaison est inférieure à celle de tous les opérateurs arithmétiques. Les expressions suivantes n'ont donc pas besoin de parenthèses.

```
x**2 < a+b
b**2-4.0*a*c .gt. 0
-b+sqrt(delta) < .5*cos(2.0*omega)
```

Remarque 2.2 (Egalité entre réels) *L'opérateur de comparaison* `==` *ne doit être utilisé pour des expressions réelles qu'avec beaucoup de précaution car il n'y a pas absolue égalité entre deux réels.* ◇

On peut également combiner deux expressions logiques à l'aide des opérateurs logiques classiques, présentés dans la Table 2.1.2.

Remarque 2.3 *On peut imprimer une expression logique. On obtient alors un* `F` *(pour* `.FALSE.`*) ou un* `T` *(pour* `.TRUE.`*)* ◇

opérateur	signification
`.and.`	*et* logique, i.e. vrai si les deux opérandes sont vrais
`.or.`	*ou* logique, i.e. vrai si au moins un opérande est vrai
`.not.`	négation
`.eqv.`	équivalence logique, i.e. vraie si les deux opérandes sont tous vrais ou tous faux, fausse dans le cas contraire
`.neqv.`	non équivalence logique, négation de l'opérateur précédent.

TAB. 2.2 – Opérateurs logiques

2.2 Les instructions élémentaires

2.2.1 L'affectation

L'*affectation* se fait avec le symbole = d'égalité. La forme la plus simple pour l'affectation est

```
variable = constante
```

La forme générale est

```
variable = expression
```

comme dans

```
s=Pi*R**2
h=sin(a)**2
boole=(x<y .or. abs(z)<eps)  ! affectation d'une expression logique
```

2.2.2 Entrées/Sorties

La saisie des données au clavier se fait par l'instruction `read` simple (il existe une autre forme plus élaborée pour les fichiers).

```
read *,var1,var2,...
```

Les variables `var1, var2,...` sont alors entrées séparées par une virgule.

L'affichage des données à l'écran se fait soit à l'aide de l'instruction `print`, soit avec l'instruction `write` qui est aussi utilisée pour les fichiers, voir ch. 8. La forme générale pour `print` est la suivante

```
print fmt,element1,element2,...
```

`fmt` est soit le caractère `*`, qui représente la sortie standard (i.e. écran), ou une spécification de format sous forme de chaîne de caractères[1]. Voici quelques exemples d'instructions d'impression simple

[1] On rappelle qu'une chaîne de caractère est délimitée par (') ou des guillemets (")

```
print *,"a=",a,"  b=",b,"  c=",c
print *,'La solution est x=',x
```

Pour formater une sortie, il faut spécifier un format. En Fortran 90, la spécification de format se fait à l'aide d'une chaîne de caractères. On peut spécifier du texte, des nombres, des sauts de lignes,... comme dans

```
print '("a=",F15.6,"b=",E15.8)',a,b
print '(/"x=",E15.8/)',x
```

Voici les principaux champs utilisés pour spécifier un format.

`A`	Lecture ou impression d'une chaîne telle quelle. La longueur du champ imprimé sera celle de la chaîne.
`Im`	Lecture/Impression d'un entier sur m colonnes. Il y a cadrage à droite dans le champ imprimé.
`Fm.n`	Lecture/Impression d'un réel sur m colonnes avec n chiffres après la virgule. Il faut tenir compte de la virgule et du signe éventuel du réel dans le choix de m.
`Em.n`	Lecture/Impression d'un réel en notation exponentielle sur m colonnes avec n chiffres après la virgule.
`Gm.n`	Fonctionne comme `Fm.n` mais la représentation dépend de son ordre de grandeur. Si le réel est trop grand, il s'écrira avec un exposant.
`/`	Changement de ligne.
`x`	Ecriture d'un espace

On peut mentionner un facteur de repétition (une constante non nulle sans signe) devant n'importe quel descripteur. Par exemple

```
3I4
```

est équivalent à

```
I4, I4, I4
```

Il est aussi possible d'appliquer ce facteur de répétition à un groupe de descripteurs placés entre parenthèses :

```
2(I3,F15.8)
```

est équivalent à

```
I3,F15.8,I3,F15.8
```

Remarque 2.4 *Une spécification de format est une chaîne de caractères. Elle peut donc être stockée dans une variable ou une constante symbolique de type chaîne.* ◇

Voici un programme qui résume (presque) tout.

```
program AFFICHE
  implicit none
  real(8) :: x,pi
  integer :: i=12345
```

```
   print '(//,27("-"),/,"--- AFFICHAGE D''ENTIERS ---"/,27("-"),/)'
   print '("Entier format adapte         : ",I6)',i
   print '("Entier format non adapte     : ",I4)',i*i
   print '("Avec facteur de repetition   : ",3I6)',i,2*i,3*i

   x=1.d0; pi=4.d0*atan(x)

   print '(//,27("-"),/,"--- AFFICHAGE  DE REELS ---"/,27("-"),)'
   print '("Reel forme naturelle         : ",F12.8)',pi
   print '("Reel forme exponentielle     : ",E15.8)',pi
   print '("Reel format non adapte       : ",F10.8)',100.d0*pi*pi
 end program AFFICHE
```

L'exécution de ce programme donne.

```
---------------------------
--- AFFICHAGE D'ENTIERS ---
---------------------------

Entier format adapte         :  12345
Entier format non adapte     : ****
Avec facteur de repetition   :  12345 24690 37035


---------------------------
--- AFFICHAGE  DE REELS ---
---------------------------
Reel forme naturelle         :   3.14159265
Reel forme exponentielle     :  0.31415927E+01
Reel format non adapte       : **********
```

Chapitre 3

Structuration d'un programme Fortran

Le Fortran 77 ne dispose que de 2 structures de contrôle : la boucle avec compteur et l'alternative. Les branchements inconditionnels à l'aide de `goto` permettaient d'avoir les structures répétitives pré et post testées. Fortran 90 dispose de nouvelles structures qui la classeraient presque dans la catégorie des langages structurés. Toutefois, la notion de branchement n'a pas totalement disparu puisque sont apparus de nouvelles instructions de branchement inconditionnel (`exit`, `cycle`). Bien sûr les vieilles structures de Fortran 77 sont toujours acceptées, par souci de compatibilité.

3.1 Structures alternatives

L'*alternative simple* s'écrit

```
if (condition) then
   action
endif
```

ou encore

```
if (condition) then
   action1
else
   action2
endif
```

Voici un petit exemple d'alternative

```
if (x>y) then
   print *,x,' est plus grand que ',y
else
   print *,x,' est plus petit ou egal a ',y
endif
```

Dans ce petit exemple, chaque ligne représente une instruction à part entière. Donc si on veut en mettre plusieurs sur la même ligne, il faut prévoir des «;». Voici le petit exemple ci–dessus sous forme plus concentrée (pas forcément plus lisible).

```
if (x>y) then; print *,x,' est plus grand que ',y
else; print *,x,' est plus petit ou egal a ',y ; endif
```

Lorsque l'action se limite à une seule instruction, l'alternative simple peut s'écrire aussi

```
if (condition) action
```

Par exemple, l'alternative simple suivante

```
if (x>imax) then
  imax=x
endif
```

peut se réduire à

```
if (x>imax) imax=x
```

L'*alternative complète* (avec des "sinon si") est de la forme

```
if (condition_1) then
   action_1
else if (condition_2) then
   action_2
   ...
else if  (condition_n) then
   action_n
else
   action_0
endif
```

Notez qu'il n'y a qu'un seul `endif` qui ferme l'alternative.

Exemple 3.1 (Racines d'un polynôme de degré 2) Voici un programme qui calcule toutes les racines d'un trinôme en distinguant les différents cas (racines réelles, complexes, doubles, distinctes).

```
program RACINEQ2
  implicit none                                ! ne jamais oublier
  integer, parameter        :: ir8=kind(1.d0) ! sous-type == real double
  real(kind=ir8), parameter :: eps=1d-12      ! precision des calculs
  real(kind=ir8)            :: a,b,c          ! coef. du trinome
  real(kind=ir8)            :: x1,x2, delta
  complex(kind=ir8)         :: z1,z2          ! pour les racines complexes

  ! lecture des coefficients
  print *,'Entrer les coef. reels: a, b, c '
  read *,a,b,c

  if (abs(a)>eps) then

     delta=b*b-4.0*a*c             ! calcul discriminant

     if (delta> 0) then            ! cas classique : 2 racines reelles
        x1=.5*(-b-sqrt(delta))/a
```

```
         x2=.5*(-b+sqrt(delta))/a

         print *,"Deux racines reelles distinctes"
         print '("x1=",F12.8,"  x2=",F12.8)',x1,x2

      elseif (delta<0) then          ! 2 racines complexes
         x1=-.5*b/a                  ! partie reelle des racines
         x2=.5*sqrt(abs(delta))/a    ! partie imag. (valeur absolue)
         z1=cmplx(x1,-x2)            ! conversion -> z1
         z2=cmplx(x1,x2)             ! conversion -> z2

         print *,"Deux racines complexes :"
         print '("z1=(",2F12.8,")","  z2=(",2F12.8,")")',z1,z2

      else                           ! equation (x+.5*b/a)**2=0
         x1=-.5*b/a;
         print *,'("Une racine double",F12.8)',x1
      endif

   else                              ! equation bx+c=0
      if (abs(b)>eps) then
         x1=-c/b
         print '("Une racine reelle x=",F12.8)',x1
      else; print *,'Equation indeterminee '; endif
   endif
end
```

Voici quelques exemples d'exécution

```
 Entrer les coef. reels: a, b, c
 1.5, 2, 5.1897654
 Deux racines complexes :
 z1=( -0.66666667 -1.73649047)  z2=( -0.66666667  1.73649047)

 Entrer les coef. reels: a, b, c
 3.14159, 10.6573, 2
 Deux racines reelles distinctes
 x1= -3.19294328  x2= -0.19938353
```

3.2 Structures itératives (boucles)

Il y a trois types de structures itératives en Fortran 90. A noter que seule la boucle avec compteur `do` existe en Fortran 77.

3.2.1 Boucle avec compteur (boucle `do`)

C'est l'équivalent de la boucle `for` du PASCAL. La forme générale est la suivante

```
[nom:] DO var=debut,fin [,pas]
          instructions
       END DO [nom]
```

Avec

`nom` un identificateur quelconque de la boucle. On verra plus loin son utilité quand on veut "casser" une boucle.

`var` un identificateur d'une variable de type `integer`

`debut, fin, pas` expressions quelconques de type `integer`. Si `pas` est omis, il est pris par défaut égal à 1.

Voici une boucle qui calcule la somme des entiers de 1 à n.

```
s=0
do i=1,n      ! le pas vaut 1 par defaut
   s=s+i
end do
```

Dans le cas où on ne fait que la somme des nombres impairs, la boucle devient

```
s=0
do i=1,n,2
   s=s+i
end do
```

3.2.2 Boucle "tant que" (`do while`)

C'est l'équivalent de la boucle `while` du PASCAL. Cette structure de boucle n'existe pas en Fortran 77. La forme générale est la suivante.

```
[nom:] DO WHILE (expression_logique)
          instructions
       END DO [nom]
```

Pour que la boucle puisse se terminer, il faut que la valeur de `expression_logique` ait des chances d'être modifiée dans les instructions internes. Sinon, on obtient une boucle infinie.

Exemple 3.2 (Limite d'une suite récurrente) Soit la suite récurrente définie par

$$u_{n+1} = \frac{u_n^3 + 3au_n}{3u_n^2 + a}, \quad u_0 = 1$$

où a est un réel strictement positif. La suite u_n converge vers $\sqrt{a}$, pour $a > 0$ donné.

```
program SUITE_RECC
  implicit none                          ! pour inhiber le typage implicite
  integer, parameter :: IterMax=100      ! nb max d'iterations
  integer, parameter :: ir8=kind(1.d0)   ! sous-type reel double
  real(8), parameter :: eps=1d-6         ! precision des calculs
  real(kind=ir8)     :: a
  real(kind=ir8)     :: u,u1             ! termes courant et precedant

  integer            :: iter
  real(kind=ir8)     :: erreur

  ! Lecture de a et alpha
  print *,'Entrer le reel a >0 :'
```

```
  read *,a

  erreur=1
  u1=1
  iter=0
  do while (erreur>eps .and. iter<IterMax)
     iter=iter+1
     u=(u1*u1*u1+3*a*u1)/(3*u1*u1+a)     ! terme courant
     erreur=abs(u-u1)/u                  ! erreur relative entre u et u1
     u1=u
  end do

  print '("La limite de la suite est ",F12.8) ',u
  print '("Apres ",I4," iterations ")',iter

end program SUITE_REC
```

Voici quelques exécutions

```
 Entrer le reel a >0 :
3.14159
La limite de la suite est   1.77245310
Apres    4 iterations

 Entrer le reel a >0 :
32.184789
La limite de la suite est   5.67316393
Apres    5 iterations
```

Remarque 3.1 *Si le programme est destiné à un ordinateur vectoriel ou parallèle, il ne faut utiliser la boucle* `do while` *que lorsque la situation s'y prête vraiment. En effet le programme obtenu risque de ne pas être très optimisé en temps car cette boucle est fortement séquentielle.* ◇

3.2.3 Sortie anticipée d'une boucle : `exit`

Cette instruction sert à interrompre le déroulement d'une boucle. Elle peut apparaître dans n'importe quelle boucle, avec ou sans compteur. Voici un exemple simple.

```
alpha=0.67
s=stock                ! stock>0
do i=1,n
   if (s<0) exit       ! on arrete si s devient negatif
   s=s-alpha*real(i)   ! real(i) convertit l'entier i en reel
end do
```

Lorsque l'instruction `exit` apparaît dans une boucle qui est elle-même imbriquée dans une autre boucle, elle ne met fin qu'à la boucle la plus interne. Dans l'exemple suivant

```
i=0;
do while (i<m)
   u=1;
   do j=1,n
```

```
      s=m*m-i*j
      if (s<0) exit
      u=s*u
   end do
   print *,"i=",i,"  u=",sqrt(u)
end do
```

seule la boucle en `j` est prématurément arrêtée. Si on veut interrompre la boucle en `i`, on peut toujours ajouter une deuxième instruction `exit` mais il y a mieux. Il suffit de donner un nom à la boucle et de préciser juste après l'instruction `exit` le nom de la boucle à interrompre. Si on veut sortir de la boucle en i dans l'exemple ci–dessus, il faut donc faire

```
i=0;
Julie : do while (i<m)      ! on donne un nom a la boucle critique
   u=1;
   do j=1,n
      s=m*m-i*j
      if (s<0) exit Julie   ! on sort de la boucle julie
      u=s*u
   end do
   print *,"i=",i,"  u=",sqrt(u)
end do Julie                ! fin Julie
```

3.2.4 Bouclage anticipé : `cycle`

Cette instruction permet de passer prématurément au tour de boucle suivant. Elle marche avec toutes les boucles avec les mêmes mécanismes que l'instruction `exit`. Pour les boucles avec compteur non prédéfini, il faut veiller à ne pas "cycler" avant d'avoir incrémenter le compter.

```
j=0
Emma : do while (j<10)
   j=j+1
   s=real(j*j)
   do i=1,n
      s=s+i
      if (s>seuil) cycle Emma  ! on passe au prochain tour de boucle de Emma
   end do
   print *,"Somme =",s
end do emma
```

3.2.5 Boucle infinie : instruction `do`

Il existe une boucle "sans fin" en Fortran, qui peut sembler curieuse à première vue. La forme générale de cette boucle est la suivante.

```
[nom :] do
   ...
   ...
end do
```

En pratique, il faut arrêter la boucle avec l'instruction `exit` comme dans l'exemple suivant.

```
do
   print *,"Entrez un entier positif"
   read *,i
   if (i>0) exit     ! on arrete si i>0
end do
```

3.3 Structure de choix multiple : l'instruction `select case`

Fortran 90 (contrairement à la version 77) dispose d'une vraie structure de choix multiple avec l'instruction `select case`. La forme générale est la suivante (avec la convention `[]` pour les options).

```
[nom :] select case (exp_scal)
   case (selecteur) [nom]
      ...
   [ case default [nom]
      ...]
end select [name]
```

Avec :

`exp_scal` expression scalaire de type `integer` ou `character`

`selecteur` liste composée de 1 ou plusieurs éléments de la forme
- valeur
- intervalle de la forme `[valeur1] :valeur2` ou `valeur1 :[valeur2]`

les valeurs concernées devant être du même type que `exp_scal`.

Mais sans plus attendre, voici un exemple "parlant".

```
program SELECTCASE
implicit none
integer :: n
print *,"Donnez un nombre entier "
read *,n
select case (n)
   case (0)
      print *,"n=0"
   case (1,2)
      print *,"n=1 ou n=2"
   case (3:10)
      print *,"3 <= n <= 10"
   case (11:)
      print *,"n >= 11"
   case default
      print *,"n < 0"
end select
end program SELECTCASE
```

Les lignes de la forme `case (...)` sont des instructions à part entière. Si on veut les placer sur la même ligne que l'instruction suivante, il faut utiliser un point–virgule comme séparateur.

3.4 L'instruction `stop`

L'instruction `stop` remplace avantageusement les branchements "sauvages" à l'instruction `end`, de fin de programme, à l'aide de l'instruction `goto`. L'instruction `stop` peut s'utiliser n'importe où comme n'importe quelle instruction exécutable. On peut donc écrire :

```
if (...) then
   print *,"Probleme insurmontable -- on arrete tout"
   stop
endif
```

Chapitre 4

Les tableaux

Un tableau est un ensemble ordonné d'éléments de même type représentés par un identificateur unique ; chaque élément étant repéré par un indice. Comme tous les langages, Fortran permet de manipuler des tableaux. Mais Fortran 90 introduit de nombreuses facilités, fort puissantes et absentes de la plupart des autres langages : opérations globales, manipulation de portion de tableau, affectation conditionnelle, nombreuses fonctions intrinsèques, etc.

Notons que certaines notions liées aux sous–programmes (transmission de tableaux, fonctions à valeurs tableaux, fonctions intrinsèques relatives aux tableaux,...) seront abordées dans les chapitres 6 et 7.

4.1 Quelques définitions utiles

Définition 4.1 (Rang d'un tableau) *Le rang d'un tableau est son nombre de dimensions. Un vecteur est de rang 1, une matrice de rang 2, etc... Un scalaire est considéré comme de rang 0. En Fortran 90, un tableau peut avoir jusqu'à 7 dimensions au maximum.*

Définition 4.2 (Etendue d'un tableau) *Dans chaque dimension, un tableau a une étendue, qui est le nombre de composantes du tableau dans cette dimension.*

Définition 4.3 (Profil d'un tableau) *Le profil d'un tableau est la suite des étendues de ce tableau selon ses dimensions successives sous forme d'un vecteur d'entiers (soit 1 entier pour un vecteur, 2 pour une matrice, etc.).*

Le produit des étendues représente la taille du tableau, i.e. son nombre d'éléments.

Définition 4.4 (Tableaux conformants) *Deux tableaux sont dits conformants s'ils ont le même profil. Par convention un scalaire est conformant avec tout tableau.*

4.2 Déclaration, initialisation

Pour déclarer un tableau, il suffit de préciser l'attribut `dimension` (après le type) lors de sa déclaration. Voici quelques déclarations de tableaux.

```
integer, dimension(5)       ::  idx   ! simple vecteur
real(8), dimension(3,4)     ::  A     ! matrice 3 lignes 4 colonnes
real, dimension(-1:10,0:10) ::  C     ! matrices 12 lignes 11 colonnes
```

Comme on le constate, lorsque que la valeur initiale des indices n'existe pas, elle est prise par défaut égale à 1. Ainsi, la déclaration

```
integer, dimension(5) :: v
```

est équivalente à

```
integer, dimension(1:5) :: v
```

On peut maintenant tester les définitions vues plus haut à partir des déclarations suivantes.

```
real, dimension(-5:4,0:2)  :: x
real, dimension(0:9,-1:1)  :: y
real, dimension(2,3,0:5)   :: z
```

Les tableaux `x` et `y` sont de *rang* 2, tandis que `z` est de *rang* 3. L'étendue des tableaux `x` et `y` est 10 dans la première dimension et 3 dans la deuxième. Ils ont le même profil : le vecteur (10 3). Ils sont donc conformants. Leur taille est égale à 30. Le profil du tableau `z` est le vecteur (2 3 6). Sa taille est égale à 36.

Définition 4.5 (Constructeur de tableau) *Un constructeur de tableau est une liste de scalaires (de même type !) dont les valeurs sont encadrées par les caractères* `(/` *et* `/)`.

La liste peut être explicite comme

```
(/3, 5, 1, 8, 12/)
```

ou comportée une boucle implicite comme

```
(/ (3*i+1, i=1,3) /)    ! liste (/ 4,7,10 /)
```

Pour le compteur, on utilise la même règle que dans la boucle `do`, i.e.

```
(/ (expression, compteur=debut, fin [,pas]) /)
```

Par exemple

```
(/ (3*i+1, i=1,6,2) /)
```

représente une liste de 3 entiers : `(/ 4,10,16 /)`.

Remarque 4.1 *La syntaxe bizarre* `(/ ... /)` *est utilisée pour éviter tout conflit avec les nombres complexes. En effet le nombre complexe* `c=(0,1)` *n'est différent de la liste* `c=(/ 0,1 /)` *que par la présence des barres obliques.*

Grâce au constructeur, on peut initialiser un tableau au moment de sa déclaration ou lors d'une instruction d'affectation. Toutefois, ceci n'est possible que pour les tableaux de rang 1. Pour les tableaux de rang supérieur à 1, on utilisera la fonction `reshape` détaillée plus loin.

Voici quelques exemples d'initialisation.

```
integer               :: i                        ! compteur
integer, dimension(5) :: idx=(/ 2,6,11,8,2 /)     ! a la declaration
real, dimension(0:90) :: x=(/ (2*i-3, i=0,90) /)  ! notez que i a ete declare
integer, dimension(10):: kx                       ! declaration simple
 ...
kx=(/ (2*i+1,i=1,10) /)                           ! dans une affectation
```

Si un compteur est utilisé dans le constructeur, il doit absolument avoir été déclaré avant utilisation.

Grâce au constructeur de tableau, on peut déclarer des tableaux constants, i.e. en `parameter`. En voici un exemple :

```
integer, dimension(4), parameter :: idx=(/ 2,3,1,2 /)
```

Pour utiliser un constructeur avec un tableau de dimension supérieure à 1, on utilise la fonction `reshape` dont la forme simple est :

```
reshape(source,shape)
```

avec

`source` un tableau de rang 1 de type quelconque. Le tableau `source` contient la liste d'éléments qui serviront dans l'initialisation.

`shape` un tableau d'entiers non négatifs de rang 1. Le tableau `shape` contient le profil de la matrice à remplir.

Considérons les déclarations suivantes :

```
integer, dimension(6)   :: idx=(/ (i,i=1,6) /)       ! liste constante
integer, dimension(3,2) :: v=reshape(idx,(/ 3,2 /))  ! initialisation de v
```

L'instruction d'initialisation

```
v=reshape(idx,(/ 3,2 /))
```

est équivalente à

```
v(1,1)=idx(1)
v(2,1)=idx(2)
v(3,1)=idx(3)
v(1,2)=idx(4)
v(2,2)=idx(5)
v(3,2)=idx(6)
```

L'ordre de remplissage de `v` paraît bizarre, à première vue. Cela vient du Fortran historique. En effet, en Fortran, les matrices sont stockées par colonne de sorte qu'en mémoire une matrice est un vecteur constitué de colonnes de la matrice mises bout a bout. Le remplissage se fait donc par colonne !

On verra, dans la partie opérations globales sur les tableaux, une autre forme d'initialisation (par un scalaire).

4.3 Opérations globales sur les tableaux

On peut affecter une valeur à tous les éléments d'un tableau, puisqu'un scalaire est conformant avec tout tableau. Par exemple, soit la déclaration

```
real, dimension(10) :: u
```

L'instruction d'*affectation globale* suivante

```
u=0.0
```

consiste en l'affectation de la valeur (scalaire) 0 à tous les éléments du tableau `u`. Elle est donc équivalente à la boucle

```
do i=1,10
   u(i)=0.0
end do
```

De la même manière, avec

```
integer, dimension(10,25) :: A
```

l'instruction

```
A=1
```

affecte la valeur 1 aux 250 éléments du tableau `A`.

L'affectation globale d'un scalaire à tous les éléments d'un tableau peut aussi être utilisée pour initialiser un tableau lors de sa déclaration. Dans l'exemple suivant

```
integer, parameter   :: dim=100
real, dimension(dim) :: x=0
```

le tableau `x` est initialisé à 0 lors de sa déclaration.

En Fortran 90, on peut utiliser les opérateurs `+` et `*` directement sur les tableaux (et non seulement sur leurs éléments). On obtient ce qu'on appelle une *expression tableau*, i.e. une expression qui fournit comme résultat un tableau.

Soit les déclarations

```
integer, parameter   :: dim=100
real, dimension(dim) :: x,y,z
```

L'instruction

```
z=x+y
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)+y(i)
end do
```

De même, l'instruction

```
z=x*y
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)*y(i)
end do
```

Donc, dans le cas du produit `x*y` il s'agit d'un produit élément par élément et non d'un produit scalaire (qu'on obtient grâce à la fonction intrinsèque `dot_product`, cf. ch. 7).

On a vu qu'un scalaire était conformant avec tout tableau. Donc dans une expression tableau, on peut avoir des scalaires comme opérandes. Ainsi l'instruction

```
z=x+y+3.14159
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)+y(i)+3.14159
end do
```

Bien sûr, comme on pouvait s'en douter, toutes ces opérations ne sont possibles que si les tableaux opérandes ont le même profil, i.e. le même nombre d'éléments dans chaque dimension. Il n'est pas nécessaire que les indices aient les mêmes limites. Par exemples, avec les déclarations

```
real, dimension(-5:5)  :: x    ! profil (/ 11 /)
real, dimension(11)    :: y,z  ! profil (/ 11 /)
real, dimension(5:16)  :: u    ! profil (/ 11 /)
```

on peut écrire

```
x=z
z=y+u
```

La notion de profil devient encore plus importante lorsqu'il s'agit de tableaux à plusieurs dimensions. Soit les déclarations

```
real, dimension(10,20)      :: a      ! profil (/ 10,20 /)
real, dimension(-2:7,10:29) :: b      ! profil (/ 10,20 /)
real, dimension(10, 0:19)   :: c      ! profil (/ 10,20 /)
```

Alors l'instruction

```
c=a+b
```

est équivalente à

```
do i=1,10
   do j=1,20
      c(i,j-1)=a(i,j)+b(i-3,j+9)
   end do
end do
```

Remarque 4.2 *La valeur d'une expression tableau est entièrement évaluée avant d'être affectée. Ce qui nous permet d'écrire*

```
x=2*x          ! multiplie tous les elements de x par 2
x=x+1          ! augmente de 1 la valeur des elements de x
```

Remarque 4.3 *Dans les expressions de la forme*

```
z=x+y
c=a+b
```

il n'apparaît pas d'emblée que ce sont des tableaux. Pour augmenter la lisibilité du code, on peut écrire

```
z(:)=x(:)+y(:)
c(:,:)=a(:,:)+b(:,:)
```

4.4 Section de tableau, vecteur d'indices

En pratique, avec les tableaux statiques, il y a une différence entre la taille physique d'un tableau (fixée lors de la déclaration) et la taille effective (i.e. celle qui est spécifiée dans les boucles). Donc si on veut initialiser seulement la partie effective d'un tableau ou effectuer les opérations précédentes sur des portions de tableaux, on ne peut plus utiliser les facilités des opérations globales. En Fortran 90 il est possible de faire référence à une partie d'un tableau appelée *section tableau* (ou sous–tableau). Une section tableau est elle–même un tableau (avec un rang et un profil). La section est dite *régulière* si les indices (du tableau d'origine) ayant servi à la créer forment une progression arithmétique. Dans le cas contraire on parle de section `irrégulière`.

4.4.1 Section régulière

On définit une section régulière de la façon suivante

```
var_tableau([debut]:[fin][:pas])
```

avec `debut, fin` et `pas` des expressions entieres quelconques. Lorsqu'une de ces expressions est omise, c'est sa valeur par défaut qui est prise en compte. Les valeurs par défaut sont :

- la valeur du premier indice du tableau pour `debut`,
- la valeur du dernier indice du tableau pour `fin`,
- 1 pour `pas`.

Soit le tableau déclaré ci–après

```
real, dimension(100) :: u
```

Voici quelques sections tableau de `u`

```
u(:)         ! tout le tableau u
u(:50)       ! les 50 premiers elements de u
u(51:)       ! les 50 derniers elements de u
u(10:20)     ! les elements  u(i), 10<=i<=20
u(1:100:2)   ! tous les elements d'indices impairs de u
```

Avec les déclarations suivantes

```
real, dimension(10) :: x
real, dimension(5)  :: y
```

on peut écrire

```
y=x(1:5)        ! on affecte a y les 5 premiers elements de x
```

Une section régulière peut apparaître à gauche d'une instruction d'affectation. On peut donc écrire

```
x(1:5)=1       ! on initialise a 1 les 5 premiers elements de x
```

Voici d'autres affectations correctes

```
y(:)=x(1:5)*x(6:10)+1
y(1:4)=.5*(x(2:5)-x(7:10))
```

Comme on l'a déjà signalé, une expression tableau est entièrement évaluée avant d'être affectée. Ainsi on peut écrire (avec le tableau `x` précédent)

```
x(2:9)=(x(1:8)+x(3:10))/2
```

On remplace chaque composante de `x`, exceptés les deux extrêmes, par la valeur moyenne des deux composantes voisines. Sans utiliser de section tableau, il ne faudrait surtout pas écrire

```
do i=2,9
   x(i)=(x(i-1)+x(i+1))/2
end do
```

car le résultat sera fort éloigné de celui escompté.

Remarque 4.4 *Une section régulière est en réalité une pseudo–boucle* `do`. *Ainsi dans l'instruction*

```
x(1:n)=0
```

il ne se passera rien si $n < 1$, *comme dans une boucle* `do`.

4.4.2 Vecteur d'indices (section non régulière)

Lorsque les indices d'une section tableau ne forment pas une progression arithmétique, on peut les regrouper dans un vecteur d'entiers. Soit les déclarations

```
real, dimension(5) :: x
real, dimension(10)  :: y
```

On sait que `(/ 1,3,7,10 /)` est un vecteur (constant) d'entiers. Alors `y((/ 1,3,7,10 /))` est un tableau de 4 éléments constitué des éléments `y(1), y(3), y(7), y(10)`. On peut donc écrire

```
y((/ 1,3,7,10 /)) = 0.
x(2:5)=y((/ 1,3,7,10 /))
```

Ce qui correspond à

```
y(1)=0.;   y(3)=0.;   y(7)=0.;   y(10)=0.
x(2)=y(1); x(3)=y(3); x(4)=y(7); x(5)=y(10)
```

En général, pour des raisons évidentes de lisibilité, on préfère utiliser une variable tableau, avec un contenu pouvant évoluer. Si l'on déclare

```
integer, dimension(4) :: idx=(/ 1,3,7,10 /)
```

l'affectation précédente devient

```
y(idx)=0.
x(2:5)=y(idx)
```

Les indices peuvent être répétés dans les vecteurs d'indices comme dans

```
integer, dimension(4) :: idx=(/ 2,2,7,10/)
```

Il n'y a aucune ambiguïté lorsqu'on veut seulement utiliser la valeur de cette section tableau. Ainsi, l'instruction

```
x(2:5)=y(idx)
```

correspond à

```
x(2)=y(2); x(3)=y(2); x(4)=y(7); x(5)=y(10)
```

On voit tout de suite qu'on ne peut pas écrire

```
y(idx)=x(2:5)  ! INTERDIT
```

puisque `y(2)` recevrait 2 valeurs distinctes.

Remarque 4.5 *D'une manière générale, lorsqu'une section tableau apparaît à gauche d'une affectation, elle ne doit pas faire intervenir deux fois le même élément.* ◇

4.4.3 Cas des tableaux à plusieurs dimensions

Pour un tableau de rang supérieur à 1, tout ce qu'on vient de voir est valable pour chaque dimension. Soit les déclarations

```
real, dimension(5,10) :: a
real, dimension(5,5)  :: b
```

Alors `a(1:5,1:5)` est un tableau de rang 2 de profil `(/ 5,5 /)`, comme `b`. On peut donc écrire

```
b=2*a(1:5,1:5)+1
```

On peut combiner une section pour certaines dimensions et un indice pour d'autres. La notation `a(2,1:5)` représente un tableau de rang 1 de 5 éléments mais `a(2:2,1:5)` représente un tableau de rang 2 de profil `(/ 1,5 /)`.

Voici quelques exemples de manipulations de tableaux de rang 2 avec des sections de tableaux.

Exemple 4.1 (Permuter deux lignes i et j d'une matrice) L'opération se fait sans boucle mais il nous faut un vecteur de stockage.

```
integer, parameter     :: ndmax=100   !  dimension physique des tab.
integer                :: nd          !  dimension reelle (effective) des tab.
real, dimension(ndmax)       :: x     !  vecteur de travail
real, dimension(ndmax,ndmax) :: A
...
x(1:nd)=A(i,1:nd)                     !  recopie la ligne i dans x
A(i,1:nd)=A(j,1:nd)                   !  recopie la ligne j dans la ligne i
A(j,1:nd)=x(1:nd)                     !  recopie x dans la ligne j
...
```

Exemple 4.2 (Combinaison linéaire des lignes i et j d'une matrice) On effectue la combinaison linéaire de deux lignes i et j d'une matrice et on met le résultat dans la ligne i.

```
integer, parameter     :: ndmax=100   !  dimension physique des tab.
integer                :: nd          !  dimension reelle (effective) des tab.
real                   :: c1,c2       !  coeff. reels de la comb. lineaire
real, dimension(ndmax,ndmax) :: A
...
A(i,1:nd)=c1*A(i,1:nd)+c2*A(j,1:nd)   ! en  une ligne seulement!!
...
```

Exemple 4.3 (Créer une matrice par bloc) Soit A_1 et A_2 des matrices carrées d'ordre n. On veut créer une nouvelle matrice A de la forme

$$A = \begin{pmatrix} A_1 & O \\ O & A_2 \end{pmatrix}$$

où O est une matrice carrée d'ordre n ne contenant que des 0.

```
integer, parameter     :: nmx =100    !  dim. physique de A1 et A2
integer, parameter     :: nmx2=2*nmax !  dim. physique de A
integer                :: n           !  dim. effective de A1 et A2.
integer                :: n2          !  dim. effective de A (n2=2*n)
real, dimension(nmx,nmx)   :: A1, A2
real, dimension(nmx2,nmx2) :: A
...
A(1:n2,1:n2) = 0.                     ! on initialise (juste ce qu'il faut)
A(1:n,1:n) = A1(1:n,1:n)              ! bloc A1
A(n+1:n2,n+1:n2) = A2(1:n,1:n)        ! bloc A2
...
```

4.5 L'instruction `where`

On a déjà vu comment les expressions tableaux simplifient la vie de l'Homo Numericus. Une autre évolution qui simplifie la vie est que les fonctions mathématiques élémentaires usuelles ne s'appliquent plus seulement aux scalaires mais aussi aux tableaux et, dans ce cas, retourne un tableau de même profil. Par exemple

```
y(1:n)=sqrt(x(1:n))
```

est équivalent à

```
do i=1,n
   y(i)=sqrt(x(i))
end do
```

Alors se pose le problème du domaine de définition de la fonction $\sqrt{x_i}$. Mais que faire pour sélectionner les bonnes valeurs ? Pour éviter les boucles, il existe un «filtre» en Fortran 90 : c'est l'instruction `where` qui sélectionne les éléments d'un tableau suivant un test. La forme générale de l'instruction `where` est la suivante

```
where (expression_logique)
  bloc_1
elsewhere
  bloc_2
end where
```

Voici un exemple d'utilisation.

```
real, dimension(10) :: x,y
...
where (x>0)
  y=log(x)
elsewhere
  y=1
end where
```

Le bout de code ci–dessus est équivalent à

```
real, dimension(10) :: x,y
integer             :: i
...
do i=1,10
   if (x(i)>0) then
      y(i)=log(x(i))
   else
      y(i)=1
   endif
end do
```

Lorsque `bloc_2` est absent et `bloc_1` se résume en une seule instruction, on peut utiliser la forme simplifiée

```
where (expression_logique) instruction
```

comme dans

```
where (x>0) y=sqrt(x)
```

4.6 Tableaux dynamiques

Il arrive fréquemment que l'on ait à manipuler des tableaux dont les étendues ne sont pas connues lors de l'écriture du programme. Ce qui est le cas lorsque l'étendue d'un tableau varie d'une exécution à une autre. Parfois le tableau ne sert que pendant une partie de l'exécution du programme. En Fortran 77, il était nécessaire de «surdimensionner» les tableaux en question.

Un apport intéressant de Fortran 90 est la possibilité de faire de l'allocation dynamique de mémoire. Pour allouer un tableau dynamiquement, il faut le déclarer avec l'attribut `allocatable`. Le rang du tableau doit être connu. Par exemple pour des tableaux de réels de rang 1 et 2, la déclaration est donc obligatoirement de la forme

```
real, dimension(:),   allocatable :: vecteur   ! vecteur (reel) dynamique
real, dimension(:,:), allocatable :: matrice   ! matrice (reelle) dynamique
```

L'allocation s'effectuera grâce à l'instruction `allocate` à laquelle on indiquera le profil désiré. Voici différentes manières d'allouer les tableaux allouables `vecteur` et `matrice` ci–dessus.

```
allocate(vecteur(n))              ! vect. de taille n
allocate(vecteur(n1:n2))          ! vecteur(i), n1<=i<=n2
allocate(matrice(m,n))            ! mat. de taille m*n
allocate(matrice(m1:m2,n1:n2))
allocate(vecteur(n),matrice(m,n)) ! allocation simultannee
```

La fonction intrinsèque `allocated` permet de savoir si un tableau a été déjà alloué ou non. A noter qu'en cas d'échec d'allocation, à cause d'une mémoire insuffisante par exemple, il y a arrêt de l'exécution. Pour éviter ce comportement brutal, un paramètre optionnel `stat` permet de savoir si l'allocation a réussie ou echouée. Voici un exemple.

```
program ALLOC
  implicit none
  real, dimension(:,:), allocatable :: a     ! tableau dynamique
```

```
   integer                          :: m,n   ! futur profil du tableau
   integer                          :: aerr  ! pour l'erreur d'allocation

   print *,'Entrer le profil de la matrice (m,n) :'
   read *,m,n
   ...
   if (.not. allocated(a)) then   ! a n'est pas encore alloue
      allocate(a(m,n),stat=aerr)  ! allocation de a : profil (m,n)
                                  ! recuperation de l'err dans aerr
      if (aerr /= 0) then         ! si aerr<>0, l'alloc a echoue
         print *,''Erreur dans l'allocation du tableau a :''
         stop
      endif
   endif
   ...
   deallocate(a)                  ! on libere l'emplacement de a
   ...
end program ALLOC
```

Exemple 4.4 (Méthode de la puissance) Soit A une matrice réelle d'ordre n. On suppose que les valeurs propres de A sont ordonnées de la façon suivante

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

avec λ_1 de multiplicité 1. Etant donné un vecteur initial q^0, de norme euclidienne 1, considérons pour $k = 1, 2, \ldots$ la méthode itérative suivante, connue sous le nom de *méthode de la puissance*

$$\begin{aligned} v^k &= Aq^{k-1} \\ q^k &= \frac{v^k}{\| v^k \|} \\ \lambda^k &= (Aq^k, q^k) \end{aligned}$$

On montre que λ^k tend vers λ_1, la plus grande valeur propre de A, lorsque $k \longrightarrow +\infty$. On arrête les calculs si

$$\frac{|\lambda^k - \lambda^{k-1}|}{|\lambda^k|} < \varepsilon,$$

où $\varepsilon > 0$ est la précision des calculs.

Le programme ci–apprès calcule une valeur approchée de λ_1 pour la matrice de Hilbert H définie par

$$H_{ij} = \frac{1}{i+j-1}, \quad i, j = 1, \ldots, n.$$

Tous les tableaux utilisés sont dynamiques. On utilise les fonctions intrinsèques suivantes (voir chapitre 7 pour les détails)

`dble`	convertit un entier en réel double
`dot_product`	produit scalaire de deux vecteurs
`matmul`	produit matrice/matrice ou matrice/vecteur avec les contraintes mathématiques usuelles sur le produit matriciel.

```
program HILBERTVP
  implicit none
  ! Plus grande valeur propre de la matrice de Hilbert
  ! Methode de la puissance
  character(len=*), parameter :: fmt='("Iter=",I4,&
                                    &"  lambda=",E15.8,"  Err=",E15.8)'
  integer, parameter          :: IterMax=250          ! nb max d'iterations
  real(8), parameter          :: eps=1.d-8            ! precision du resultat
  real(8), dimension(:,:), allocatable :: a           ! matrice de Hilbert
  real(8), dimension(:),   allocatable :: v,q         ! vect. de travail
  real(8)             :: lambda                       ! plus grande val. propre
  real(8)             :: lambda1, erreur
  integer             :: n,i,j,iter

  ! lecture de la taille de la matrice
  print *,'Entrer la taille de la matrice de Hilbert '
  read *,n

  ! allocation des tableaux a,v,q,aq
  allocate(a(n,n))
  allocate(v(n))
  allocate(q(n))


  ! remplissage de la matrice de Hilbert
  do i=1,n
     a(i,:)=(/ (1.d0/dble(i+j-1), j=1,n) /)   ! utilisation d'une liste
  end do

  ! initialisation de q tel que |q|=1
  q=0; q(1)=1.d0

  lambda1=1.d0; erreur=1
  iter=0;
  do while (erreur>eps .and. iter<IterMax)    ! boucle principale
     iter=iter+1
     v=matmul(a,q)                            ! v=Aq
     q=v/sqrt(dot_product(v,v))               ! q=v/|v|
     lambda=dot_product(q,matmul(a,q))        ! (Aq,q)
     erreur=abs(lambda-lambda1)/abs(lambda)   ! erreur relative
     lambda1=lambda
     print fmt,iter,lambda,erreur             ! affichage
  enddo

  print '("La plus grande valeur propre est : ",F15.10)',lambda
  print '("Nombre d''iterations necessaires : ",I4)',iter
end program HILBERTVP
```

Avec des tableaux statiques, il aurait fallu préciser les sections de tableaux concernés lors de l'appel des fonctions intrinsèques. Par exemple pour le produit scalaire, il aurait fallu écrire

```
dot_product(v(1:n),v(1:n))
```

Voici quelques exemples d'exécution.

```
Entrer la taille de la matrice de Hilbert
4
Iter=   1  lambda= 0.14911498E+01  Err= 0.32937658E+00
Iter=   2  lambda= 0.15000983E+01  Err= 0.59652573E-02
Iter=   3  lambda= 0.15002128E+01  Err= 0.76327551E-04
Iter=   4  lambda= 0.15002143E+01  Err= 0.97031100E-06
Iter=   5  lambda= 0.15002143E+01  Err= 0.12333994E-07
Iter=   6  lambda= 0.15002143E+01  Err= 0.15678210E-09
La plus grande valeur propre est :    1.5002142801
Nombre d'iterations necessaires :    6

Entrer la taille de la matrice de Hilbert
25
Iter=   1  lambda= 0.18439228E+01  Err= 0.45767793E+00
Iter=   2  lambda= 0.19431849E+01  Err= 0.51082167E-01
Iter=   3  lambda= 0.19511112E+01  Err= 0.40624852E-02
Iter=   4  lambda= 0.19517082E+01  Err= 0.30586424E-03
Iter=   5  lambda= 0.19517529E+01  Err= 0.22916229E-04
Iter=   6  lambda= 0.19517562E+01  Err= 0.17162877E-05
Iter=   7  lambda= 0.19517565E+01  Err= 0.12853585E-06
Iter=   8  lambda= 0.19517565E+01  Err= 0.96262572E-08
La plus grande valeur propre est :    1.9517565153
Nombre d'iterations necessaires :    8

Entrer la taille de la matrice de Hilbert
1000
Iter=   1  lambda= 0.19917362E+01  Err= 0.49792548E+00
Iter=   2  lambda= 0.23071073E+01  Err= 0.13669546E+00
Iter=   3  lambda= 0.24056663E+01  Err= 0.40969537E-01
Iter=   4  lambda= 0.24332490E+01  Err= 0.11335741E-01
Iter=   5  lambda= 0.24405716E+01  Err= 0.30003547E-02
Iter=   6  lambda= 0.24424821E+01  Err= 0.78220227E-03
Iter=   7  lambda= 0.24429781E+01  Err= 0.20301830E-03
Iter=   8  lambda= 0.24431066E+01  Err= 0.52628135E-04
Iter=   9  lambda= 0.24431400E+01  Err= 0.13638216E-04
Iter=  10  lambda= 0.24431486E+01  Err= 0.35339411E-05
Iter=  11  lambda= 0.24431508E+01  Err= 0.91569552E-06
Iter=  12  lambda= 0.24431514E+01  Err= 0.23726861E-06
Iter=  13  lambda= 0.24431516E+01  Err= 0.61479285E-07
Iter=  14  lambda= 0.24431516E+01  Err= 0.15930047E-07
Iter=  15  lambda= 0.24431516E+01  Err= 0.41276765E-08
La plus grande valeur propre est :    2.4431516130
Nombre d'iterations necessaires :   15
```

4.7 Entrées/Sorties de tableaux

Une instruction d'entrée–sortie peut contenir n'importe qu'elle forme de tableau (éléments, sous–tableau, tableau). Soit les déclarations

```
integer, dimension(5) :: m
real, dimension(5,5)  :: x
```

On peut seulement faire apparaître les éléments, comme dans

```
read  *, m(1), m(2)
print *, x(1,2), x(1,3), x(1,5)
```

On peut aussi utiliser le nom du tableau ou des expressions tableaux, comme dans

```
read  *, m       ! equivalent a read *, m(1),m(2),m(3),m(4),m(5)
print *, x+1     ! equivalent a print *,x(1,1)+1,x(2,1)+1,x(3,1)+1,...
```

Dans ce cas chaque nom de tableau représente la liste de tous ses éléments. Pour un tableau de rang 1, l'ordre est naturel. Pour les tableaux de rang supérieur à 1, l'ordre de la liste est celui d'arrangement des éléments en mémoire, c'est–à–dire colonne par colonne.

On peut aussi utiliser une section de tableau. Ainsi les instructions

```
read  *, m((/1,4,2/))
print *, x(1:3,2)
```

sont équivalentes à

```
read  *, m(1), m(4), m(2)
print *, x(1,2), x(2,2), x(3,2)
```

Toutefois, dans le cas d'une lecture, il faut éviter que, dans une même section, le même élément ne soit cité deux fois. Ainsi

```
read  *, m((/2,4,2/))
```

est incorrect puis qu'équivalent à

```
read  *, m(2),m(4),m(2)
```

Une alternative aux sections tableau est l'utilisation de listes implicites comme dans

```
read  *, (m(i),i=1,4)
```

Pour obtenir l'affichage d'un tableau de rang 2 suivant l'ordre naturel, on combine liste implicite et boucle

```
do i=1,5
   print *,(x(i,j), j=1,5)
end do
```

Exemple 4.5 (Affichage de la matrice de Hilbert) Voici un programme qui affiche la matrice de Hilbert dans le format usuel, i.e. ligne par ligne.

```
program HILBERTMAT
  implicit none
  character(len=10), parameter  :: fmt='(8F15.8)'  ! constante format
  integer, parameter            :: nmax=10         ! taille max
  real(8), dimension(nmax,nmax) :: h               ! matrice de Hilbert
  integer                       :: n               ! taille reelle
  integer                       :: i,j

  print *,'Entrer la taille de la matrice '
  read *,n

  ! remplissage de la matrice de Hilbert
  do i=1,n
     do j=1,n
        h(i,j)=1.d0/dble(i+j-1)
     end do
  end do

  ! affichage
  print '(//"Matrice de Hilbert d''ordre : ",I3/)',n
  do i=1,n
     print fmt,(h(i,j),j=1,n)
  end do
end program HILBERTMAT
```

Voici quelques résultats d'exécution.

```
Entrer la taille de la matrice
3


Matrice de Hilbert d'ordre :   3

    1.00000000     0.50000000     0.33333333
    0.50000000     0.33333333     0.25000000
    0.33333333     0.25000000     0.20000000



Entrer la taille de la matrice
5


Matrice de Hilbert d'ordre :   5

    1.00000000     0.50000000     0.33333333     0.25000000     0.20000000
    0.50000000     0.33333333     0.25000000     0.20000000     0.16666667
    0.33333333     0.25000000     0.20000000     0.16666667     0.14285714
    0.25000000     0.20000000     0.16666667     0.14285714     0.12500000
    0.20000000     0.16666667     0.14285714     0.12500000     0.11111111
```

On décide de remplacer la déclaration de la constante de format par

```
character(len=10), parameter    :: fmt='(8E15.8)'  ! constante format
```

On obtient comme affichage :

```
Entrer la taille de la matrice
4


Matrice de Hilber d'ordre :   4

0.10000000E+01 0.50000000E+00 0.33333333E+00 0.25000000E+00
0.50000000E+00 0.33333333E+00 0.25000000E+00 0.20000000E+00
0.33333333E+00 0.25000000E+00 0.20000000E+00 0.16666667E+00
0.25000000E+00 0.20000000E+00 0.16666667E+00 0.14285714E+00
```

Chapitre 5

Types dérivés (structures), pointeurs

En Fortran 77, les seuls types non scalaires sont les tableaux. L'utilisateur ne peut donc pas définir ses propres types de données comme dans la plupart des autres langages. Quant aux notions de pointeurs et de gestion dynamique de la mémoire... ce sont des astres inaccessibles depuis la planète Fortran 77. En Fortran 90, tout devient possible.

5.1 Structures ou types dérivés

On a déjà vu, avec les tableaux, comment on pouvait désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice. Parfois, il est plus avantageux de désigner sous un seul nom un ensemble de valeur pouvant être de types différents. A titre d'exemple, en Mécanique (des fluides ou des solides) la résolution numérique d'un problème par éléments finis se fait sur un maillage composé de triangles ou de quadrilatères, eux–mêmes composés d'arêtes et de points. Avec les tableaux il en faut 3 (triangles, arêtes, sommets) pour représenter un maillage. Et ces trois tableaux sont à transborder d'une unité de compilation à une autre, avec leurs dimensions physiques et effectives respectives. En C++, on peut tout représenter en un seul objet.

Voici un exemple de code permettant de déclarer un nouveau type de structure nommé `point` qui représente un point du plan.

```
type point
  real :: x,y
end type point
```

On peut alors déclarer une ou plusieurs variables du nouveau type comme suit

```
type (point) :: a,b,c  ! 3 points du plan
```

On peut même déclarer des tableaux d'objets du type `point` comme dans

```
type (point), dimension(20) ::  coord  ! tableau de 20 points
```

La désignation d'un champ de la structure se note en faisant suivre le nom de la variable du symbole `%` suivi du nom du champ. L'affectation globale entre objets de même type ainsi que l'impression globale sont prédéfinies ! Voici quelques exemples :

```
a%x=2.5           ! affecte la valeur 2.5 au champ x de a
print *,b%y       ! imprime le champ y de b
coord(5)=a        ! affectation globale <=> coord(5)%x=a%x, coord(5)%y=a%y
print *,a         ! impression globale  <=> print *,a%x,a%y
```

Pas la peine d'écrire un constructeur ! Il est prédéfini comme pour les tableaux. Par exemple

```
point(2.5,0)
```

représente une structure de type `point` dans laquelle le premier champ (i.e. `x`) vaut 2.5 et le deuxième champ (i.e. `y`) vaut 0. Un tel constructeur peut apparaître dans une initialisation comme dans

```
type (point) :: Origin=point(0.0,0.0)
```

ou dans une affectation, comme dans

```
...
h=0.1
do i=1,20
   coord(i)=point(i,h*i)
end do
...
```

Un type dérivé peut avoir comme champ un autre type dérivé. Voici quelques types dérivés du type `point`

```
type cercle
  type (point) :: centre
  real         :: rayon
end type cercle

type triangle
  type (point) :: a,b,c
end type triangle
```

Remarque 5.1 *Les attributs* `parameter`, `allocatable` *et* `target` *sont interdits dans un champ.* ◇

5.2 Pointeurs

Si en C/C++ un pointeur est une variable ayant une adresse pour valeur, en Fortran 90 un pointeur n'est en fait qu'un *alias* pour des raisons d'efficacité. Les pointeurs sont typés en Fortran 90 : un pointeur vers un scalaire de type `real` ne pourra être associé qu'à des scalaires de type `real` et rien d'autre. De plus toute variable susceptible d'être cible d'un pointeur doit être déclarée comme cible potentielle grâce à l'attribut `target`. Bref en Fortran 90, les pointeurs sont moins flexibles qu'en C/C++ mais beaucoup plus optimisés.

5.2.1 Déclaration

Un pointeur est une variable déclarée avec l'attribut `pointer`. Son type (et donc son sous–type) et son rang sont fixés une fois pour toutes à la déclaration :

```
real, pointer                   :: pr   ! pointeur -> reel
real, dimension(:,:), pointeur :: pa   ! pointeur -> tab. rang 2
```

Par définition, `pr` ne peut être associé qu'à un scalaire réel du sous–type par défaut et `pa` ne peut être associé qu'à un tableau de réels du sous–type par défaut, de rang 2.

Les variables susceptibles d'être cibles d'un pointeur doivent être déclarées comme telle grâce à l'attribut `target`. Pour nos pointeurs `pr` et `pa`, déjà déclarés, il nous faut donc des cibles potentielles réels et tableaux de réels de rang 2 :

```
real, target                   :: x,y  ! cible d'1 pointeur -> reel
real, dimension(5,3), target   :: a,b  ! cible d'1 pointeur -> tab. rang 2
real, dimension(10,10), target :: c    ! cible d'1 pointeur -> tab. rang 2
```

5.2.2 Affectation d'adresse, de valeur

Un pointeur peut se voir attribuer une (nouvelle) valeur d'adresse par le symbole d'association => grâce à l'opération (binaire)

```
lhs => rhs
```

L'opérande de gauche `lhs` doit toujours être un pointeur, tandis que celui de droite `rhs` peut être soit un pointeur, soit une variable cible.

Avec les pointeurs et cibles déjà déclarés au 5.2.1, considérons les associations :

```
pr => y     ! association de pr et de la cible y
pa => b     ! association de pa et de la cible a
```

Dès lors `pr` et `y`, et `pa` et `b` peuvent être utilisés indifféremment puisque faisant référence à la même entité. Si la valeur de `y` change, celle de `pr` change aussi. Même chose pour `b` et `pa`.

Si `pr2` est un autre pointeur sur un scalaire réel, alors

```
pr2 => pr     ! meme chose que pr => y
```

fait de `pr2` un pointeur vers `y`. La variable `y` possède maintenant 2 alias.

Considérons le bout de code suivant :

```
x=3.14159
pr => y           ! association pr et y
pr=x              ! equivalent (en valeur) a y=x
```

La dernière instruction change la valeur de la cible de `pr`, i.e `y`, qui devient celle de `x`. Si la valeur de `x` change, celle de `pr` et `y` ne changera pas car `pr` n'est pas associé à `x`. Mais

```
pr = 3.0          ! equivalent a y=3.0
```

changera à nouveau la valeur de `y`.

Pour un pointeur tableau, l'association peut se faire avec un tableau tout entier ou avec une *section régulière* de rang cohérent avec celui du pointeur. Un pointeur tableau ne peut être associé avec une section non régulière (vecteur d'indices) d'un tableau.

Avec les mêmes déclarations qu'au 5.2.1, l'instruction

```
pa => a(3:5,1:2)
```

associe `pa` à la section spécifiée du tableau `a`. Dès lors `pa(1,1)` rrprésente `a(3,1)`, `pa(2,2)` représente `a(3,2)`, etc. La fonction `size(pa)` donne 6 et `shape(pa)` retourne `(/ 3,2 /)`. L'affectation

```
pa = 0
```

est alors équivalente à

```
a(3:5,1:2)=0
```

5.2.3 Gestion dynamique

La cible d'un pointeur peut être créée par allocation dynamique de mémoire. Comme on sait déjà le faire pour les tableaux dynamiques, l'instruction `allocate` résèrve de la place mémoire et l'associe à un pointeur. La syntaxe est la même que pour les tableaux dynamiques. Les instructions

```
allocate(pr,stat=ierr1)
allocate(pa(nmx,md),stat=ierr2)
```

allouent de l'espace mémoire, la première pour un scalaire réel et la seconde pour un tableau de rang 2. Ces objets (i.e. les deux zones mémoire ainsi réservées) sont automatiquement associés aux pointeurs `pr` et `pa` respectivement.

Comme pour les tableaux dynamiques, `deallocate` permet de libérer l'emplacement lorsqu'on en a plus besoin. La fonction `associated` permet de connaître à tout moment le statut d'association d'un pointeur :

```
associated(pa)    ! .true. si pa est associe a quelque chose
associated(pa,b)  ! .true. si pa est associe a b
```

Si le pointeur est défini mais non associé, la fonction retourne `.false.` Si le pointeur est indéfini, le résultat de la fonction `associated` l'est aussi. Pour éviter les pointeurs indéfinis, on utilise l'instruction `nullify`. Par exemple

```
nullify(pr)
nullify(pa)
```

permet d'initialiser «proprement» les pointeurs `pr` et `pr`.

5.3 Pointeurs vs. tableaux dynamiques

L'utilité des pointeurs n'est pas toujours évidente en Fortran 90 tant les tableaux dynamiques semblent «faire l'affaire» pour ce qui est de la gestion de la mémoire.

Voici deux programmes `TABPOINT1` et `TABPOINT2`, qui se contentent de faire l'affectation globale entre deux tableaux. L'un utilise 2 tableaux statiques tandis que l'autre utilise 2 pointeurs de tableaux.

```
program TABPOINT1
  !
  ! affectation globale de deux tableaux
  ! methode 1 : tableaux statiques
  !
  integer, parameter       :: nmx=1000, IterMax=1000
```

```
   real, dimension(nmx,nmx) :: a,b
   integer                  :: iter

   a=0; b=0
   do i=1,IterMax
      a=b
   end do
 end program TABPOINT1


 program TABPOINT2
   !
   ! affectation globale de deux tableaux
   ! methode 2 : pointeurs tableaux
   !
   integer, parameter              :: nmx=1000, IterMax=1000
   real, dimension(:,:), pointer   :: pr1,pr2,pr
   real, dimension(nmx,nmx),target :: a,b
   integer                         :: iter

   a=0; b=0
   pr1 => a
   pr2 => b
   do i=1,IterMax
      pr  => pr2
      pr2 => pr1
   end do
 end program TABPOINT2
```

Et voici maintenant les statistiques d'exécution des deux programmes (sur SGI Octane 2).

Statistiques d'exécution du programme `TABPOINT1` :

```
------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                        4699: Total Samples
                           0: Samples with incomplete traceback
                     140.970: Accumulated Time (secs.)
                        30.0: Sample interval (msecs.)
------------------------------------------------------------------------
Function list, in descending order by exclusive time
------------------------------------------------------------------------
      excl.secs excl.%   cum.%  incl.secs incl.%    samples  procedure  (dso: file, line)

 [1]    140.970 100.0%  100.0%    140.970 100.0%       4699 tabpoint1 (exe1: ctp1.f90, 1)
 [2]      0.000   0.0%  100.0%    140.970 100.0%       4699 __start (exe1: crt1text.s,103)
 [3]      0.000   0.0%  100.0%    140.970 100.0%       4699 main (libftn.so: main.c, 76)
```

Statistiques d'exécution du programme `TABPOINT2` :

```
------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
```

```
                              7: Total Samples
                              0: Samples with incomplete traceback
                          0.210: Accumulated Time (secs.)
                           30.0: Sample interval (msecs.)
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
      excl.secs excl.%   cum.%  incl.secs incl.%    samples  procedure  (dso: file, line)

 [1]      0.210 100.0%  100.0%      0.210 100.0%       7 tabpoint2 (exe2: ctp2.f90, 1)
 [2]      0.000   0.0%  100.0%      0.210 100.0%       7 __start (exe2: crt1text.s,103)
 [3]      0.000   0.0%  100.0%      0.210 100.0%       7 main (libftn.so: main.c, 76)
```

En conclusion le programme avec tableaux a duré 140s environ, tandis que celui avec les pointeurs n'a même pas mis 1s. Que s'est–il passé ? C'est simple, le programme avec tableaux change le contenu des tableaux, tandis que celui avec les pointeurs se contente de changer ... les noms des tableaux. Comme la taille des tableaux est de 1000000 et qu'il y a 1000 itérations, le programme `TABPOINT1` a effectué $1000 \times 10^6 = 10^9$ affectations tandis que `TABPOINT2` n'a effectué que 1000 affectations.

La morale de tout ça : Dans les méthodes itératives, on a en permanence les deux solutions approchées les plus récentes (cf. exemple 3.2). Il y a donc à chaque fin d'itération échange de valeurs entre les deux variables correspondantes. Dans le cas où les variables en question sont des tableaux de grande taille, les exemples ci–dessus montrent que l'utilisation de pointeurs tableaux peut faire économiser un temps considérable.

Chapitre 6

Procédures et fonctions

Un code Fortran digne de ce nom est en général constitué d'un programme principal et de plusieurs sous–programmes, la plupart du temps dans des fichiers séparés. En Fortran 77, chaque sous–programme est une unité de compilation distincte. Elle est compilée comme un tout, indépendamment de toutes les autres et du programme principal susceptible de l'utiliser. La visibilité, entre unités de compilation, est donc très réduite et le contrôle de cohérence des arguments très limité. On parle alors d'*interface implicite*. C'est l'interface par défaut du Fortran 90. Fortran 90 introduit la notion d'*interface explicite* qui augmente la visibilité entre unités de compilation. En outre la vocation des arguments peut être précisée. D'où un meilleur contrôle, par le compilateur, de la cohérence des arguments.

On distingue deux types de sous–programmes en Fortran : les fonctions ou sous–programmes «expression» et les procédures ou sous–programmes «instruction». Les fonctions intrinsèques (ou prédéfinies) seront étudiées au chapitre suivant. A ces notions classiques s'ajoute, en Fortran 90, la notion de *module* qui permet de fiabiliser la communication entre unités de compilation. On nommera unité de programme un module, un sous–programme externe ou le programme principal.

6.1 Les procédures

6.1.1 Procédure externe (interface implicite)

On commence par étudier les procédures externes simples (i.e. sans interface), héritage de Fortran 77. C'est le cas qui conduit le plus souvent à des erreurs non détectées à la compilation. Mais c'est sous cette forme qu'existe la plupart des sous–programmes des bibliothèques (héritage oblige). Voici un exemple de procédure (calcul des racines réelles d'un trinôme) :

```
subroutine trinome(a,b,c,x1,x2)
  implicit none                    ! tres important
  real, intent(in)  :: a,b,c       ! arguments donnees (coefficients)
  real, intent(out) :: x1,x2       ! arguments resultats (racines eventuelles)
  real              :: delta       ! discriminant (variable locale)

  delta=b*b-4.0*a*c;
  if (delta<0) then
     print *,"Delta<0  : Pas de racines reelles"
     x1=0; x2=0
```

```
   else
      x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
   endif
 end subroutine trinome
```

Comme on le voit, la structure d'une procédure est assez voisine de celle du programme principal : en-tête (commençant par le mot clé `subroutine`), déclarations, instructions exécutables et l'instruction `end`. L'en–tête de la `subroutine` contient simplement la liste des arguments sans leurs types. Ces arguments sont appelés *arguments formels* (ou paramètres formels ou muets) par opposition à ceux utilisés lors de l'appel de la procédure appelés *arguments effectifs*. Les déclarations des arguments se font à l'intérieur de la procédure. L'attribut (optionnel) `intent` permet de préciser la vocation des arguments pour des vérifications supplémentaires lors de la compilation. En Fortran 90, on peut distinguer 3 types d'arguments :

`intent(in)`	argument donnée. Sa valeur ne doit pas être modifiée dans la procédure. L'argument se comporte comme une constante locale à la procédure. A l'appel, l'argument effectif peut être une variable existante, une constante littérale, une expression,...
`intent(out)`	argument résultat. La procédure ne doit pas utiliser sa valeur mais seulement lui en fournir une. C'est l'exécution de la procédure qui fixe l'état de l'argument qui est initialement indéterminé. A l'appel, l'argument effectif doit toujours être une variable existante (allouée ou non).
`intent(inout)`	argument donnée et résultat. La procédure peut utiliser sa valeur mais doit aussi lui en fournir une. A l'appel, l'argument effectif doit toujours être une variable existante (allouée ou non).

La vocation d'un argument ne doit pas être confondue avec la manière dont l'information correspondante est réellement transmise (par adresse ou par valeur). En Fortran 90 c'est le compilateur qui est libre de choisir le mode de transmission approprié.

Pour l'appel, on utilise le mot clé `call` comme dans l'exemple ci–dessous.

```
 program racine
   implicit none                                  ! ne pas oublier
   real  :: a,b,c                                 ! coef. du trinome
   real  :: x1,x2                                 ! solutions eventuelles

   print *,'Entrer les coef. a, b, c du trinome :'
   read *, a, b, c                                ! lecture a,b,c

   call trinome(a,b,c,x1,x2)                      ! appel a la procedure trinome

   print *,'Les racines sont : ',x1,x2            ! affichage solutions x1,x2
 end program racine
```

Comme signalé plus haut, la procédure `trinome` n'est visible du programme principal qu'a travers son en–tête. Il n'y a donc aucun contrôle de cohérence des arguments lors de la compilation. Voici quelques exemples d'appels (corrects ou non) qui «passent» à la compilation

```
call trinome(1,0,-4,x1,x2)  ! correct a,b,c const. dans la procedure
call trinome(1,0,a*a,x1,x2) ! correct var avec attribut intent(in)
call trinome(a,b,c,1.5,x1)  ! argument constante numerique --> DANGER
```

```
call trinome(a,b,c,i,j)     ! pas correct si i,j de type integer
call trinome(a,c,x1,x2)     ! pas correct oubli de b
```

Remarque 6.1 *Le mode de transmission par défaut n'est pas* `intent(inout)` *car dans ce cas l'argument effectif doit toujours être définissable. Ce qui n'est pas obligatoirement nécessaire avec le mode par défaut (l'argument pouvant même être une constante).*

Remarque 6.2 *On peut, comme en Fortran 77, ne pas préciser la vocation des arguments. Dans ce cas le compilateur ne fera aucun contrôle sur l'utilisation des arguments dans la procédure.*

Remarque 6.3 *Un argument effectif doit avoir l'exact sous–type de l'argument formel auquel il correspond sauf pour les chaînes dont les longueurs peuvent varier.*

6.1.2 Procédure interne (interface explicite)

Sans atteindre la totale liberté d'emboîtement des langages à structure, Fortran 90 permet la définition de procédure à l'intérieur du programme principal ou de procédures externes. Toutefois, le niveau d'emboîtement est limité à 1 : une procédure interne ne peut en contenir à son tour. Une procédure interne n'est accessible qu'à son hôte. Voici comment se présenterait le programme `racine` du 6.1.1 si nous avions fait de `trinome` une procédure interne.

```
program racine
  implicit none                                  ! instruction globale
  real  :: a1,b1,c1                              ! variables globales
  real  :: y1,y2                                 ! variables globales

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a1,b1,c1                               ! lecture coef.

  call trinome(a1,b1,c1,y1,y2)                   ! appel a  trinome

  print *,'Les racines sont : ',y1,y2            ! affichage solutions

  !--- Fin partie executive de racine ------------

  contains                          ! mot cle indiquant la presence
                                    ! de sous-prog. internes

  subroutine trinome(a,b,c,x1,x2)
    real, intent(in)  :: a,b,c      ! arguments donnees (coefficients)
    real, intent(out) :: x1,x2      ! arguments resultats (racines eventuelles)
    real              :: delta      ! discriminant (variable locale)
    delta=b*b-4.0*a*c;
    if (delta<0) then
      print *,"Delta<0  : Pas de racines reelles"
      x1=0; x2=0
    else
```

```
      x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
    endif
  end subroutine trinome

end program racine   ! Fin de racine
```

La définition de la procédure `trinome` est la même qu'au 6.1.1 mais elle est placée entre le mot clé `contains` et la fin du programme `racine`. Le mot clé `contains` précise que le programme `racine` contient des procédures internes dont la définition vient à la suite. La procédure interne a accès à *toutes les variables* définies par son hôte qui deviennent de facto des variables globales avec toutes les (fâcheuses) conséquences qui peuvent survenir. D'où le changement de noms des variables de `racine` pour bien distinguer (à la lecture) variables locales et globales.

La procédure interne étant un cas d'interface explicite, la visibilité entre appelant et appelé est maximale. D'où un meilleur contrôle de la cohérence des arguments lors de la compilation. Des appels incohérents tels que

```
call trinome(a,b,c,1.5,x1)  ! argument constante numerique --> DANGER
call trinome(a,b,c,i,j)     ! pas correct si i,j de type integer
call trinome(a,c,x1,x2)     ! pas correct oubli de b
```

ne «passeraient» plus à la compilation.

Si l'interface explicite par procédure (ou fonction) interne est simple et permet de résoudre à la compilation tous les cas d'erreurs de cohérence d'arguments, elle présente deux inconvénients majeurs qui limitent son utilisation :

- la procédure interne n'est pas visible de l'extérieur,
- programmation assez lourde et non modulaire.

6.2 Les fonctions

Il y a deux types de fonctions en Fortran

- les fonctions intrinsèques, c'est–à–dire fournies avec le langage (étudiées au chapitre suivant),
- les fonctions définies par l'utilisateur.

La définition d'une fonction, par l'utilisateur, se fait de manière assez voisine de celle d'une procédure. Une fonction pourra être interne ou externe. Tout ce qui a été dit précédemment sur les variables locales, les variables globales et les arguments reste valable.

6.2.1 Fonctions externes (interface implicite)

Voici la forme générale d'une fonction externe en Fortran 90.

```
[type] function nomfct(liste_arguments)
  implicit none
  [type nomfct]
  ! declaration arguments
  ...
  ! declaration variable locale
  ...
  nomfct=...    ! valeur de la fonction
end function nomfct
```

Le type du résultat de la fonction est précisé soit dans l'en–tête (héritage Fortran 77) soit à l'intérieur de la fonction. Voici un exemple de fonction externe qui calcule le n–ième terme de la suite (entière) de Fibonacci $u_n = u_{n-1} + u_{n-2}$, pour $n \geq 2$ avec $u_1 = u_0 = 1$.

```
function fibonac(n)
  implicit none
  integer, intent(in) :: n          ! declaration argument donnee
  integer             :: fibonac    ! declaration resultat
  ! variable locales
  integer             :: u0, u1     ! 2 premiers termes de la suite
  integer             :: u2         ! terme courant
  integer             :: i

  u0=1; u1=1
  do i=2,n
     u2=u1+u0;
     u0=u1;
     u1=u2
  end do

  fibonac=u2                        ! resultat de la fonction fibonac
end function fibonac
```

Voici un programme qui affiche le n–ième terme de la suite de Fibonacci.

```
program FIBO
  implicit none
  integer :: n                  ! terme de la suite
  integer :: fibonac            ! on declare le type de la fonction

  print *,"Entrer un entier >=2 "
  read  *,n

  print *,"Le n-ieme terme de la suite de Fibonacci est :",fibonac(n)

end program FIBO
```

L'exécution du programme `FIBO` donne :

```
Entrer un entier >=2
10
Le n-ieme terme de la suite de Fibonacci est : 89


Entrer un entier >=2
25
Le n-ieme terme de la suite de Fibonacci est : 121393
```

On peut utiliser la fonction `fibonac` comme n'importe quelle fonction intrinsèque, dans une affectation simple comme

```
ii=fibonac(n)
```

ou dans une expression arithmétique

```
k=i*fibonac(10)+n*fibonac(n)
```

Mais il est nécessaire que le compilateur connaisse le type de la fonction `fibonac`. C'est pour cela que dans le programme `FIBO` on déclare la variable `fibonac`. En l'absence de cette déclaration, le compilateur signalera que la variable `fibonac` n'est pas déclarée (en présence de `implicit none`, dans le cas contraire...)

6.2.2 Fonctions internes (interface explicite)

Comme pour les procédures, pour définir une fonction interne, il suffit de la placer entre les mots–clé `contains` et `end` du programme ou du sous–programme hôte. A la différence des fonctions externes, plus besoin de déclarer le type de la fonction dans le programme hôte puisque ce dernier a une visibilité imprenable sur sa fonction interne.

6.3 Variables locales à sous–programme

Dans un programme principal, les variables ont leur emplacement mémoire défini une fois pour toutes : elles sont dites *statiques*. Les variables locales à un sous–programme sont gérées différemment. Fortran 90 a prévu

- de ne leur attribuer un emplacement mémoire qu'au moment où l'on commence à exécuter le sous–programme
- de libérer l'emplacement correspondant à la fin de l'exécution du sous–programme.

On dit que les variables locales sont *automatiques*. La conséquence immédiate est que par défaut leur valeur n'est pas conservée d'un appel à un autre. Mais on peut toujours imposer à une variable locale d'avoir un emplacement permanent et, ainsi, de conserver sa valeur d'un appel au suivant. Il suffit de la déclarer avec l'attribut `save` comme dans

```
subroutine ...
  ...
  integer, save :: p
  real, dimension(10), save :: x
  ...
```

Une autre manière de rendre une variable locale statique est de l'initialiser. En effet, en Fortran 90, si on initialise une variable locale, elle devient d'office statique.

Voici en exemple, une procédure qui se contente de comptabiliser le nombre d'appels et de l'écrire.

```
subroutine JeMeCompte
  implicit none
  integer  :: nb_appels=0    ! variable locale -> statique

  nb_appels=nb_appels+1
  print *,'Appel No ',nb_appels
end subroutine compter
```

6.4 Tableaux transmis en argument d'une procédure externe

La procédure externe étant une unité de compilation indépendante, on voit que se pose le problème de la façon dont on va pouvoir connaître le profil d'un tableau transmis en argument. En fait on utilise une méthode très simple (héritée de Fortran 77 comme les procédures externes) : on transmet en argument le tableau et ses étendues. Cette section est surtout destinée à la compréhension des sous–programmes écrits en Fortran 77. Les modes de transmission plus adaptées à la programmation moderne sont présentés à partir du 6.5.

Les paramètres tableaux les plus simples à passer en argument sont les vecteurs car il suffit de passer aussi la taille en argument. Toutefois, cette dimension ne devra pas dépasser la dimension physique (i.e. celle déclarée dans le programme appelant) du paramètre effectif sous peine de provoquer des catastrophes à l'exécution.

Soit le sous–programme `moyenne` suivant qui calcule la moyenne de n nombres réels

$$S = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

Les n nombres x_i, $i = 1, \ldots, n$ sont naturellement stockés dans le vecteur x.

```
subroutine moyenne(x,n,som)
  implicit none
  integer                          :: n     ! declaration dim.
  real, intent(in), dimension(n) :: x     ! declaration vect. x
  real, intent(out)              :: som   ! moyenne

  ! variables locales
  integer            :: i

  som=0
  do i=1,n                              ! somme sur les elements x(i)
    som=som+x(n)
  end do
  som=som/real(n)                       ! real convertit l'entier n en reel
 end
```

On peut ne pas préciser la dimension de `x` lors de sa déclaration dans `moyenne` en écrivant simplement

```
real, intent(in), dimension(*) :: x
```

Mais dans ce cas, l'option de vérification de la taille des tableaux (`-C` en général) devient inefficace.

Remarque 6.4 *Il existe en Fortran 90 une fonction intrinsèque* `sum` *qui calcule la somme des éléments d'un tableau passé en argument.* ◇

Le programme suivant fait appel au sous–programme `moyenne` pour calculer la moyenne de n nombres.

```
program calcul_moyenne
  implicit none
```

```
  integer, parameter    :: nmax=100    ! dim. physique de x
  integer               :: n           ! dim. reelle du vect. x
  real, dimension(nmax) :: x           ! x(i), i=1,..,n
  integer               :: i
  real                  :: ss

  ! Lecture des donnees
  print *,'Entrer n : '
  read *, n

  do i=1,n
    print *,'Entrer x(',i,')'
    read *, x(i)
  end do

  ! Calcul de la moyenne
  call moyenne(x,n,ss)      ! on ne transmet que l'etendue reelle n

  print *, 'La moyenne est ',ss
end
```

En réalité, c'est l'adresse du premier élément (à utiliser) qui est transmise au sous-programme. La dimension indique alors le nombre d'éléments suivants à utiliser. Le programme appelant peut donc transmettre seulement un bout de vecteur en jouant sur son premier élément et sa longueur. Par exemple, si on veut calculer seulement la moyenne des 10 dernières composantes d'un vecteur à l'aide de la procédure `moyenne`, on fait simplement

```
...
n1=10
call moyenne(x(n-n1+1),n1,ss)
...
```

Dans la procédure `moyenne`, on aura un vecteur à 10 composantes commençant à `x(n-n1+1)`, i.e. les 10 derniers éléments de `x`.

Pour les arguments tableaux autres que les vecteurs il faut absolument transmettre les dimensions physiques et effectives. Compte tenu de la manière dont les éléments d'un tableau sont rangés en mémoire, pour un tableau de rang m (i.e. m indices), les $m-1$ prémières dimensions suffisent.

Soit le programme `calculmatrice` qui fait calculer le produit de deux matrices $A = (a_{ij})$ et $B = (b_{ij})$; A est une matrice $m_a \times n_a$ et B est une matrice $m_b \times n_b$. On sait que la matrice produit $C = (c_{ij}) = A \cdot B$ est une matrice $m_a \times n_b$ dont les éléments sont donnés par la formule

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Le produit matriciel est effectué par un sous–programme `ProduitMat`. Ce produit matriciel n'est possible que si $n_a = m_b$. On testera donc les dimensions avant l'appel du sous–programme `ProduitMat`.

```
program matrice
  implicit none        ! ne jamais oublier
```

```
  integer, parameter         :: nmax=100    ! dim. physique des tableaux
  real, dimension(nmax,nmax) :: A,B,C       ! declaration des tableaux
  integer                    :: ma,na       ! dim. reelles de A
  integer                    :: mb,nb       ! dim. reelles de B
  integer                    :: i,j

  ! Lecture des dimensions
  print *, 'dimension de A : ma na '
  read *, ma,na
  print *, 'dimension de B : mb nb '
  read *, mb,nb

  if (na /= mb) then
    print *,'Le produit matriciel est impossible'
    stop        ! on arrete le programme
   endif
  ...
  call ProduitMat(nmax,ma,na,nb,A,B,C)
  ...
end
```

Le sous-programme `ProduitMat` est alors :

```
subroutine ProduitMat(dmx,mx,nx,ny,x,y,z)
  implicit none    ! Ne jamais l'oublier
  integer      :: dmx       ! dim. physique des tableaux
  integer      :: mx,nx,ny  ! dim. reelles avec my=nx
  real, dimension(dmx,nx)  :: x
  real, dimension(dmx,ny)  :: y,z

  ! Variables locales
  integer      :: i,j,k

  do i=1,mx
  do j=1,ny
     z(i,j)=0.
     do k=1,nx
        z(i,j)=z(i,j)+x(i,k)*y(k,j)
     end do
  end do
  end do
end
```

Voici une autre forme plus compacte pour `ProduitMat` qui utilise la fonction intrinsèque `dot_product`[1] qui effectue le produit scalaire de deux vecteurs passés en argument.

```
subroutine ProduitMat(dmx,mx,nx,ny,x,y,z)
  implicit none                              ! ne jamais l'oublier
```

[1] `dot_product` = produit scalaire en anglais

```
  integer      :: dmx                          ! dim. physique des tabealux
  integer      :: mx,nx,ny                     ! dim. reelles avec my=nx
  real, dimension(dmx,nx)  :: x
  real, dimension(dmx,ny)  :: y,z

  ! Variables locales
  integer      :: i,j

  do i=1,mx                                    ! x(i,1:nx) ligne i de x
  do j=1,ny                                    ! y(1:nx,j) colonne j de y
     z(i,j)=dot_product(x(i,1:nx),y(1:nx,j))   ! produit scalaire ligne*colonne
  end do
  end do
end
```

Remarque 6.5 *Il existe une fonction intrinsèque* `matmul` *qui calcule le produit de deux matrices ou d'une matrice et d'un vecteur avec les contraintes mathématiques habituelles sur les dimensions.*

Comme signalé dans le chapitre sur les tableaux, les matrices sont stockées en mémoire colonne après colonne, i.e. les vecteurs colonnes sont stockés les uns à la suite des autres dans des zones mémoires contiguës. En fait, en mémoire, une matrice n'est qu'un long vecteur composé de colonnes de la matrice. Comme pour tout vecteur, un programme (ou un sous–programme) peut donc transmettre seulement une portion de ce «super–vecteur» à un autre. Par exemple, si je veux faire la moyenne des éléments de la deuxième colonne de la matrice `A` de l'exemple ci–dessus, je code simplement

```
call moyenne(A(2,1),ma,ss)
```

6.5 Bloc interface

Pour éviter les inconvénients de la procédure interne tout en conservant la fiabilité dans la transmission des arguments, Fortran 90 offre une solution : le *bloc interface* qui permet de donner, là où il est présent, une visibilité totale sur l'interface d'une procédure externe. Ce bloc interface peut être créé par copie de la partie déclarative des arguments formels du sous–programme à interfacer. Il sera inséré dans chaque unité de programme faisant appel au sous–programme externe.

Si on veut fiabiliser les appels à la procédure externe `trinome` du 6.1.1, on doit inclure un bloc interface dans le programme `racine` qui devient

```
program racine
  implicit none
  real  :: a,b,c
  real  :: x1,x2

  !------------- BLOC INTERFACE  --------------------------------------
  interface
    subroutine trinome(a,b,c,x1,x2)
      real, intent(in)  :: a,b,c      ! arguments donnees
      real, intent(out) :: x1,x2      ! arguments resultats
    end subroutine trinome
  end interface
```

```
  !---------------------------------------------------------------------

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a, b, c

  call trinome(a,b,c,x1,x2)    ! Appel a la procedure trinome

  print *,'Les racines sont : ',x1,x2
end program racine
```

Dès lors la visibilité du programme `racine` sur la procédure externe `trinome` est totale. Il y a un meilleur contrôle de cohérence des arguments, lors de la compilation. A la différence de la procédure interne, la procédure `trinome` reste une unité de compilation indépendante qui n'a pas accès aux variables du programme `racine`.

6.6 Modules

Comme signalé plus haut, le bloc interface d'un sous–programme est à copier dans chaque unité de programme utilisant cette procédure, ce qui peut paraître fastidieux et de surcroît sujet à de nombreuses erreurs de recopie. La première solution pour éviter cette recopie d'information est d'utiliser l'instruction `include` bien connue des adeptes de la secte C. Mais il y a mieux en Fortran 90 : la notion de `module`. L'organisation générale d'un module ressemble à celle d'un programme principal sans corps d'instruction

```
module nom_module
  [declaration]
  [ contains
      sous programmes exportables]
end module nom_module
```

La partie déclaration, si présente, ne doit pas contenir de fonctions, ni de format, ni déclarer d'objets automatiques. Le mot clé `contains`, s'il est présent, indique la présence de sous–programmes exportables dans le module.

Par défaut, toute entité (type, variable, sous–programme,...) déclarée dans un module est exportable. Toutefois, un module peut réserver des entités pour son usage interne : il les déclare alors comme «privés» dans une déclaration `private` (pour les entités quelconques) ou avec l'attribut `private` (pour les variables) comme dans

```
module m_mod
  integer          :: k
  integer, private :: i,j      ! declaration avec attribut private
  private          :: sprog1   ! declaration private

  contains                     ! mot cle indiquant la presence de sous-prog

  subroutine sprog1
    ...
  end subroutine sprog1
  subroutine sprog2
    ...
```

```
  end subroutine sprog2
end module m_mod
```

Dans cet exemple, seuls sont exportables la variable `k` et le sous–programme `sprog2`.

Le mode d'accès par défaut est donc `public`. Mais ceci peut être inversé grâce à la déclaration

```
private
```

placée juste après l'en–tête du module. Voici le module `m_mod` ci–dessus en mode `private` par défaut.

```
module m_mod
  private
  integer, public   :: k       ! var. exportable
  public            :: sprog2  ! sous-prog. exportable

  contains                     ! mot cle indiquant la presence de sous-prog

  subroutine sprog1
    ...
  end subroutine sprog1
  subroutine sprog2
    ...
  end subroutine sprog2
end module m_mod
```

Dans le cadre des sous–programmes, un module sert surtout à exporter les données, les blocs interfaces, les sous–programmes, etc. On accède aux entités d'un module grâce à l'instruction `use`. Avant d'être utilisé, un module doit être compilé séparément.

Exemple 6.1 (Module avec bloc interface) Voici le module du bloc interface de la procédure `trinome` du 6.1.1.

```
module bi_trinome
interface
    subroutine trinome(a,b,c,x1,x2)
      real, intent(in)  :: a,b,c      ! arguments donnees
      real, intent(out) :: x1,x2      ! arguments resultats
    end subroutine trinome
end interface
end module bi_trinome
```

Ce module doit être compilé séparément avant toute utilisation. Supposons que le fichier qui contient le module avec bloc interface ci–dessus s'appelle `bitrinome.f`. Après compilation on obtient le fichier objet `bitrinome.o` et un autre fichier `bi_trinome.mod` qui contient les informations du module et qui est utilisé par l'instruction `use`. A noter que ce dernier fichier porte le nom donné dans l'en–tête du module et non celui du fichier contenant le module.

Le programme `racine` utilisant la procédure trinome devient alors :

```
program racine
  !---------------- acces au module bloc interface -------------
  use bi_trinome
  !-------------------------------------------------------------
  implicit none
  real  :: a,b,c
  real  :: x1,x2

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a, b, c

  call trinome(a,b,c,x1,x2)       ! Appel a la procedure trinome

  print *,'Les racines sont : ',x1,x2
end program racine
```

Notez que l'instruction `use bi_trinome` apparaît bien avant l'instruction `implicit none`.

Exemple 6.2 (Module avec sous–programme) L'inconvénient avec le bloc interface est surtout que le contrôle de cohérence se fait entre les paramètres effectifs et les paramètres formels de l'interface et non pas ceux du sous–programme lui–même! De plus toute modification dans l'en–tête du sous–programme doit être manuellement répercutée dans le module bloc interface qui doit être recompilé. Dès lors le module avec sous–programme apparaît comme la solution la plus sûre.

Dans le cas de notre procédure `trinome`, il suffit de l'inclure dans un module comme suit.

```
module mtrinome

  contains                          ! mot cle indiquant la presence de sous-prog

  subroutine trinome(a,b,c,x1,x2)
    implicit none
    real, intent(in)  :: a,b,c      ! arguments donnees (coefficients)
    real, intent(out) :: x1,x2      ! arguments resultats (racines eventuelles)
    real :: delta                   ! discriminant (variable locale)
    delta=b*b-4.0*a*c;
    if (delta<0) then
       print *,"Delta<0  : Pas de racines reelles"
       x1=0; x2=0
    else
       x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
    endif
  end subroutine trinome
end module mtrinome
```

Le nom du module doit être différent de celui de la procédure. Dans le programme `racine` utilisant la procédure `trinome` seule l'instruction d'accès au module change et devient

```
  use mtrinome
```

Les modules avec sous–programme (procédure ou fonction) offrent donc une interface d'appel explicite sans les inconvénients du bloc interface.

6.7 Tableaux dynamiques et pointeurs

Un tableau dynamique ne peut apparaître en argument formel d'un sous–programme. Les tableaux dynamiques devront donc être alloués et libérés dans la même unité de programme. Si un tableau dynamique n'est pas libéré avant la fin d'un sous–programme, l'espace mémoire correspondant deviendra inaccessible durant tout le déroulement du programme.

Il est possible de transmettre un tableau dynamique alloué à un sous–programme. Il suffit de prévoir un tableau de profil implicite dans le sous–programme (interface explicite).

Un pointeur apparaissant en argument effectif est (par défaut) traité comme s'il s'agissait de l'objet associé. L'argument formel doit simplement être du même sous–type que le pointeur argument effectif et l'interface quelconque.

Si un sous–programme prévoit un pointeur en argument formel, alors l'argument effectif devra être obligatoirement un pointeur de même sous–type. L'information transmise sera alors l'information contenue dans le pointeur (i.e. adresse, dimension,...) et non celle contenue dans l'objet associé. L'interface explicite est ici obligatoire, sinon c'est l'objet associé qui est transmis avec des conséquences... à l'exécution.

6.8 Interface explicite : nouvelles possibilités

Nous regroupons dans cette section, les nombreux avantages offerts par une interface explicite. On rappelle qu'une interface explicite est obtenue dans les cas suivants : sous–programme interne, bloc interface, module avec bloc interface, module avec sous–programme. Les exemples seront donc donnés dans l'un de ces cas. On rappelle aussi que l'interface des fonctions et procédures intrinsèques (i.e. prédéfinies) est explicite.

6.8.1 Passage de tableaux de "profil implicite"

Grâce à l'interface explicite, plus besoin de se préoccuper du profil du tableau à transmettre. On a juste besoin de connaître son rang.

Reprenons la procédure `ProduitMat` du 6.4. On sait que grâce aux sections de tableaux, on peut transmettre que la partie utile d'un tableau. En outre, il existe une fonction intrinsèque

```
size(array [,dim])
```

qui retourne la taille de `array` ou l'étendue de `array` dans la dimension indiquée via `dim`. On peut donc se dispenser de transmettre les dimensions pour les récupérer à l'intérieur de la procédure avec `size`. La procédure `ProduitMat` retravaillée devient donc.

```
subroutine ProduitMat(x,y,z)
  implicit none                              ! ne jamais l'oublier
  real, dimension(:,:), intent(in)  :: x,y   ! rang de x, y
  real, dimension(:,:), intent(out) :: z     ! rang de z

  ! Variables locales
  integer      :: mx,ny                      ! dim. reelles avec my=nx
  integer      :: i,j                        ! compteur

  mx=size(x,dim=1)                           ! nb. de lignes de x
  ny=size(y,dim=2)                           ! nb. de colonnes de y
```

```
  do i=1,mx
  do j=1,ny
     z(i,j)=dot_product(x(i,:),y(:,j))      ! produit scalaire ligne*colonne
  end do
  end do
end subroutine ProduitMat
```

On a donc juste déclaré le rang des arguments tableaux `x`, `y` et `z`. Le programme `matrice` qui l'utilise doit incorporer, par exemple, son interface.

```
program matrice
  implicit none                              ! ne jamais oublier
  integer, parameter         :: nmax=100     ! dim. physique des tableaux
  real, dimension(nmax,nmax) :: A,B,C        ! declaration des tableaux
  integer                    :: ma,na        ! dim. reelles de A
  integer                    :: mb,nb        ! dim. reelles de B
  integer                    :: i,j
  !------------------- BLOC INTERFACE ---------------------------------
  interface
    subroutine ProduitMat(x,y,z)
      real, dimension(:,:), intent(in)  :: x,y  ! rang de x, y
      real, dimension(:,:), intent(out) :: z    ! rang de z
    end subroutine ProduitMat
  end interface
  !--------------------------------------------------------------------

  ! Lecture des dimensions
  print *, 'dimension de A : ma na '
  read *, ma,na
  print *, 'dimension de B : mb nb '
  read *, mb,nb

  if (na /= mb) then
    print *,'Le produit matriciel est impossible'
    stop        ! on arrete le programme
   endif
  ...
  call ProduitMat(A(1:ma,1:na),B(1:mb,1:nb),C(1:ma,1:nb))
  ...
end program matrice
```

L'appel à `ProduitMat` doit se faire obligatoirement avec des sections de tableau précisant les blocs utiles.

6.8.2 Tableaux automatiques

Comme les autres variables locales non initialisées, les tableaux déclarés localement voient leur emplacement alloué à chaque appel. La nouveauté avec Fortran 90 est qu'on peut faire varier les profils de ces tableaux d'un appel à un autre. Ce qui est très utile lorsqu'on a besoin d'un tableau auxiliaire.

Voici, comme exemple, une procédure qui échange la valeur de deux matrices (i.e. tableaux de rang 2). On utilise la fonction `size` vue à la Section 6.8.1.

```
subroutine SwapMat(x,y)
  implicit none                                          ! ne jamais l'oublier
  real, dimension(:,:)  :: x,y                           ! matrices a echanger
  real, dimension(size(x,dim=1),size(x,dim=2)) :: z      ! matrice auxiliaire

  z=x; x=y; y=z                                          ! C'est tout!!!

end subroutine SwapMat
```

Pour procéder à l'échange, le sous–programme a besoin d'une matrice auxiliaire `z` de même profil que les deux autres. La fonction intrinsèque `size` nous permet de récupérer l'étendue de chaque dimension de la matrice `x`. Ce qui nous permet d'avoir son profil.

Remarque 6.6 *La fonction intrinsèque* `size` *n'étant pas qualifiée pour l'initialisation, il n'est pas possible de récupérer sa valeur en dehors de l'attribut* `dimension`*. Donc impossible de passer par une constante symbolique pour alléger la déclaration de la matrice* `z`*.*

Le programme qui fait appel à `SwapMat` doit simplement avoir une interface explicite avec `SwapMat` pour que le compilateur puisse prévoir de transmettre correctement les informations nécessaires. En voici un exemple avec un bloc interface comme interface explicite.

```
program EchangeMatrice
  implicit none                    ! toujours present
  integer,parameter  :: nmax=100   ! taille max. des tableaux
  real,dimension(nmax,nmax) :: a,b ! tableaux a echanger
  integer                   :: n   ! taille reelle des tableaux

  !------------- bloc interface ----------------------------------
  interface
    subroutine SwapMat(x,y)
      real, dimension(:,:)  :: x,y   ! matrices a echanger
    end subroutine SwapMat
  end interface
  !---------------------------------------------------------------
  ...
  call SwapMat(a,b)                    ! on echange tout le contenu
  ...
  call SwapMat(a(1:n,1:n),b(1:n,1:n))  ! on echange que les blocs utiles
  ...

end  program EchangeMatrice
```

6.8.3 Fonction à valeur tableau

Comme Fortran 90 accepte les expressions tableaux, il est tout à fait naturel qu'il accepte qu'une fonction fournisse un résultat de type tableau. Ce dernier peut être directement incorporer dans des expressions tableaux sans passer par des affectations intermédiaires.

Voici une fonction fournissant comme résultat la matrice de Hilbert d'ordre n. On rappelle que la matrice de Hilbert d'ordre n est la matrice symétrique définie par $H_{ij} = 1/(i+j-1)$, $i, j = 1, 2, \ldots, n$.

```
 function HilbertMat(n)
   implicit none                                  ! Ne jamais oublier
   integer, parameter  :: r8=kind(1.d0)           ! on travaille en double
   integer, intent(in) :: n                       ! taille de la matrice
   real (kind=r8), dimension(n,n) :: HilbertMat ! matrice resultat (ajustable)

   ! variables locales
   integer i,j

   do i=1,n
      HilbertMat(i,:)=(/ (1.d0/dble(i+j-1),j=1,n) /)
   end

 end function HilbertMat
```

Cette fonction `HilbertMat` peut alors être utilisée dans n'importe quelle expression tableau (avec les règles habituelles relatives aux expressions tableaux) à condition que son interface soit connue. Voici un exemple d'utilisation avec un bloc interface comme interface explicite.

```
program MHilbert
   implicit none
   integer, parameter :: nmx=100
   integer, parameter :: ir8=kind(1.d0)
   real, dimension(nmx,nmx) :: H

   integer            :: i, n=4
   !---------------------- Bloc interface ----------------------
   interface
      function HilbertMat(n)
        integer,parameter    :: r8=kind(1.d0)
        integer, intent(in) :: n
        real(kind=r8),dimension(n,n) :: HilbertMat
       end function HilbertMat
   end interface
   !------------------------------------------------------------
   ...
   H(1:n,1:n)=HilbertMat(n)+1   ! matrice de Hilbert d'ordre n + 1
   ...
end program MHilbert
```

La matrice de Hilbert a été fabriquée temporairement par la fonction `HilbertMat`. Son emplacement est alloué lors de l'appel et libéré à la sortie. On peut donc aboutir, dans certains cas, a des économies de mémoire.

6.8.4 Arguments à mot clé

Grâce à l'interface explicite, il devient possible de repérer les arguments d'un sous–programme, non seulement classiquement par leur position mais aussi par le nom même de l'argument formel.

Par exemple avec le module avec bloc interface de la procédure externe `trinome` suivant

```
interface
  subroutine trinome(a,b,c,x1,x2)
    real, intent(in)  :: a,b,c      ! arguments donnees
    real, intent(out) :: x1,x2      ! arguments resultats
  end subroutine trinome
end interface
```

les appels suivants seraient rigoureusement équivalents :

```
call trinome(a1,b1+a1,c1,x,y)              ! appel par position
call trinome(a=a1,b=a1+b1,c=c1,x1=x,x2=y)  ! par mot cle dans l'ordre
call trinome(c=c1,x1=x,a=a1,b=a1=b1,x2=y)  ! par mot cle dans le desordre
call trinome(a1,a1+b1,c1,x2=y,x1=x)        ! mixage
```

Comme on le voit, à partir du moment où l'on nomme les arguments, il n'est plus nécessaire de respecter l'ordre.

6.8.5 Arguments optionnels

Dans certains cas, on n'a pas besoin de tous les arguments d'un sous–programme. L'attribut `optional` permet de déclarer certains arguments comme optionnels. Leur présence lors de l'appel sera testée grâce à la fonction intrinsèque `present`.

Exemple 6.3 Soit à écrire une procédure `maxmin` qui cherche l'élément minimum et maximum d'un vecteur passé en argument. En option, on pourra avoir en sortie l'indice de l'élément minimum ou maximum. Nous avons choisi le module avec procédure comme interface explicite.

```
module mmaxmin
  contains
  subroutine maxmin(v,vmax,vmin,imax,imin)
    real, dimension(:),intent(in) :: v           ! vecteur a analyser
    real, intent(out)             :: vmax,vmin   ! elt. max. et min. dans v
    real, optional, intent(out)   :: imax,imin   ! arguments optionnels

    real, dimension(1)            :: rg          ! rg de taille 1!!!

    ! pour la recherche, on utilise les fct. intrinseques maxval et minval

    vmax=maxval(v)           ! maxval -> elt. max. dans v (intrinseque)
    vmin=minval(v)           ! minval -> elt. min. dans v (intrinseque)

    ! test de presence des arguments optionnels
    ! on utlise les fct. intrinseques maxloc et minloc
```

```
    if (present(imax)) then
       rg=maxloc(v)          ! maxloc -> indice elt. max. dans v (intrinseque)
       imax=rg(1)
    endif
    if (present(imin)) then
       rg=minloc(v)          ! minloc -> indice elt. min. dans v (intrinseque)
       imin=rg(1)
    endif
  end subroutine maxmin
end module mmaxmin
```

Les fonctions `maxloc` et `minloc` fournissent les éléments extrémaux du tableau passé en argument, voir 7.4.2. Elles renvoie un vecteur dont la taille est le rang du tableau passé en argument. Ici le tableau passé en argument est de rang 1 (vecteur) donc le résultat de `maxloc` et `minloc` sera un vecteur de longueur 1 et non un scalaire.

Voici un programme qui utilise le module `mmaxmin`.

```
program PROGMINMAX
  use mmaxmin       ! acces au module de maxmin
  implicit none
  integer, parameter    :: nmax=5
  real, dimension(nmax) :: v=(/ 1.,2.,9.,4,-8. /)
  real                  :: vmin,vmax
  integer               :: rgmin,rgmax

  !----- appel avec tous les arguments ---------
  call maxmin(v,vmax,vmin,rgmax,rgmin)
  !----- appel sans les arguments optionnels ---
  call maxmin(v,vmax,vmin)
  !----- appel sans rgmin ----------------------
  call maxmin(v,vmax,vmin,rgmax)
  !----- appel sans rgmax (mot cle indispensable)
  call maxmin(v,vmax,vmin,imin=rgmin)

end program PROGMINMAX
```

Dans le dernier appel, l'argument `imin` est utlisé avec mot clé car il est impossible d'omettre un argument en cours de liste comme dans

```
call maxmin(v,vmax,vmin,,rgmin)   ! INTERDIT
```

6.8.6 Partages de données

En Fortran 77, il existe un autre moyen de communication entre unités de compilation autre que la correspondance par arguments. Ce moyen consiste à définir une zone mémoire commune grâce à la directive `common`. Cette zone est accessible en consultation et en modification. Pour qu'elle soit accessible, sa déclaration doit être recopiée dans les unités de compilation comme pour le bloc interface, avec tout ce que cela comporte comme risques. De plus les variables déclarées dans cette zone ne peuvent être initialisées que dans un bloc spécial appelé `block data`.

En Fortran 90, le partage de données entre unités de programme se fait en plaçant les variables à partager dans un module. Pour y accéder depuis une unité de compilation, il suffit

d'utiliser `use`. L'attribut `save` sera nécessaire si la variable à exporter n'est pas initialisée (pour qu'elle devienne statique).

Exemple 6.4 Voici un exemple de module de partage de données.

```
module mdonnees
  implicit none
  integer, parameter :: nmax=10             ! variable statique
  integer, dimension(nmax), save :: coeff   ! save obligatoire
end module mdonnees


subroutine impdonnees
  use mdonnees
  integer  :: i

  print '("Nb elements :",I4)',nmax
  print '("Coeff=",10I3)',(coeff(i),i=1,nmax)
end subroutine impdonnees


program PARTGDONNEES
  use mdonnees        ! acces au module mdonnees
  integer    :: i

  coeff=(/(i,i=1,nmax)/)  ! initialisation du tab. coeff

  call impdonnees

end program PARTGDONNEES
```

Après compilation séparée et édition de liens, l'exécution donne :

```
Nb elements :  10
Coeff=  1  2  3  4  5  6  7  8  9 10
```

Chapitre 7

Fonctions intrinsèques

Le Fortran 77 était déjà riche en fonctions intrinsèques (surtout mathématiques). Le Fortran 90 hérite encore d'une foule de nouvelles fonctions prédéfinies. Comme dans MATLAB, les fonctions mathématiques usuellement appliquées à des scalaires deviennent applicables, en Fortran 90, à des tableaux et renvoient un tableau. Ainsi, l'instruction

```
y(1:n)=sin(x(1:n))
```

est équivalente à la boucle

```
do i=1,n
   y(i)=sin(x(i))
end do
```

Les fonctions intrinsèques ayant toujours une interface explicite, les noms des arguments donnés dans tout le chapitre sont significatifs. Ils peuvent donc être utilisés dans les appels avec mot clé comme

```
sin(x=1.276543)
mod(a=5,p=2)
```

7.1 Quelques fonctions numériques intrinsèques

Voici quelques fonctions numériques usuelles. Le résultat est du sous–type du ou des arguments sauf pour `abs(a)` quand `a` est complexe.

`abs(a)`	valeur absolue. L'argument peut être `integer`, `real` ou `complex`. Dans ce dernier cas le résultat est un réel, le module du nombre complexe argument.
`aimag(z)`	partie imaginaire de `z`. Le résultat est de type `real` avec la variante de `z`
`cmplx(x,y)`	conversion en nombre complexe, convertit le couple de réels (x, y) en $x + iy$.
`conjg(z)`	fournit le complexe conjugué du complexe `z`.
`dble(x)`	conversion en double précision.
`int(a)`	partie entière du réel `a`.
`min(a1,a2,...)`	valeur minimale des arguments.
`max(a1,a2,...)`	valeur maximale des arguments.
`mod(a,p)`	reste de la division de `a` par `p`.
`real(a)`	conversion en réel. Si `a` est de type `complex`, le résultat est la partie réelle.

7.2 Quelques fonctions mathématiques intrinsèques

Leur argument s'appelle toujours `x`. Voici quelques unes.

`acos(x), asin(x)`	arccos$x$ et arcsin$x$, avec $\|x\| \leq 1$.
`atan(x)`	arctan$x$
`cos(x), sin(x), tan(x)`	cos$x$, sin$x$ et tan$x$.
`cosh(x), sinh(x), tanh(x)`	cosh$x$, sinh$x$ et tanh$x$.
`exp(x), log(x), log10(x)`	e^x, $\ln x$ et $\log x$.
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$.

7.3 Quelques fonctions de précision

`range(x)`	pour le sous–type de l'argument entier ou réel `x` fourni, retourne la valeur entière maximale de l'exposant r tel que : • $\|x\| < 10^r$ soit représentable, si x est entier • $10^{-r} < \|x\| < 10^r$ soit représentable, si x est réel.
`precision(x)`	retourne le nombre maximal de chiffres décimaux significatifs pour le sous–type de l'argument réel fourni.
`epsilon(x)`	retourne la quantité considérée comme négligeable devant 1, pour le sous–type réel foruni.
`tiny(x)`	retourne la plus petite valeur réelle représentable dans le sous–type de `x` (i.e. limite d'*underflow*).
`huge(x)`	retourne la plus grande valeur réelle représentable dans le sous–type de `x` (i.e. limite d'*overflow*).

7.4 Quelques fonctions relatives aux tableaux

Comme déjà signalé plus haut, toutes les fonctions prédéfinies élémentaires classiques sont applicables aux tableaux et renvoient un tableau. Nous ne présentons ici que les fonctions spécifiques aux tableaux. Les arguments optionnels seront représentés entre `[]`.

7.4.1 Interrogation sur le profil

`shape(source)`	retourne le profil du tableau `source` passé en argument. Le résultat est un vecteur de taille le rang du tableau argument.
`size(array [,dim])`	retourne la taille ou l'étendue de la dimension indiquée via `dim` du tableau `array` passé en argument.
`lbound(array [,dim])`	retourne, en l'absence de `dim`, les bornes inférieures du tableau `array`. Lorsque `dim` est présent, renvoie la borne inférieure de la dimension spécifiée.
`ubound(array [,dim])`	comme `lbound` mais renvoie les bornes supérieures.

Soit la déclaration suivante

```
integer, dimension(-2:27,0:49) :: T
```

Voici le résultat de l'application des fonctions ci–dessus au tableau `T`.

`shape(T)`	vaut	`(/30,50/)`
`size(T)`	vaut	`1500`
`size(T,dim=1)`	vaut	`30`
`ubound(T)`	vaut	`(/27,49/)`
`ubound(T,dim=2)`	vaut	`49`
`lbound(T)`	vaut	`(/-2,0/)`
`lbound(T,dim=1)`	vaut	`-2`

7.4.2 Interrogation sur le contenu

Plus la peine d'écrire et réécrire des boucles pour connaître où se trouve le plus grand ou le plus petit élément d'un tableau. Les fonctions

```
minloc(array [,mask])
maxloc(array [,mask])
```

fournissent un vecteur d'entiers (de taille égale au rang de `array`), dont les valeurs repèrent un élément respectivement minimal et maximal. `mask`, s'il est présent, doit être un tableau logique conformant avec le tableau `array`. En pratique, `mask` est un «filtre» sous forme d'expression logique de sorte que seuls sont pris en compte les éléments de `array` associés à une valeur `.true.` de `mask`.

Soit le tableau d'entier suivant

```
integer, dimension(5) :: v=(/ 2,-1,10,3,-1 /)
```

Alors on a

`minloc(v)`	vaut	`(/ 2 /)`,	i.e. `v(2)` élément minimal
`maxloc(v)`	vaut	`(/ 3 /)`,	i.e. `v(3)` élément maximal

Les tableaux renvoyés par `minloc` `maxloc` dans le cas ci–dessus sont des vecteurs de taille 1, i.e. le rang de `v`.

Soit maintenant la matrice `a` déclarée comme suit

```
integer, dimension(0:2,-1:2) :: a
```

dont le contenu est

$$\begin{pmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{pmatrix}$$

Alors on a

`maxloc(a,mask=a<5)`	vaut	`(/ 2,2 /)`,	i.e. `a(2,2)` est le max des $a_{ij} < 5$
`minloc(a,mask=a>5)`	vaut	`(/ 3,3 /)`,	i.e. `a(3,3)` est le min des $a_{ij} > 5$

7.4.3 Fonctions de réduction

Ces fonctions sont appelées ainsi parce que, appliquées à un tableau de rang n, le résultat est soit scalaire soit de rang $n-1$.

On veut juste connaître la valeur minimale ou maximale dans un tableau ? Pas de problème ! Les fonctions

```
minval(array [,dim][,mask])
maxval(array [,dim][,mask])
```

fournissent les éléments extrémaux du tableau `array`. Si `mask` est présent, il sert de «filtre» comme pour `minloc` et `maxloc`. Si `dim` est présent, la recherche se fait sur toutes les sections de `array` qu'on peut obtenir en fixant tous les indices sauf celui relatif à la dimension spécifiée par `dim`. C'est un peu compliqué mais voyant un exemple. Soit le tableau `A` dont le contenu est

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

`minval(A)`	vaut	`1`	(retourne le plus petit élément de `A`)
`maxval(A)`	vaut	`6`	(retourne le plus grand élément de `A`)
`minval(A,dim=1)`	vaut	`(/ 1,3,5 /)`	(recherche par colonne)
`minval(A,dim=2)`	vaut	`(/ 1,2 /)`	(recherche par ligne)
`maxval(A,dim=1)`	vaut	`(/ 2,4,6 /)`	(recherche par colonne)
`maxval(A,dim=2)`	vaut	`(/ 5,6 /)`	(recherche par ligne)
`minval(A,dim=1,mask=A>1)`	vaut	`(/ 2,3,5 /)`	
`maxval(A,dim=2,mask=A<3)`	vaut	`(/ 1,2 /)`	

Voici deux autres fonctions de réduction importantes qui fournissent, respectivement, la somme et le produit des éléments d'un tableau.

```
sum(array [,dim][,mask])
product(array [,dim][,mask])
```

Le sous type résultat est celui de `array`. Les arguments optionnels `dim` et `mask` fonctionnement de la même manière qu'avec `minval` et `maxval`.

Remarque 7.1 *Si* `array` *est vide ou si* `mask=.false.` `sum` *retourne* `0` *et* `product` *retourne* `1`. ◇

Soit `x=(/ 2,5,-6 /)`. Alors on a

`sum(x)`	vaut	`1`
`product(x)`	vaut	`-60`

Appliquons les fonctions `sum` et `product` au tableau `A` utilisé pour `minval` et `maxval`.

`sum(A,dim=1)`	vaut	`(/ 3,7,11 /)`, somme sur les colonnes
`sum(A,dim=2,mask=A<2)`	vaut	`(/ 1,0 /)`, somme sur les lignes, à cause de `mask=A<2`, la 2ème ligne est vide donc `sum` retourne 0.
`product(A,dim=2)`	vaut	`(/ 15,48 /)`
`product(A,dim=1,mask=A>4)`	vaut	`(/ 1,1,30 /)`, à cause de `mask=A>4`, les 2 premières colonnes sont vides donc `product` retourne 1.

7.4.4 Fonctions de multiplications

En Fortran 90, il existe deux fonctions de multiplication

`dot_product(vector_a,vector_b)`	retourne le produit scalaire de deux vecteurs passés en argument.
`matmul(matrix_a,matrix_b)`	effectue le produit de deux matrices ou d'une matrice et d'un vecteur. Les arguments doivent respecter les contraintes usuelles sur le produit matriciel.

Soit les vecteurs `v1=(/ 2,-3,-1 /)` et `v2=(/ 6,3,3 /)`. Alors on a

`dot_product(v1,v2)` vaut `0`

`dot_product(v1(1 :2),v2(1 :2))` vaut `3`

Soit le vecteur `v=(/ 2,-4,1 /)` et la matrice `A` dont le contenu est

$$\begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$$

Alors on a

`matmul(A,v)` vaut `(/ 29,-7,10 /)`

`matmul(A(1 :2,1 :2),v(2 :3)` vaut `(/ -18,5 /)`

7.4.5 Fonctions de transformations

Il existe plusieurs fonctions de transformations en Fortran 90 mais nous ne donnons ici que celle qui nous semble la plus utile pour le cours de Programmation Numérique, à savoir la fonction

```
transpose(array)
```

qui fournit la transposée de la matrice (i.e. tableau de rang 2) passée en argument. Donc si `array` est de profil `(/m,n/)`, le tableau resultat sera de profil `(/n,m/)`.

Chapitre 8

Fichiers

Un fichier est une liste d'enregistrements stockés en mémoire auxiliaire. La taille du fichier est le nombre d'enregistrements. Pour un fichier texte (i.e. lisible par le programmeur) chaque ligne est un enregistrement. Pour un fichier binaire, un enregistrement est une liste d'entrée/sortie.

8.1 Ouverture et fermeture : `open/close`

En Fortran, un fichier est identifié par un entier naturel appelé numéro d'unité logique. L'association fichier externe/ numéro d'unité logique se fait à l'ouverture du fichier par l'ordre `open` comme suit (avec les spécifications les plus utiles seulement)

```
open([unit=]unite_logique, file=nomfichier, iostat=ierr, status=etat, &
     access=methode, action=mode, recl=long)
```

avec

- `unit=unite_logique` spécifie l'unité logique attachée au fichier. Ce nombre doit être libre, i.e. ne pas être attaché à un autre fichier.
- `file=nomfichier` donne le nom de fichier à associer au numéro d'unité logique
- `iostat=ierr` récupère l'erreur d'ouverture. Fonctionne comme `stat` de `allocate`. Tout s'est bien passé si à la sortie `ierr==0`. Si `ierr >0` le fichier n'a pu être ouvert.
- `status=etat` spécifie le statut du fichier à ouvrir. `etat` peut prendre l'une des valeurs suivantes
  - `'old'` le fichier existe (erreur s'il n'existe pas)
  - `'new'` le fichier n'existe pas (erreur s'il existe)
  - `'replace'` l'ancien fichier sera détruit
  - `'scratch'` fichier temporaire, sera détruit à la fermeture
- `access=methode` spécifie la méthode d'accès au fichier. `methode` peut prendre l'une des valeurs suivantes
  - `'sequential'` fichier à accès séquentiel (ligne par ligne). C'est la méthode par défaut.
  - `'direct'` fichier à accès direct. La longueur de chaque enregistrement `recl` doit être connue.
- `action=mode` spécifie ce qui peut être fait du fichier à ouvrir. `mode` peut prendre l'une des valeurs suivantes
  - `'read'` ouverture en lecture seulement
  - `'write'` ouverture en écriture seulement

- `'readwrite'` ouverture en lecture/écriture. C'est le mode d'ouverture par défaut.
- `recl=long` spécifie la longueur d'un enregistrement pour un fichier à accès direct.

Voici un exemple d'ouverture de fichier avec test d'erreur d'ouverture.

```
open(1,file='output.dat',iostat=ierr,status='replace',&
       access='sequential',action='write')
if (ierr /= 0) then
  print *,'Impossible de creer ''output.dat'' '
  stop
endif
```

La fermeture d'un fichier se fait simplement par

```
close(unite_logique)
```

8.2 Lecture et écriture : `read/write`

La syntaxe de l'ordre de lecture dans un fichier est la suivante (avec les spécifications les plus utiles)

```
read(unite_logique,[fmt=]format, iostat=ierr) liste
```

avec

- `unite_logique` un numéro d'unité logique valide ou `*` pour spécifier l'unité de lecture standard (i.e. le clavier).
- `[fmt=]format` fournit la spécification de format pour les données à lire. En général il faut éviter les formats en lecture sauf quand le fichier à lire est formaté.
- `iostat=ierr` récupère (dans la variable entière `ierr`) l'erreur de lecture. Si `ierr==0`, tout s'est bien passé. Si `ierr<0` c'est la fin du fichier à lire. Si `ierr>0`, il y a eu erreur de lecture (enregistrement insuffisant, fichier devenu inaccessible, ...)

L'ordre d'écriture `write` a la même syntaxe

```
write(unite_logique,[fmt=]format, iostat=ierr) liste
```

Les spécifications ont la même signification que dans `read`. Si `unite_logique` vaut `*`, c'est la sortie standard (i.e. l'écran) qui est utilisé.

Remarque 8.1 *L'unité logique standard d'entrée/sortie* `*` *(i.e. le clavier ou l'écran) s'utilise directement sans passer par* `open`.

Exemple 8.1 (Création d'un fichier contenant la matrice de Hilbert) Le programme suivant crée un fichier `hilbertmat.dat` contenant la matrice de Hilbert d'ordre n. La prémière ligne du fichier contient l'ordre de la matrice puis les suivantes les lignes de la matrices.

```
program HILBERTMAT
  implicit none
  character(len=*), parameter          :: fmt='(10F12.8)'
  real(8), dimension(:,:), allocatable :: a
  integer                              :: n,i,j

  ! Lecture de la dimension n
```

```
   ! le facteur de repetition est 10
   ! donc pas plus de 10 nombres sur une meme ligne
   do
      print *,'Entrer la taille de la matrice n<=10 : '
      read *,n
      if (n<=10) exit
   end do

   allocate(a(n,n))                          ! allocation de a

   do i=1,n
      a(i,:)=(/ (1.d0/dble(i+j-1),j=1,n) /)  ! remplissage de a ligne par ligne
   end do

   ! ouverture du fichier hilbertmat.dat en ecriture
   open(1,file='hilbertmat.dat',status='replace',action='write')

   write(1,'(I4)') n                         ! ecriture de la dimension de a
   do i=1,n
      write(1,fmt) (a(i,j),j=1,n)            ! ecriture des ligne de a
   end do

   ! fermeture du fichier hilbertmat.dat
   close(1)
 end program HILBERTMAT
```

Après exécution, avec $n = 5$, on trouve dans le fichier `hilbertmat.dat` :

```
 5
 1.00000000  0.50000000  0.33333333  0.25000000  0.20000000
 0.50000000  0.33333333  0.25000000  0.20000000  0.16666667
 0.33333333  0.25000000  0.20000000  0.16666667  0.14285714
 0.25000000  0.20000000  0.16666667  0.14285714  0.12500000
 0.20000000  0.16666667  0.14285714  0.12500000  0.11111111
```

Pour lire le fichier `hilbertmat.dat`, il suffit remplacer `write` par `read`.

```
   ! ouverture du fichier hilbertmat.dat en lecture
   open(1,file='hilbertmat.dat',status='old',action='read')
   read(1,*) n                               ! lecture de la dimension de a
   do i=1,n
      read(1,*) (a(i,j),j=1,n)               ! lecture des ligne de a
   end do
```

Les formats sont déconseillés en lecture pour permettre la lecture des nombres tels quels avec les blancs comme séparateurs.

Annexe A

Factorisation LU et Applications

A.1 La méthode

Soit à resoudre le système linéaire

$$A \cdot x = b, \tag{A.1}$$

où $A = (a_{ij})$ est une matrice $n \times n$ et b un vecteur de $\mathbb{R}^n$. Supposons qu'on puisse mettre A sous la forme

$$A = L \cdot U, \tag{A.2}$$

où L est une matrice triangulaire inférieure (*lower* en anglais) et U une matrice triangulaire supérieure (*upper* en anglais). Pour la suite, on pose

$$L = \begin{bmatrix} \ell_{11} & 0 & \cdots & 0 \\ \ell_{21} & \ell_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

A l'aide de la décomposition (A.2), le système (A.1) devient

$$(L \cdot U) \cdot x = L \cdot (U \cdot x) = b.$$

Le système (A.1) peut alors être résolu en deux étapes.

1. On résoud d'abord le système intermédiaire en y

$$L \cdot y = b. \tag{A.3}$$

2. On résoud ensuite le système final en x

$$U \cdot x = y. \tag{A.4}$$

Comme les matrices L et U sont triangulaires, les systèmes (A.3)-(A.4) sont résolus par substitution. La solution de (A.3) est

$$y_1 = \frac{b_1}{\ell_{11}}, \tag{A.5}$$

$$y_i = \frac{1}{\ell_{ii}} \left[b_i - \sum_{j=1}^{i-1} \ell_{ij} y_j \right], \quad i = 2, 3, \ldots, n \tag{A.6}$$

et celle de (A.4)

$$x_n = \frac{y_n}{u_{nn}}, \tag{A.7}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right], \quad i = n-1, n-2, \ldots, 1. \tag{A.8}$$

A.2 L'algorithme de décomposition

L'algorithme de décomposition LU d'une matrice est le suivant (voir cours d'Analyse Numérique pour les détails).

#0. $\ell_{ii} = 1$, $i = 1, 2, \ldots, n$.

#1. Pour chaque $j = 1, 2, \ldots, n$ faire

#1.1 Pour $i = 1, 2, \ldots, j$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \tag{A.9}$$

#1.2 Pour $i = j+1, \ldots, n$

$$\ell_{ij} = \frac{1}{u_{jj}} \left[a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} u_{kj} \right] \tag{A.10}$$

L'élément u_{jj} de la formule (A.10) est le pivot. Pour des raisons de stabilité numérique, $|u_{jj}|$ ne doit pas être trop petit. Une manière simple pour augmenter la stabilité de la méthode est la recherche d'un pivot partiel. Le candidat au rôle de pivot est donné par

$$\max_{j \le i \le n} |u_{ij}|.$$

Le terme de pivot partiel vient du fait que la recherche n'est effectuée que sur une partie de la colonne. Si le candidat au rôle de pivot est sur une ligne $i_0 \neq j$, il faut permuter les lignes i_0 et j en prenant soins d'enregistrer les lignes permutées et le nombre de permutations. Dans la section suivante on verra pourquoi enregistrer les permutations.

A.3 Les applications

Une fois obtenue la décomposition LU de A, le système (A.1) peut être résolu avec différents second membres. On se limite alors aux substitutions (A.5)-(A.8). Les permutations éventuelles des lignes, soigneusement enregistrées lors de la décomposition, sont alors appliquées aux différents second membres.

On peut aussi calculer le déterminant de A à partir de sa décomposition LU. On a

$$\det A = (-1)^p \prod_{i=1}^{n} u_{ii},$$

où p est le nombre de permutations effectuées lors de la décomposition.

Pour calculer l'inverse de A on doit resoudre n systèmes linéaires auxiliaires. Soit e_i, le i–ème vecteur de la base canonique de $\mathbb{R}^n$ et $\mathbb{I}_n$ la matrice unité d'ordre n. On a que $\mathbb{I} = (e_1\ e_2\ \cdots e_n)$.

La matrice inverse de A est donc construite colonne après colonne, en resolvant les n systèmes linéaires en $\tilde{x}_i$

$$A \cdot \tilde{x}_i = e_i, \quad i = 1, 2, \ldots, n. \tag{A.11}$$

On a donc, à la fin des calculs, $A^{-1} = (\tilde{x}_1\ \tilde{x}_2\ \cdots\ \tilde{x}_n)$.

On remarquera au passage qu'il est plus coûteux d'inverser une matrice que de résoudre un système linéaire.

A.4 Les problèmes de stockage

On a pas besoin de stocker les matrices L et U séparement. Les formules (A.5)-(A.8) sont telles que les matrices L et U peuvent être stockées sous la forme

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ \ell_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ \ell_{21} & \ell_{32} & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & u_{nn} \end{bmatrix}.$$

La diagonale de L n'est pas stockée, puisqu'elle ne contient que des 1 (voir cours d'Analyse Num.).

Pour $i = j$, la formule (A.9) est parfaitement identique à (A.10), à un facteur multiplicatif près. En effet, pour $i = j$, la formule (A.9) devient

$$u_{jj} = a_{jj} - \sum_{k=1}^{j-1} \ell_{jk} u_{kj}.$$

Si on pose

$$\tilde{\ell}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} u_{kj},$$

on voit qu'on peut d'abord calculer tous les termes $u_{jj}, \tilde{\ell}_{j+1,j}, \ldots, \tilde{\ell}_{nj}$; pour rechercher le pivot. Comme $\ell_{ij} = \tilde{\ell}_{ij}/u_{ij}$, pour $i = j+1, \ldots, n$, il suffit ensuite de diviser les termes $\tilde{\ell}_{j+1,j}, \ldots, \tilde{\ell}_{nj}$ par le pivot.

Et toutes ces opérations peuvent s'effectuer dans la même matrice A du système de départ (A.1). Dans ce cas à la fin des calculs, la matrice A est remplacée par sa décomposition LU.

A noter que le stockage de matrices en Fortran permet de construire A^{-1} dans (A.11) sans utilisation de vecteurs auxiliaires.

A.5 Le code Fortran à écrire

Le but du TP est d'écrire un code Fortran comprenant un programme principal et un module LU dans un fichier séparé. Le programme principal doit pouvoir, grâce au module LU, resoudre un système linéaire, calculer le déterminant d'une matrice ou inverser une matrice. Pour cela le module LU doit avoir en entrée un indice de choix `ichoix` tel que

`ichoix=1` Le module effectue la décomposition LU et resoud le système $Ax = b$.

`ichoix=2` Le module effectue la décomposition LU seulement.

`ichoix=3` Le module resoud $Ax = b$ par substitution. On suppose donc que la matrice a deja été décomposée.

Le module doit aussi avoir un indice d'erreur `ierr` en sortie :

`ierr=0` La décomposition s'est effectuée normalement

`ierr=1` La décomposition a echouée parce que la matrice A est singulière, i.e. un pivot nul a été trouvé.

On pourra utiliser les problèmes test suivants

$$A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} 4 \\ 4 \\ 4 \\ 4 \end{bmatrix}, \quad x^* = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad x^* = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}.$$

Pour la robustesse, on peut inverser la matrice de Hilbert $H = (h_{ij})_{i,j=1,\ldots,n}$ avec

$$h_{ij} = \frac{1}{i+j-1}, \quad \forall i, j = 1, \ldots, n.$$

Il faut prendre $n = 50, 100, 250, 500, \ldots$

Annexe B

Méthodes Itératives de Résolution de Systèmes Linéaires

B.1 Principe

On souhaite résoudre le système linéaire

$$Ax = b \tag{B.1}$$

où x et b sont des vecteurs de $\mathbb{R}^n$ et $A = (a_{ij})$ est une matrice $n \times n$ de rang plein. Soit x^* l'unique solution de (B.1). Lorsque n est très grand ou lorsque le matrice A est creuse, les méthodes directes (Gauss, décomposition,...) deviennent moins compétitives à cause du coût de calcul et des erreurs d'arrondis..

Notons que (B.1) est équivalent à $Ax - b = 0$. Soit x^0 un vecteur quelconque de $\mathbb{R}^n$. Formons $r^0 = Ax^0 - b$, vecteur de $\mathbb{R}^n$ appelé résidu du système (B.1).

Le principe des itérations est le suivant : on part d'un vecteur x^0 et l'on forme r^0 puis on construit, à partir de x^0, x^1 et l'on a le résidu correspondant r^1. Le processus se poursuit jusqu'à ce qu'un critère d'arrêt soit vérifié. On obtient ainsi une suite $\{x^k\}$ de vecteurs de $\mathbb{R}^n$.

Le principe des méthodes itératives est donc la construction d'une suite $\{x^k\}_{k\geq 0}$ de vecteurs de $\mathbb{R}^n$, définie par la relation

$$\begin{aligned} &x^0 \in \mathbb{R}^n, \\ &x^{k+1} = H(x^k) \end{aligned}$$

avec $H : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ choisie pour que $x = H(x) \Leftrightarrow Ax = b$.

B.2 Convergence des méthodes itératives

Les méthodes itératives étudiées dans ce TP correspondent à une méthode de décomposition de la matrice A de (B.1) sous la forme $A = M - N$, où M est supposée non singulière. Le système (B.1) peut alors s'écrire

$$Mx = Nx + b,$$

i.e. comme le problème de point fixe (quand M est inversible !!)

$$x = (M^{-1}N)x + M^{-1}b. \tag{B.2}$$

Le problème de point fixe (B.2) introduit un schéma récursif, à savoir

$$x^{k+1} = H(x^k) \tag{B.3}$$

avec $H : x \mapsto (M^{-1}N)x + M^{-1}b$. Pour x^* la solution de (B.1), on a

$$x^{k+1} - x^* = (M^{-1}N)^{k+1}(x^* - x^0).$$

On doit donc étudier la convergence de la suite de matrices $\{(M^{-1}N)^k\}_{k\geq 0}$.

Théorème 1 *Les trois propositions suivantes sont équivalentes.*

(i) *La méthode (B.3) est convergente*

(ii) $\rho(M^{-1}N) < 1$, *i.e. le rayon spectral*[1] *de la matrice* $M^{-1}N$ *est strictement inférieur à* 1.

(iii) $\| M^{-1}N \| < 1$, *pour au moins une norme matricielle subordonnée.*

B.3 Schémas itératifs particuliers

On présente maintenant 3 classes de schémas reposant sur des décompositions de A construites à partir des éléments suivants : D, E et F définis comme sur la figure B.1.

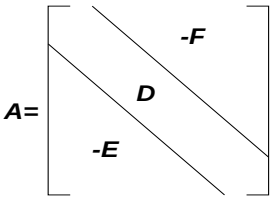

FIG. B.1 – Décomposition générale de la matrice A

$D = \text{diag}\{a_{11}, \ldots, a_{nn}\}$ est la diagonale de A. Les matrices $-E$ et $-F$ sont les parties de A triangulaires respectivement au dessous et au dessus de la diagonale. On a donc $A = D - E - F$.

B.3.1 La méthode de Jacobi

On pose $M = D$, $N = E + F$. Le schéma itératif est donc

$$Dx^{k+1} = (E + F)x^k + b.$$

En explicitant, on trouve

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^k - \sum_{j=i+1}^{n} a_{ij}x_j^k \right], \quad i = 1, 2, \ldots, n. \tag{B.4}$$

En pratique, on applique la méthode de Jacobi essentiellement aux matrices à diagonale dominante, i.e. vérifiant

$$|a_{ii}| \geq \sum_{k\neq i} |a_{ik}|$$

[1] C'est la plus grande, en module, des valeurs propres d'une matrice

B.3.2 La méthode de Gauss–Seidel

On pose $M = D - E$, $N = F$. Le schéma itératif est

$$(D - E)x^{k+1} = Fx^k + b.$$

En explicitant, on trouve

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right], \quad i = 1, 2, \ldots, n. \tag{B.5}$$

Remarques (i) La méthode de Jacobi requiert le stockage simultané de x^k et x^{k+1}, ce qui peut être gênant pour des très gros systèmes. Néanmoins elle présente l'avantage d'être fortement parallélisable.

(ii) La méthode de Gauss–Seidel ne nécessite que le stockage d'un seul vecteur. Le gros inconvénient est qu'elle est trop séquentielle.

B.3.3 Méthode de relaxation

On peut accélérer les méthodes précédentes en introduisant un *coefficient de relaxation* ω. On obtient la méthode dite de relaxation. En fait on utilisera cette idée surtout avec la méthode de Gauss–Seidel. La décomposition correspondante est

$$M = \frac{D}{\omega} - E, \quad N = \frac{1-\omega}{\omega} D + F.$$

$\omega < 1$ on parle de *sous–relaxation*

$\omega > 1$ on parle de *sur–relaxation*

$\omega = 1$ c'est la méthode de Gauss–Seidel

Le schéma itératif est

$$\left(\frac{1}{\omega} D - E \right) x^{k+1} = \left(\frac{1-\omega}{\omega} D + F \right) x^k + b.$$

En développant, on trouve

$$x_i^{k+1} = (1-\omega) x_i^k + \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right]$$

On montre que pour $0 < \omega < 2$, la méthode de relaxation converge. La grande difficulté dans la mise en œuvre des méthodes de relaxation est le calibrage du coefficient ω.

B.4 Le critère d'arrêt

Les critères d'arrêt les plus utilisés reposent sur l'utilisation des normes vectorielles et matricielles. En effet, on stoppe la méthode dès que

$$\frac{\| x^{k+1} - x^k \|}{\| x^{k+1} \|} \leq \varepsilon \tag{B.6}$$

ou

$$\frac{\| Ax^k - b \|}{\| b \|} \leq \varepsilon. \tag{B.7}$$

Le choix de la norme est à la discrétion du programmeur. Pour mémoire les normes les plus utilisées dans les cas vectoriel sont

$$\| v \|_1 = \sum_{i=1}^{n} |v_i|$$

$$\| v \|_2 = \left[\sum_{i=1}^{n} (v_i)^2 \right]^{1/2}$$

$$\| v \|_\infty = \max_{i=1..n} |v_i|$$

B.5 Le TP

Ecrire un programme Fortran permettant la résolution d'un système linéaire saisi par fichier par l'une des méthodes suivantes :

- Jacobi
- Gauss–Seidel
- Relaxation.

Une précision de convergence ε et un point de départ x^0 étant donnés, le programme fournira :

- la solution trouvée
- le nombre d'itérations

Pour la relaxation, on pourra étudier l'influence de ω en complétant le tableau suivant

ω	.1	.2	.3	.4	.5	.6	.7	.8	.9	1.0	1.1	1.2	1.3	1.4
Nb iter.														

ω	1.5	1.6	1.7	1.8	1.9
Nb iter.					

On pourra utiliser le problème test suivant

$$A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix}, \quad b = \begin{pmatrix} 4 \\ 4 \\ 4 \\ 4 \end{pmatrix}, \quad x^* = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$$

Annexe C

La méthode QR

La méthode QR est l'une des méthodes les plus utilisées pour le calcul de l'ensemble des valeurs propres d'une matrices quelconque, notamment symétrique. Nous présentons une forme simplifiée pour les besoins du TP.

C.1 Le principe

Soit $A = A_0$ une matrice carrée quelconque ; on écrit sa factorisation QR par la méthode de Householder, soit $A_0 = Q_0 R_0$. On forme alors A_1 en commutant Q_0 et R_0, soit $A_1 = R_0 Q_0$. On factorise ensuite A_1, soit $A_1 = Q_1 R_1$, et ainsi de suite. On obtient une suite de matrices $\{A_k\}$ qui sont toutes semblables à A puisque

$$A_1 = R_0 Q_0 = Q_0^T A Q_0$$
$$A_k = R_k Q_k = Q_k^T A_{k-1} Q_k = (Q_0 Q_1 \cdots Q_k)^T A (Q_0 Q_1 \cdots Q_k).$$

La matrice A_k est donc l'espression de A dans la base formée par les vecteurs colonnes de la matrice Q_k.

Sous certaines hypothèses, on montre que les éléments diagonaux des matrices A_k convergent vers les *valeurs propres de* A.

C.2 La factorisation QR

On appelle *matrice de Householder* une matrice de la forme

$$H_v = \mathbb{I}_n - 2\frac{vv^T}{\| v \|^2}, \tag{C.1}$$

pour un vecteur v non nul de $\mathbb{R}^n$. La matrice H_v est symétrique et orthogonale. L'intérêt des matrices de Householder, en Analyse Numérique, provient du résultat suivant.

Théorème 2 *Soit v un vecteur de $\mathbb{R}^n$ tel que $\| v \| > 0$. Il existe deux matrices de Householder H_v telles que les $(n-1)$ dernières composantes de $H_v \cdot v$ soit nulles.*

A l'aide du théorème 2, on montre que toute matrice inversible admet une factorisation QR.

L'existence des matrices Q et R pour une matrice carrée symétrique A est donc établie. Maintenant il nous faut construire la matrice de Householder utilisée pour construire Q et R. Rappelons que la matrice H doit avoir la propriéte suivante, pour $u \in \mathbb{R}^n$,

$$(Hu)_i = \begin{cases} u_i & \text{pour } i = 1, \ldots, k-1 \\ 0 & \text{pour } i = k+1, \ldots, n. \end{cases}$$

Soit k, l'indice de la colonne de la matrice A à traiter. Des considérations précédentes, la matrice de Householder (C.1) est construite à partir du vecteur

$$v^k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a_{kk} + \text{signe}(a_{kk})s \\ a_{k+1,k} \\ \vdots \\ a_{nk} \end{bmatrix}, \tag{C.2}$$

où

$$s = \left((a_{kk})^2 + \cdots + (a_{nk})^2\right)^{\frac{1}{2}}.$$

Le choix du signe dans la k–ième composante est dû au souci d'éviter les dénominateurs trop «petits» dans (C.1).

C.3 L'algorithme

La description formelle de l'algorithme QR de diagonalisation est la suivante.

Etape 0. $A_0 = A$

Etape k. Construire Q et R telles que $A_k = QR$.
Construire A_{k+1} en commutant le produit, i.e. $A_{k+1} = RQ$.
Tester le plus grand élément non diagonal de A_{k+1}.

La matrice Q_k est elle–même le produit des matrices de Householder H_{v^k} :

$$Q_k = H_{v^1} H_{v^2} \cdots H_{v^k}$$

Théorème 3 (Convergence de la méthode QR) *On suppose que la matrice A est inversible, et que ses valeurs propres λ_i, $1 \leq i \leq n$, sont toutes de modules différents. Alors la suite de matrices $\{A_k\}$ est telle que*

$$\lim_{k \to +\infty} (A_k)_{ii} = \lambda_i, \quad 1 \leq i \leq n,$$
$$\lim_{k \to +\infty} (A_k)_{ij} = 0, \quad 1 \leq j < i \leq n.$$

La matrice limite des A_k n'est donc pas diagonale pour une matrice A quelconque. Concrètement, le «cœur» de l'algorithme de diagonalisation QR est le suivant.

Algorithme QR

1. Initialisation $Q \leftarrow \mathbb{I}$

2. Pour $k = 1, \ldots, n-1$.
Calcul de v^k donné par (C.2). $r \leftarrow \sqrt{2s(s+|a_{kk}|)} = \| v^k \|$.
$w^k \leftarrow v^k / r$.
Calcul de H : $H_{ij} \leftarrow \delta_{ij} - 2w_i^k w_j^k$
Calcul de Q : $Q \leftarrow Q \cdot H$
Calcul de HA : $A \leftarrow H \cdot A$

3. Mise à jour de A : $A \leftarrow A \cdot Q$.

4. Test sur les éléments non diagonaux. Si le test n'est pas vérifié aller en 1.

Remarque. Par construction, $w_i^k w_j^k = 0, \quad i = 1, \ldots, k-1$ ou $j = 1, \ldots, k-1$.
Le calcul de H pourra se faire comme suit (en ayant pris soin d'initialiser H par la matrice indentité)

$$H_{ij} \leftarrow -2w_i^k w_j^k, \quad k \leq i, j \leq n$$
$$H_{ii} \leftarrow 1 + H_{ii}, \quad k \leq i \leq n.$$

C.4 Le TP

Ecrire un programme Fortran de recherche de valeurs propres d'une matrice par la méthode QR. Compte tenu du coût (en nombre d'opérations) de la méthode on se limitera aux matrices d'ordre au plus 25.

Il faut prévoir un test de secours sur le nombre d'itérations et les éléments sous–diagonaux. Il n'y a, en effet, aucune garantie sur la diagonalisation si la matrice n'est pas symétrique.

On pourra tester le programme sur les matrices suivantes.

$$A = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{pmatrix}, \quad \lambda = (1,\ 1,\ 4)$$

$$A = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix}, \quad \lambda = (4-\sqrt{2},\ 4,\ 4+\sqrt{2})$$

Annexe D

Méthodes du Gradient Conjugué de Fletcher–Reeves et Polak–Ribière

Soit f une fonction de $\mathbb{R}^n$ dans $\mathbb{R}$, de classe $\mathcal{C}^2$. On considère le problème d'optimisation sans contraintes

$$(P) \qquad \min_{x \in \mathbb{R}^n} f(x)$$

On suppose que f possède la propriété de croissance à l'infini, i.e. $f(x) \longrightarrow +\infty$ lorsque $\| x \| \rightarrow +\infty$. On sait que cela garantit l'existence d'un point stationnaire de f.

Les algorithmes du gradient conjugué de Fletcher–Reeves (1964) et de Polak–Ribière (1971), pour resoudre (P), sont des extensions directes de la méthode du gradient conjugué pour les fonctions quadratiques. Ces méthodes sont intéressantes à double titre. D'une part elles ne nécessitent, toutes les deux, que le stockage de 4 vecteurs de longueur n. D'autre part leur vitesse de convergence est très supérieure à celle des méthodes du gradient simple. Elles représentent donc un bon compromis vitesse de convergence / encombrement mémoire.

D.1 L'algorithme de Fletcher–Reeves

Initialisation x_0, $\nabla f(x_0)$, $\varepsilon > 0$ la précision.
$k \leftarrow 0$, $d_0 \leftarrow -\nabla f(x_0)$

Repéter

Recherche Linéaire : Trouver $t_k > 0$ qui minimise la fonction $\phi(t) = f(x_k + t d_k)$

Nouvelle approximation de la solution : $x_{k+1} = x_k + t_k d_k$

Nouvelle direction de descente :

$$d_{k+1} = -\nabla f(x_{k+1}) + \beta_k d_k \tag{D.1}$$

$$\beta_k = \frac{\| \nabla f(x_{k+1}) \|^2}{\| \nabla f(x_k) \|^2} \tag{D.2}$$

$k \leftarrow k + 1$

Jusqu'à $\| \nabla f(x_k) \|^2 < \varepsilon$

Comme on peut le remarquer, on ne garde en mémoire que le point courant x_k, les gradients $\nabla f(x_k)$ et $\nabla f(x_{k+1})$; et la direction de descente d_k.

D.2 La méthode de Polak–Ribière

Elle est identique à la méthode de Fletcher–Reeves en tout point sauf pour le calcul du coefficient β_k. La formule (D.2) est remplacée par

$$\beta_k = \frac{\nabla f(x_{k+1})^T \Big(\nabla f(x_{k+1}) - \nabla f(x_k) \Big)}{\| \nabla f(x_k) \|^2} \tag{D.3}$$

D.3 Remarques sur la convergence

Les deux méthodes appliquées à une fonction quadratique ont un comportement identique à celui de la méthode du gradient conjugué pour les fonctions quadratiques (voir cours), i.e. une convergence en n étapes au plus. Appliquées à une fonction non quadratique, les deux méthodes ont en général un comportement différent.

Comme la convergence en n étapes n'est plus garantie, il se peut que le nombre d'itérations k dépasse n. Dans ce cas les directions construites ne sont plus linéairement indépendantes dans $\mathbb{R}^n$. Pour garantir la convergence globale des deux algorithmes, il faut donc les réinitialiser toutes les n itérations. La réinitialisation se fait en remplaçant la formule (D.1) par

$$d_{k+1} = -\nabla f(x_{k+1})$$

toutes les n itérations.

D.4 La Recherche Linéaire

A chaque itération k on doit trouver un pas de déplacement $t_k > 0$ qui minimise la fonction $\phi(t) = f(x_k + t d_k)$. C'est la recherche linéaire ou minimisation unidimensionnelle. En fait cette recherche n'a de linéaire que le nom car le problème correspondant est fortement non linéaire. C'est la partie la plus importante des deux algorithmes car c'est là que la fonction et son gradient sont le plus souvent évalués.

Il y a plusieurs algorithmes pour déterminer t_k. Pour le TP nous avons choisi la *méthode de dichotomie* qui est l'une des plus simples. On recherche un point qui annule $\phi'(t)$ dans un intervalle approprié. Notons que la dérivée de la fonction ϕ est donnée par

$$\phi'(t) = \nabla f(x_k + t d_k)^T d_k.$$

D'abord on recherche un intervalle $[t_0,\ t_1]$ tel que

$$\phi'(t_0) < 0, \quad \phi'(t_1) > 0. \tag{D.4}$$

Comme ça on est sûr d'encadrer au moins un minimum de ϕ, puisque la fonction est décroissante à gauche et croissante à droite de l'intervalle. Comme d_k est une direction de descente, on a deja que $\phi'(0) < 0$.

Ensuite, on applique la méthode de dichotomie à l'équation

$$\phi'(t) = 0$$

dans l'intervalle $[t_0,\ t_1]$. L'algorithme de dichotomie est le suivant.

Initialisation t_0 et t_1 vérifiant (D.4)
$\varepsilon_1 > 0$ la précision des calculs.

Tant que *t_1-t_0* $> 2\varepsilon_1$
$t_m \leftarrow (t_0 + t_1)/2$
$h \leftarrow \phi'(t_m)$
Si $h < \varepsilon_1$ alors STOP : $t_k \leftarrow t$
Sinon Si $h > 0$ alors $t_1 \leftarrow (t_0 + t_1)/2$
Si $h < 0$ alors $t_0 \leftarrow (t_0 + t_1)/2$

La recherche de l'intervalle $[t_0,\ t_1]$ peut se faire à l'aide l'algorithme suivant.

Initialisation $0 < \delta_0 < 1$ fixé, $t_0 = 0$.
$t \leftarrow \delta_0$

Repéter
Si $\phi'(t) < 0$ alors $t_0 \leftarrow t$ et $t \leftarrow 2t$
Sinon Si $\phi'(t) > 0$ alors $t_1 \leftarrow t$

Jusqu'à $(\phi'(t) > 0)$

D.5 Le TP

Ecrire un code Fortran pour resoudre les problèmes de minisation sans contrainte dans $\mathbb{R}^n$ par les méthodes du gradient conjugué de Fletcher–Reeves et de Polak–Ribière. L'utilisateur devra avoir le choix de la méthode à utiliser.

La qualité d'un code d'optimisation se juge aussi par le nombre d'évaluations de la fonction et de son gradient. Pour éviter les évaluations anarchiques de la fonction et de son gradient, il faut écrire une seule procédure (subroutine) pour calculer la fonction et son gradient. Il faut veiller à ne pas évaluer la fonction plusieurs fois avec la même valeur de x.

Il faut prévoir des test d'arrêt de secours sur le nombre d'itérations et le nombre d'appels à la procédure qui calule la fonction et son gradient.

On pourra utiliser les fonctions suivantes pour la mise au point du code.

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \qquad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \qquad x^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$f(x) = (x_1 + x_2)^2 + \left(2(x_1^2 + x_2^2 - 1) - \frac{1}{3}\right)^2, \quad x_0 = \begin{pmatrix} \sqrt{7/6} \\ 0 \end{pmatrix}, \quad x^* = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ ou } x^* = \begin{pmatrix} -\sqrt{7/12} \\ \sqrt{7/12} \end{pmatrix}$$

Puis testez la robustesse des algorithmes avec

$$f(x) = \sum_{i=2}^{n} i(2x_i - x_{i-1})^2, \quad x^{(0)} = (1, 1, \ldots, 1)^T$$

pour différentes valeurs de n, $n = 10$, 50, 1000.

Bibliographie

[1] Aberti C. *Fortran 90, Initiation à partir du Fortran 77.* Studio Image (Série Informatique), 1992 (145 pages).

[2] Delannoy C. *Programmer en Fortran 90, Guide complet.* Eyrolles, 1993 (413 pages).

[3] Lignelet P. *Fortran 90 : approche par la pratique.* Studio Image (Série Informatique), 1993.

[4] Lignelet P. *Manuel complet du langage Fortran 90 et Fortran 95.* Calcul intensif et génie logiciel, coll. Mésures Physiques, Masson, 1996.

[5] Lignelet P. *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95.* Masson, 1996.

[6] Metcalf M. et Reid J. *Fortran 90 : concepts fondamentaux.* éditions AFNOR, 1993.

[7] Olagnon M. *Traitement des données numériques avec Fortran 90.* Masson, 1996.

Index