Elements de programmation Fortran

Frédéric Hourdin, LPAOSF

11 octobre 2009

Fortran est un langage de programmation pensé dans les années 50 pour le calcul scientifique. Il continue à être largement utilisé dans le domaine de la recherche scientifique pour le calcul intensif. Contrairement à des langages plus évolués, le déroulement des opérations individuelles en machine reste relativement contrôlable dans le langage, ce qui permet d'écrire des codes relativemet efficaces en termes de calcul. Le Fortran a beaucoup évolué ces dernières années avec l'apparition de la norme Fortran 90.

Les normes récentes (Fortran 95) sont beaucoup plus riches mais aussi beaucoup plus difficiles à appréhender que la norme Fortran 77. Nous présentons donc ici les bases fondamentales de Fortran en se basant principalement sur Fortran 77 mais en utilisant (implicitement ou explicitement) quelques avancées significatives provenant de la norme 90-95.

Ce cours est une introduction destinée à des applications simples sur des ordinateurs de type PC-Linux. Les exemples commençant par

```
program NON
```

sont disponibles sous la forme de fichiers **NOM.f** ou **NOM.f90** dans le répetroire

```
/home/hourdin/COURS/FORTRAN/EXEMPLES
```

sur le réseau du LPAOSF. Tous les programmes ont été testés avec le compilateur gratuit développé pour Linux dans le cadre du projet GNU : g95.

1 Les débuts

Un programme Fortran est d'abord une suite d'instructions en caractères ASCII. Le fichier contenant ces instructions doit se terminer en .f, .F (ou .f90 ou .F90 si ont suit la norme Fortran 90 pour l'écriture des fichiers).

En fortran 77, les lignes se limitent à 80 caractères. Les 6 premiers caractères ont un statut particulier :

- Une ligne correspondant à une instruction normale commence au caractère 7 (après 6 caractères blancs).

- Un caractère "c" ou "!" dans la première colonne signifie que la ligne est une ligne de commentaire.

- Un caractère quelconque dans la 6eme colonne indique que la ligne est la suite de la ligne précédente.

Les majuscules et les minuscules sont indifférenciées en Fortran.

Un programme Fortran commence par une instruction "program" et se termine par une instruction "end".

On montre ici un petit exemple

```
      program premiers_pas
c   Ceci est un petit programme d'intiation à Fortran.
      print*,'Ca marche !'
      print*,'Si la ligne est trop longue, on peut la couper en',
     s   'deux en mettant un caractère dans la 5eme case'
      end
```

La commande

```
print*,chaine_de_caracteres
```

imprime à l'écran une chaîne de caractère. Une chaîne de caractères peut être une suite de caractères entourés par des '.

Après avoir écrit l'ensemble de ces lignes dans un fichier

```
test.f
```

par exemple, **on doit compiler le programme** , c'est à dire le transformer dans un langage directement compréhensible par le cœur de l'ordinateur, le processeur. Cette compilation s'effectue en tappant, sur une fenêtre "shell", l'instruction

```
g95 test.f
```

qui va créer un **executable a.out** ou bien

```
g95 test.f -o test
```

qui renommera l'executable **test au lieu de a.out**. On lance un executable en tappant simplement

```
a.out
```

ou "test" dans la fenêtre shell.

En fortran 90, on peut écrire dés le premier caractère. Les commentaires commencent par un !. Les lignes peuvent dépasser les 80 caractères.

Pour l'essentiel, il existe une compatibilité amont entre les deux versions de Fortran, c'est à dire qu'un code écrit en fortran 77 peut être compilé en fortran 90/95. Ceci ne concerne pas cependant le format d'écriture décrit ci-dessus. Si on suit le format fortran 77, on nommera le fichier program.f. Si on suit le fortran 95, program.f90. Dans tous les cas, la compilation s'effectuera avec la même commande g95.

Le programme premiers_pas.f, devient, avec la norme d'écriture f90 le programme premiers_pas.f90 ci-dessous

```
program premiers_pas
!   Ceci est un petit programme d'intiation à Fortran.
print*,'Ca marche !'
print*,'Meme si la ligne est trop longue, on peut la laisser en sur une ligne si on utilise la norme
print*,'On peut aussi la couper mais avec une syntaxe un peu', &
& 'différente'
end
```

2 Les types de variables

Une variable est un emplacement dans la mémoire vive (RAM) de l'ordinateur, dans laquelle on va pourvoir stoker des valeurs.

On distingue en Fortran essentiellement 4 types de variables : les caractères (character), les nombres réels (real) les nombres entiers (integer), les variables logiques vrai/faux (logical).

Une variable possède un nom (xxx, par exemple) et on peut attribuer une valeur à cette variable via l'instruction

```
xxx=1.
```

En théorie, en Fortran, les vraiables ont un type prédéfini en fonction de leur première lettre. Par exemple, une variable commençant par un 'x' est un réel alors qu'une variable commençant par un 'i' est un entier. Cette attribution automatique des types de variables est en fait un piège, et il DOIT ETRE OUBLIE CAR IL EST LA SOURCE D'ERREUR A L'INFINI. Concrêtement, on rajoute juste après la première ligne du programme une ligne

```
implicit none
```

qui annule cette attribution automatique des types.

Du coup, **toute variable doit être déclarée avant d'être utilisée**. Les déclarations doivent être placées immédiatemment après la commande 'implicit none'.

>>>>>>>>>> Implcite none tu utiliseras

Exemple de déclaration et attribution de variables

```
program attribution
implicit none

!   Exemple de déclaration et attribution de variables

real xxx
integer ceci_est_un_entier
logical oui_ou_non
character*10 chaine1
character*50 chaine2
character*60 chaine

xxx=2.
ceci_est_un_entier=2
oui_ou_non=.false.
chaine1='Voici un '
chaine2='exemple de declaration et attribution de variables'

print*,'Voici un '//'exemple de declaration et attribution de variables'

print*,'Bis : ',chaine1//chaine2

chaine=chaine1//chaine2
print*,'Ter : ',chaine

print*,'xxx=',xxx
print*,'oui_ou_non et ceci_est_un_entier prennent les valeurs', &
     &    oui_ou_non,'et',ceci_est_un_entier

      end
```

La valeur d'un nombre real comporte en général un '.'. Par exemple, on peut écrire le réel $1,02 \times 10^{-3}$: 0.00102, .001020 ou encore 1.02E-3. L'instruction 'xxx=1' va également attribuer la valeur real 1. à la variable xxx mais seuelement parce que xxx a été déclaré en real. Il est donc plus propre d'écrire 'xxx=1.'.

La représentation en machine des nombres entier '1' et réel '1.' est complètement différente. La base de la représentation en machine est toujours le codage en base 2, avec des 0 et des 1. Un entier sera directement décomposé en base deux, en gardant un bit pour le signe. Les réels sont représentés sous la forme $x = \pm 0.m \times 2^e$. Les nombres m, e et le signe sont codés en base deux.

Suivant les compilateurs, les 'real' peuvent prendre plus ou moins de place dans la mémoire vive de la machine. Sur un PC-Linux standard, avec g95, les 'real' occupent par défaut 4 octets (bytes en anglais) soit 4×8=32 bits. Les entiers sont codés pour leur part

sur deux octets (16 bits).[1]

Une chaîne de caractère est une suite de caracères compris entre deux signes '. On peut concaténer deux chaînes de caractères (les mettre bout à bout) avec le signe // (par exemple 'premiere chaine '//'seconde chaine')

La commande

```
print*,A,B,C
```

imprime à l'écran successivement A, B et C, où A, B et C peuvent être des noms de vriables de n'importe quel type ou, directement les valeurs correspondantes.

Exercice :
quel sera le résultat de l'execution du programme

```
program attribution_exercice
implicit none

!   Exemple de déclaration et attribution de variables

real xxx
integer ceci_est_un_entier
logical oui_ou_non

xxx=1.
ceci_est_un_entier=2
oui_ou_non=.false.

print*,'xxx,ceci_est_un_entier,oui_ou_non'
print*,xxx,ceci_est_un_entier,oui_ou_non
print*,1.,2,.false.
end
```

On peut également attribuer des valeurs aux variables par les instructions 'data' ou 'parameter', située après la déclaration de la variable mais avant la fin des déclarations.

```
program attribution_par_data_et_parametres
implicit none
```

[1]On peut forcer le nombre d'octet pour une variable donnée. par exemple une déclaration 'real*8 xxx' plutôt que 'real xxx' veut dire que la variable 'xxx' est codée sur 8 octets, soit 64 bits. Le standard correspond donc à la déclaration 'real*4 xxx'. Les processeurs des machines sont en fait eux-mêmes écrits soit en 32 soit en 64 bits. Sur une machine 32 bits, on pourra utiliser des 'real*4' ou des 'real*8'. C'est le cas des PCs standards. Sur une machine 64 bits, on pourra utiliser des 'real*8' et 'real*16'. Les entiers peuvent eux aussi être codés sur 2, 4 ou 8 octets. La déclaration 'real xxx' correspondra à un 'real*4' sur une machine 32 bits et à un 'real*8' sur une machine 64 bits. On peut aussi déclarer un nombre en précision double par rapport à la précision standard de la machine via l'instruction 'double precision xxx' qui correspondra alors à un real*8 sur une machine 32 bits et à un real*16 sur une machine 64 bits.


```
! La façon de faire en Fortran 77
integer im
parameter (im=3)
real xxx
data xxx/1./

! La version Fortran 90
integer,parameter :: jm=2
real :: yyy=2.

print*,'ben alors :' ,xxx,yyy,im,jm
end
```

3 Les tableaux

On peut définir des tableaux contenant une suite de variables d'un même type. Cette déclaration se fait en rajoutant des parenthèses après la variable comme

```
real xtab(3,2)
```

les nombres 10 et 3 s'appellent les dimensions du tableau. Il est préférable de définir ces dimensions préalablement comme des variables (entières). Il faut alors attribuer des valeurs à ces variables via l'instruction parameter.[2]

```
program les_tableaux
implicit none

integer,parameter:: im=3,jm=2
real,dimension(im,jm) :: xxx

data xxx/11.,12.,13.,21.,22.,23./
print*,'Le tableau xxx contient les elements ',xxx, ' ranges dans cet ordre.'
print*,'xxx(1,2)=',xxx(1,2)
print*,'sous forme matricielle '
print*,xxx(:,1)
print*,xxx(:,2)

end
```

Les éléments d'un tableau sont en fait rangés dans la mémoire de l'ordinateur comme une suite linéaire de variables, organisée de la façon suivante :
xxx(i=1,j=1),xxx(i=2,j=1,) ... xxx(i=im,j=1), xxx(i=1,j=2), xxx(i=2,j=2) ...

[2]Contrairement à une attribution classique, du type 'xxx=1', qui sera réalisée lors de l'execution du programme, l'instruction 'parameter' attribue des valeurs aux variables lors de la compilation. Le compilateur peut ainsi savoir à l'avance quelle seront les dimensions du tableau et prévoir quelle sera la place à réserver en mémoire lors de l'execution du programme

4 Les boucles

Les boucles iteratives peuvent prendre différentes formes. La plus classique est la forme 'Do ... enddo'

```
program les_boucles

!Le meme que les_boucles.f mais en fortran90

implicit none
integer im,jm,i,j
parameter (im=3,jm=2)
real xxx(im,jm)
do j=1,jm
   print*,'boucle exterieur j=',j
   do i=1,im
      print*,'boucle interieure i=',i
      xxx(i,j)=10.*j+i
   enddo
enddo
print*,'Le tableau xxx contient les elements ',xxx,' ranges dans cet ordre.'
print*,'xxx(1,2)=',xxx(1,2)
end
```

[3]

Il existe aussi en fortran 95 des boucles conditionnelles "Do while ..." ou "Tant que, faire ..."

```
program do_while
implicit none

integer i
i=1
do while (i<10)
   print*,i
   i=i+1
enddo
end
```

Les caractères 'i<10' correspondent en fait à une valeur logique qui est vraie tant que 'i<10'. On revient sur la définition des variables logiques ci-dessous.

Il existe aussi en fortran la possibilité d'effectuer des boucles implicites. Pour un tableau 'real xxx(im)', on peut par exemple récrire l'initialisation des valeurs du tableau à 0., à savoir

[3] Quand c'est possible, il est toujours préférable d'enchainer les boucles dans un ordre tel que l'accès au contenu du tableau se fait dans l'ordre logique. Respecter cet ordre permet souvent d'accélerer considérablement les codes. Ca devient rapidement un enjeu en calcul scientifique. Cette remarque est encore plus vraie sur les ordinateurs dits 'vectoriels', conçus justement pour accélérer les calcul sur des suites de nombres rangés linéairement dans des mémoires particulièrement rapides.


```
do i=1,im
   xxx(i)=0.
enddo
```

plus simplement comme

```
xxx(1:im)=0.
```

ou encore, de façon plus synthétique

```
xxx(:)=0.
```

ou même

```
xxx=0.
```

La forme 'xxx(1 :im)' est souvent préférable car elle rappelle à quelqu'un qui lit le programme, que 'xxx' est un tableau à une dimension, de dimension 'im'.

On montre ci-dessous un petit exemple qui calcule la dérivée en y d'un champ bi-dimensionnel utilisant des boucles plus ou moins implicites. Les 3 façons de faire sont identiques mais la plus synthétique n'est pas forcément la plus facile à lire.

```
PROGRAM gradient
implicit none

! Declarations
integer i,j
integer,parameter :: im=3,jm=2

real, dimension(im,jm) :: champ
real, dimension(im,jm-1) :: grady
real :: dy=2.

! Initialisation du champ
do j=1,jm
   do i=1,im
      champ(i,j)=sqrt(j*(2.*j+i))
   enddo
enddo
print*,'Le champ dont ont prend le gradient'
write(*,'(3f10.2)'),champ(:,1)
write(*,'(3f10.2)'),champ(:,2)

! Premier calcul de gradient
do j=1,jm-1
   do i=1,im
      grady(i,j)=(champ(i,j+1)-champ(i,j))/dy
   enddo
```

```
enddo
print*,'premier calcul du gradient en y'
print*,grady

! Second calcul
do j=1,jm-1
   grady(:,j)=(champ(:,j+1)-champ(:,j))/dy
enddo
print*,'premier calcul du gradient en y'
print*,grady

! Troisieme calcul
grady(:,1:jm-1)=(champ(:,2:jm)-champ(:,1:jm-1))/dy
print*,'premier calcul du gradient en y'
print*,grady

end PROGRAM gradient
```

>>>>>>>>>> Des programmes lisibles tu écriras.

D'autres formes de boucles implicites existent pour la lecture ou l'écriture. Par exemple, si on veut écrire à l'écran juste la première ligne du tableau 'xxx' du programme 'les_boucles', on pourra simplement écrire

```
print*,'premiere ligne du tableau xxx :',(xxx(i,1),i=1,im)
```

ou encore, en fortran 90

```
print*,'premiere ligne du tableau xxx :',xxx(:,1)
```

5 Les conditions et variables logiques

Les variables logiques prennent la valeur vrai (.true.) ou faux (.false.).[4] Les valeurs logiques peuvent être utilisées notamment pour des boucles 'do while' ou pour des tests 'if, else, endif'.

En pratique, les variables logiques sont souvent le résultat d'égalités ou inégalités : 'i<10', 'i==10' ou de combinaison d'inégalités via des combinaisons avec des opérateurs logiques '.and.', '.or.' ou '.not.'. On peut mettre des parenthèses autour des conditions '(i<10).and.(i>10)'.

Dans la norme fortran 77, les égalités et inégalités ne pouvaient pas s'écrire avec les signes '<, > ou =='.

Math	$\leq$	$<$	$=$	$>$	$\geq$
Fortran 77	.le.	.lt.	.eq.	.gt.	.ge.
Fortran 95	=<	<	==	>	>=

[4]Lors d'une impression à l'écran, avec la commande print* par exemple, le .true. devient T et le .false. devient F.

On montre ci-dessous un petit exemple d'utilisation de if, else, endif

```
PROGRAM logical
implicit none

! Declarations
integer i
logical test

i=10
test=i<12
print*,'i=',i,'  i<12=',test
test=i.lt.12
print*,'i=',i,'  i.lt.12=',test

test=i==10
print*,'i=',i,'   i==10',test,i==10,.not.i==12

do i=1,20
   if (.not.i==12 .and. (i>=5 .and. i.lt.15)) then
      print*,' i=',i,'est different de 12 et 5=<i< 15'
   else
      print*,'ce n est pas le cas pour i=',i
   endif
enddo
end PROGRAM logical
```

6 Les opérations et fonctions

On peut effectuer sur les variables entières ou réelles les opérations classiques : addition '+', multiplication '*', division '/'. On peut inclure dans les opérations des parenthèses. Les règles de composition suivent les conventions mathématiques classiques. $x * y + z$ est l'équivalent de $(x * y) + z$ et $x + y * z$ correspond à $x + (y * z)$.

Fortran possède également tout un tas de fonctions prédéfinies dont l'argument est inséré dans une parenthèse comme la racine carrée (notée sqrt(x)) , les fonctions trigonométriques (sin(x), cos(x), tan(x), tanh(x) ...).

Les puissances comme x^y se nottent $x * *y$.

7 Entrées/Sorties

Les entrées et sorties sont des éléments essentiels d'un programme. Le minimum des sorties est un 'print*' qui donne, à l'écran le résultat d'un calcul.

Les commandes de base pour écrire (sortie) ou lire (entrée) des variables depuis ou vers l'extérieur sont respectivement les instructions write et read. Les écritures et lectures

se font vers ou depuis soit des fichiers de données, soit les 'sorties et entrées standard', à savoir l'écran et le clavier. Dans le cas où on lit/écrit sur des fichiers, ces fichiers peuvent être soit des fichiers de caractères ASCII, soit direcment des représentations binaires des nombres (c'est à dire par exemple la suite des 32 bits utilisés pour coder un real).

La syntaxe de base est la même pour les deux. Par exemple pour écrire le contenu de la variable reelle xxx sous forme de caractères (par exemple 1.02000e-3) on écrira

```
write(unit,format) xxx
```

'unit' repère là où on va écrire ou lire et 'format' définit le format.

'unit' prendra la valeur '*' pour read (resp. write) si on veut lire (resp. écrire) sur l'entrée (resp. la sortie) standard, c'est à dire depuis le clavier (resp. sur l'écran). Si 'format' prend la valeur '*', le format est libre, ce qui veut dire que c'est Fortran qui choisit le format. Par exemple

```
integer i
write(*,*) 'entrer une valeur de i'
read(*,*) i
write(*,*) 'vous avez entre i=',i
```

lit une valeur d'un entier entré au clavier. Remarquer que les instructions 'print*,' et 'write(*,*) ' sont équivalentes.

Si on ne veut pas utiliser le format standard, on doit spécifier un format. Ceci peut se faire de deux façons. Soit en définissant un format avec un numéro.[5]

```
1000   format(3f10.2)
```

qui veut dire qu'on définit un format, étiqueté 1000, et qui consiste en l'écriture de 3 réels succsessifs, écrits chacun sur 10 caractères avec 2 caractères après la virgule.

Exemple :

```
program ecriture_avec_format
implicit none
integer im,jm,i,j
parameter (im=3,jm=2)
real xxx(im,jm)

! Initialisation du tableau xxx
do j=1,jm
   do i=1,im
      xxx(i,j)=10.*j+i
   enddo
enddo
```

[5]En Fortran 77, exceptionnellement, ce numéro sera écrit dans les premières colonnes du programme fortran.


```
! Ecriture avec le format 1000
1000  format(3f10.2)
write(*,1000) xxx

! Ecriture avec le meme format mes directement
print*,'On peut obtenir la même chose avec '
write(*,'(3f10.2)') xxx

end
```

Comme le tableau xxx contient en fait ici 6 éléments et que le format n'écrit que 3 nombres réels, les 6 éléments seront écrits trois par trois sur deux lignes consécutives.

Pour savoir comment spécifier les formats, il faut se reporter à une documentation plus complète. Mais, en général, il est plus simple d'utiliser le format libre '*' qui permet de se débrouiller dans la plupart des cas.

Pour écrire ou lire dans un fichier, il faut d'abord ouvrir ce fichier en utilisant la commande open.

Si le fichier est un fichier ascii, l'ouverture se fait par

```
open(10,file='le_fichier_ascii',form='formatted')
```

Et on écrira ou lira alors dans ce fichier avec une instruction du genre 'read(10,*)' ou 'write(10,*)'. On peut aussi écrire 'file=chaine' ou chaine est déclaré comme une chaîne de caractères à laquelle on attibue une valeur.

Pour les fichiers binaires, il en existe (au moins) deux types. Les fichiers binaires standard et les fichiers à accès direct. Dans les premiers, on écrit les binaires les uns à la suite des autres. Pour relire ces fichiers, on est essentiellement obligé de tout relire depuis le début. Les seconds, appelés fichiers à accès direct, sont organisés sous la forme d'enregistrements (ou 'record') de tailles identiques.

Les ouvertures de ces fichiers s'écrivent

```
integer taille_d_un_record
taille_d_un_record=4*100
open(10,file='fichier_binaire',form='unformatted')
open(11,file='fichier_acces_direct',form='unformatted' &
&  access='direct',recl=taille_d_un_record
```

Dans le second cas, on ouvre, avec l'étiquette 11, un fichier en accès direct composé d'enregistrements de 400 octets, soit par exemple de quoi stoker 100 nombres 'real*4'.

L'écriture dans un fichier binaire se fait sans format avec une instruction 'write(10) xxx' par exemple pour un fichier binaire simple et en spécifiant le numéro de l'enregistement pour le fichier à accès direct. 'wtite(11,rec=2) xxx' signifie qu'on écrit le tableau xxx sur l'enregistrement 2 du fichier 11.

On montre ci-dessous un exemple d'ecriture dans un fichier en acces direct et relecture avec changement de l'ordre de lecture.

```
program acces_direct
implicit none
integer im,jm,i,j
parameter (im=3,jm=2)
real xxx(im,jm)

do j=1,jm
   do i=1,im
      xxx(i,j)=10.*j+i
   enddo
enddo
print*,'Le champ xxx original'
write(*,'(3f10.2)') xxx


open (11,file='tt',form='unformatted',access='direct',recl=4*im)
do j=1,jm
   write(11,rec=j) (xxx(i,j),i=1,im)
enddo

do j=1,jm
   read(11,rec=jm-j+1) (xxx(i,j),i=1,im)
enddo

print*,'Le champ xxx apres ecriture et relecture'
write(*,'(3f10.2)') xxx

end
```

On peut spécifier beaucoup d'autres options dans les commandes open. Par exemple ...,status='old',.. signifie que le fichier est ancien. Dans ce cas, si le fichier n'existe pas déjà, il ne sera pas ouvert. status peut prendre la valeur 'old', 'new' ou 'unknown'. La valeur 'new' est utile si on veut eviter d'écraser un fichier qui existe déjà (je programme refusera d'ouvrir le fichier dans ce cas).

8 Sous-programmes

La base

Dés qu'il est un peu long, un programme doit être découpé en tâches indépendantes et qu'on puisse tester indépendamment.

On va se concentrer ici sur l'utilisation des sous-programmes ou 'subroutine'. Un sous-programme diffère d'un programme (dit principal) parce qu'il ne peut pas être éxecuter seul. Il ne sera éxecuté que si un programme principal (ou un sous-programme lui-même appelé par un programme) fait appel à lui via l'instruction call.

Les sous-programmes comencent donc par une instruction


```
subroutine sous_programme(arguments,...)
implicit none
```

(ne pas oublier de répéter la commande 'implicit none' dans tous les sous-programmes !) et se termine par

```
return
end
```

Entre les deux, on trouve les mêmes déclarations et instructions que dans un programme principal. L'instruction 'return' signifie qu'on retourne au programme principal. Il peut éventuellement y en avoir plusieurs dans le même sous-programme, sous différentes options contrôlées par une instruction 'if' par exemple.

Les arguments

Le sous-programme peut disposer d'un certain nombre d'arguments. Les arguments sont des variables qui sont échangées entre le programme appelant et le sous-programme appelé. Le meme argument peut avoir un nom différent dans les deux programmes. Un exemple simple

```
program arguments
!   Ceci est un petit programme d'intiation à Fortran.
real x,xs2
integer i

do i=1,4
   x=i
   call divise_par_2(x,xs2)
   print*,'x et x/2 ',x,xs2
enddo

end

subroutine divise_par_2(x,xdivisepar2)
implicit none
real x,xdivisepar2
xdivisepar2=x/2.
return
end
```

Attention : le même argument doit avoir le même type dans les programmes appelant et appelé. Il peut être bon de tester le programme

```
PROGRAM probleme
implicit none

real x
integer i
x=1.
print*,'x=',x
call pas_bien(x)
end PROGRAM probleme

SUBROUTINE pas_bien(x)
implicit none
integer x
print*,'x=',x
return
end SUBROUTINE pas_bien
```

Pour le passage de tableaux en argument, il existe différentes conventions. En Fortran 77 Standard, ce qui est échangé entre les programmes appelant et appelé, c'est l'adresse du premier élément du tableau. Comme les tableaux sont écrits de façon séquentielle dans la mémoire de l'ordinateur (occuppant des cases mémoires consécutives), on sait ce qui se trouve derrière le premier élément.

En Fortran 90, on peut aussi passer directement un tableau complet, ou sous-parties d'un tableau en spécifiant la partie du tableau que l'on passe. Par exemple tab(4 :6,2 :10) correspond à la sous-matrice de la matrice tab(i,j) avec i variant de 4 à 6 et j variant de 2 à 10.

On montre ci-dessous des exemples d'utilisation de ces différents modes de passage d'arguments.

```
PROGRAM arguments_tableaux
implicit none

integer,parameter :: im=7,jm=5
integer i,j
real tab(im,jm)

do j=1,jm
   do i=1,im
      tab(i,j)=10.*j+i
   enddo
enddo
print*,'Contenu du tableau tab, de dimension im=',im,'  jm=',jm
write(*,'(7i3)') nint(tab)

print*
print*,'call sous_tableau(im*jm,tab)'
call sous_tableau(im*jm,tab)
```



```
print*
print*,'call sous_tableau(4,tab(2:5,2))'
call sous_tableau(4,tab(2:5,2))


print*
print*,'call sous_tableau2(im,jm,tab)'
call sous_tableau2(im,jm,tab)

print*
print*,'call sous_tableau2(3,2,tab(2:4,4:5))'
call sous_tableau2(3,2,tab(2:4,4:5))

end PROGRAM arguments_tableaux


SUBROUTINE sous_tableau(n,tab)

implicit none
integer i,n
real tab(n)

write(*,*) nint(tab)

return
end SUBROUTINE sous_tableau

SUBROUTINE sous_tableau2(im,jm,tab2)
implicit none
integer i,im,jm,j
real tab2(im,jm)

do j=1,jm
   write(*,*) nint(tab2(:,j))
enddo

return
end SUBROUTINE sous_tableau2
```

Les différents types de variables

On vient de voir comment on passe des variables en arguments entre programme appelant et programme appelé.

Il existe aussi un moyen de réserver une zone mémoire pour des variables, qu'on pourra ensuite appeler à partir de n'importe quelle partie du programme. Il s'agit des "common block". Un common est composé d'un identifiant et d'une liste de variables. Par exemple :

```
program common
implicit none
common/constantes/pi,gravite
real pi,gravite


call initialisation_des_constantes

print*,'On a initialise les constantes '
print*,'pi=',pi
print*,'g=',gravite

end

subroutine initialisation_des_constantes
implicit none
common/constantes/pi,gravite
real pi,gravite

pi=2.*asin(1.)
gravite=9.8

return
end
```

Les common sont considérés comme obsolètes en fortran 95 mais sont encore beaucoup utilisés pour stoker des constantes.

Par oposition aux variables globales des 'common', on peut définir des variables locales, internes à un sous-programme, qui ne seront pas vues de l'extérieure.

Suivant la version de fortran, les options de compilation, etc ... les variables locales à un sous-programme peuvent être gérées de façon très différentes par l'ordinateur.

A priori, il faut partir du principe que d'un appel à l'autre, une variable locale a pu changer de valeur. C'est à dire que la zone mémoire attribuée à cette variable pendant l'execution du sous-programme a pu, entre deux appels au sous-programme, être utilisée, par exemple pour mettre une variable locale d'un autre sous-programme. C'est ce qu'on appelle l'allocation dynamique de mémoire.

Si on veut forcer à ce que la valeur d'une variable soit retenue d'un appel sur l'autre, il faut inclure une instruction save pour sauvegarder cette variable particulière. On montre ci-dessous un exemple où on calcule un sinus à partir d'un nombre en degrés. On a besoin du nombre π pour transformer les degrés en radian. Pour éviter de recalculer π à chaque fois, on ne le calcule qu'au premier appel, identifié lui-même par une clef logique sauvegardée, et on le sauvegarde pour les appels suivants. Remarquer que l'instruction 'data' initialise la valeur de la variable au départ mais pas à chaque appel.

Le fait que des variables puissent utiliser des mêmes zones mémoires dans des sous-programmes différents, s'appelle l'**allocation dynamique de mémoire**. Par opposition, on parlera d'allocation statique si chaque variable se voit attribuer, en début de programme, une zone mémoire bien définie qui lui sera reservée pendant toute l'exécution de programme.

L'allocation dynamique est beaucoup plus économique pour la mémoire.

L'allocation dynamique de mémoire est sans doute le gain principal du passage de la norme 77 à 90. Avec l'allocation statique, les dimensions d'un tableau doivent être définies une fois pour toute, soit en dur – tab(10,7) par exemple – soit avec des dimensions définies au préalable comme 'parameter' comme on l'a vu plus haut.

Avec l'allocation dynamique de mémoire, on peut définir, dans un sous programme, des tableaux dont la dimension n'était pas connue a priori. C'est le cas par exemple de l'exemple ci-dessous.

```
PROGRAM dynamique
implicit none
integer im,jm

print*,'entrer les dimensions im et jm d un tableau'
read(*,*) im,jm

call tableau(im,jm)

end PROGRAM dynamique

subroutine tableau(im,jm)
implicit none
integer i,j,im,jm
real tab(im,jm)

do j=1,jm
   do i=1,im
      tab(i,j)=10.*j+i
   enddo
enddo

do j=1,jm
   write(*,*) nint(tab(:,j))
enddo
return
end SUBROUTINE tableau
```

Organisation en fichiers

Les sous-programmes peuvent être écrits dans le même fichier que le programme principal. Ils peuvent aussi être écrits dans des fichiers indépendants : program.f, sub1.f, sub2.f ... Ceci permet d'utiliser le sous-programme sub1.f pour program.f mais aussi pour program2.f par exemple.

Il faut alors compiler d'un coup les différents fichiers

```
g95 program.f sub1.f sub2.f -o program
program
```

9 Petits conseils pour une programmation efficace

- Plus qu'un conseil : toujours utiliser l'instruction implicit none.
- Utiliser des noms explicites pour les variables utilisées dans des grands programmes.
- Couper les programmes en petits sous-programmes qu'on puisse tester indépendamment.

10 A suivre

cpp
make
make_make