

Table des matières

1	Préambule	3
2	Compilation	3
3	Transition fortran 77/fortran 90	3
3.1	Instructions obsolètes ou dépréciées	3
3.2	Comparaison f77/f90	4
4	Les bases	5
4.1	Éléments de syntaxe	5
4.2	Déclaration de variables	5
4.3	Afficher et lire des informations (entrée et sortie standard)	6
4.4	Les expressions arithmétiques	6
4.4.1	Cas de la division	7
4.4.2	L'opérateur d'élévation à la puissance	7
4.5	Les expressions logiques	7
4.6	Les expressions constantes	7
4.7	Les instructions de contrôle	8
4.7.1	L'instruction <i>if</i> structuré	8
4.7.2	L'instruction <i>select case</i>	9
4.7.3	La boucle <i>for</i>	10
4.7.4	La boucle <i>tant que</i>	10
4.7.5	Les instructions <i>exit</i> et <i>cycle</i>	10
5	Les tableaux	11
5.1	Déclaration des tableaux	11
5.2	Les opérations relatives aux tableaux	11
5.2.1	Affectation collective	11
5.2.2	Les expressions tableaux	12
5.2.3	Initialisation des tableaux à une dimension	13
5.2.4	Les sections de tableau	13
5.2.5	Les fonctions portant sur des tableaux	15
5.3	L'instruction <i>where</i>	15
5.4	Les tableaux dynamiques	15
6	Les Modules	17
6.1	Struture générale	18
6.2	Appel des modules depuis un programme principal	18
6.3	Accès à tout ou partie d'un module	18
6.3.1	Protection interne	19
6.3.2	Protection externe	20
6.4	Partage de données et variables	20
7	Les Procédures (fonction et subroutine)	20
7.1	fonctions	22
7.2	Subroutine	22
7.3	transmission d'une procédure en argument	23
8	Vers les objets : les types dérivés	24
9	Astuces et petits bouts de code	25
9.1	Lire un fichier de longueur inconnue	25

10 Optimisation	25
10.1 Comparaison f77/f90	25
10.2 Profiling	25
10.2.1 flat profile	26
10.2.2 call graph	26
10.3 Des opérations équivalentes ne s'exécutent pas forcément avec la même rapidité	27
10.4 Use Lookup Tables for Common Calculations	28
10.5 Minimiser les sauts dans l'adressage mémoire	29
10.6 Utiliser les options d'optimisation du compilateur	30
11 Avancé	31
11.1 Les pointeurs	31
11.2 Attributs des variables lors de leur déclaration	32
11.2.1 Une constante : parameter	32
11.2.2 Entrée ou sortie : intent	32
11.3 Débuger des programmes fortran avec gdb	32
11.3.1 Savoir où on se trouve dans le programme	33
11.3.2 Afficher le contenu d'une variable fortran avec gdb	33
11.3.3 Mettre le programme en pause à un endroit particulier	33
11.3.4 Débuggage avancé	34
11.4 Erreurs de compilation	34
11.4.1 Utilisation de fonctions internes à un module	34
11.4.2 Utilisation de fonctions d'un module	34
11.4.3 Utilisation de subroutine en paramètre d'autres subroutines	34
11.4.4 Function has no implicit type	34


1 Préambule

Ceci est un tutoriel fortran 90, il a pour but de donner des astuces de programmations, des bonnes pratiques, présenter ce qui se faisait en fortran 77 et qu'il ne faut plus faire.

Dans la suite on considèrera le format libre, c'est à dire que les lignes peuvent avoir jusqu'à 132 caractères.

2 Compilation

Le compilateur traduit les instructions qui ont été tapées par le programmeur et produit, si aucune erreur n'a été faite, en langage machine. La traduction est placée dans un fichier objet dont le nom est identique à celui du fichier source, mais dont l'extension est cette fois .o sous UNIX. Ceci est schématisé sur [FIGURE 1]

 Dans certains cas, l'éditeur de liens est automatiquement appelé et rend le programme exécutable.

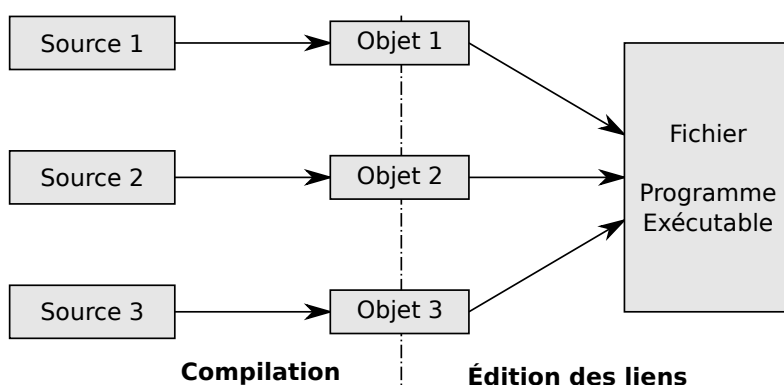



FIGURE 1 – La compilation de tous les fichiers source doit se faire avant l'édition des liens pour créer le fichier exécutable.

L'application complète comportera tous les modules liés. Tout d'abord, il conviendra de compiler séparément sans édition des liens chaque module. À l'issue de cette opération, on obtiendra des modules objets, c'est à dire en langage machine, mais sans adresse d'implantation en mémoire. On les reliera tout en fixant les adresses à l'aide de l'éditeur de liens. Le fichier obtenu sera le programme exécutable. Ceci est schématisé sur [FIGURE 2]

 La compilation d'un fichier source doit se faire *après* la compilation de tous les modules dont il dépend.

3 Transition fortran 77/fortran 90

3.1 Instructions obsolètes ou dépréciées

Obsolètes	Déprécié
IF arithmétique	format fixe
GO TO assigné	COMMON
RETURN multiple	DATA au milieu des inst.
FORMAT assigné	BLOCK DATA
DO sur une même instruc.	EQUIVALENCE
Index réel de boucle DO	GO TO calculé
branchement sur END IF	INCLUDE
PAUSE	ENTRY
descripteur H	DOUBLE PRECISION
	Instructions Fonction
	SEQUENCE
	DO WHILE

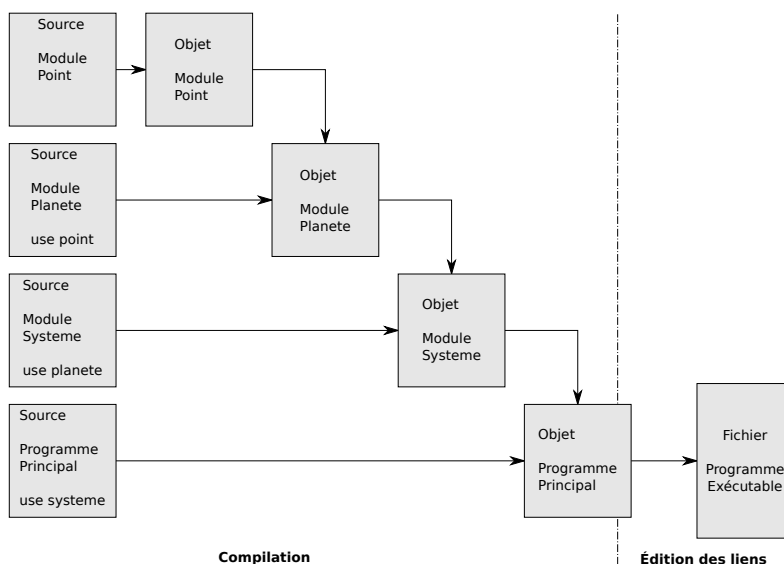


FIGURE 2 – Dans le cas présent, on doit compiler le module point, puis compiler le module planète, puis compiler le module système, et enfin compiler le programme qui fait appel au module système. La compilation d'un fichier source doit se faire *après* la compilation de tous les modules dont il dépend.

3.2 Comparaison f77/f90

En fortran 77, voici les temps d'exécution :

```

arguin.cossou> gfortran -o timings timings.f
arguin.cossou> ./timings
\nInteger powers tests:
A**4 Duration=          1.3177989999999999
A**N (N=4) Duration=    2.4546269999999994
A*A*A*A Duration=      1.2278140000000004
\nLook-up table tests:
Lookup table created, duration=      5.0052380000000003
Repeated pi/4 & sine, duration=     6.3040409999999998
Repeated sine, duration=      12.6200810000000003
All lookups, duration=      1.2658079999999998
\nLarge array tests:
X outer, Y inner, duration=      14.0598619999999995
Y outer, X inner, duration=      5.0562309999999996

```

En fortran 90, en adaptant simplement le code :

```

arguin.cossou> gfortran -o timings timings.f90
arguin.cossou> ./timings
\nInteger powers tests:
A**4 Duration=          1.3827890000000000
A**N (N=4) Duration=    2.9805470000000005
A**pN (pointer) Duration=    2.9005589999999994
A*A*A*A Duration=      1.2208150000000009
\nLook-up table tests:
Lookup table created, duration=      9.99000000000194177E-004
Repeated pi/4 & sine, duration=     1.0698379999999990
Repeated sine, duration=      12.6190810000000000
All lookups, duration=      1.1228300000000004
\nLarge array tests:
X outer, Y inner, duration=      12.9050390000000002
Y outer, X inner, duration=      4.9852419999999995

```

Ce que j'ai fait :

- enlever les labels dans les boucles **do**
- rajouter un test de plus où je défini un pointeur vers l'exposant.

4 Les bases

4.1 Éléments de syntaxe

Une ligne ne peut dépasser 132 caractères. Il est possible cependant d'étendre une instruction de plus de 132 caractères sur plusieurs lignes.

Pour continuer une ligne, en cas de ligne trop longue :

```
print *, 'Montant_HT:', montant_ht, &
      'TVA:', tva, &
      'Montant_TTC:', montant_ttc
```

Pour continuer une chaîne de caractère par contre, il faut impérativement utiliser deux caractères « & » :

```
print *, 'Entrez_un_nombre_entier_&
      &compris_entre_100_&_199'
```

Les commentaires commencent par le symbole « ! » :

```
if (n < 100 .or. n > 199) ! Test cas d'erreur
! On lit l'exposant
read *,x
! On lit la base
read *,y
if (y <=0) then ! Test cas d'erreur
  print *, 'La_base_doit_etre_un_nombre_>_0'
else
  z = y**x ! On calcule la puissance
end if
```

Les identificateurs. On appelle identificateurs, les noms des variables, des fonctions, des sous-programmes... Ils obéissent aux règles suivantes :

- ils sont composés de lettres (les 26 lettres de l'alphabet) et de chiffres (de 0 à 9) dont la totalité ne peut dépasser 31 caractères.
- ils commencent obligatoirement par une lettre.
- le symbole « souligné » () est un caractère utilisable par les identificateurs (à ne pas confondre avec le signe moins : « - »).
- il n'y a pas de distinction entre les minuscules et les majuscules.

4.2 Déclaration de variables

Le premier bloc d'instructions d'un programme source est composé de la suite de déclaration des types des différentes variables utilisées dans le programme. En fait, Fortran ne rend pas obligatoire les déclarations de type. Si une variable commence par i, j, k, l, m ou n, Fortran 90 considère par défaut que cette variable est entière. Nous déconseillons cependant fortement d'utiliser un typage implicite qui est source de nombreuses erreurs de calcul. Il est donc conseillé de commencer chaque programme par l'instruction **implicit none** qui rend obligatoire la déclaration du type de toutes les variables. Si une ou plusieurs variables ne sont pas déclarées, le compilateur retournera un message d'erreur.

La syntaxe de déclaration des variables est la suivante :

```
type [ , attribut ] :: liste_variables
```

- **type** est le nom du type de variable (integer, real, double precision, complex, logical, character)
- **attribut** est une liste d'attributs optionnels (parameter, dimension, allocatable, intent,...)
- **liste_variables** est la liste des variables que l'on déclare comme ayant ce type.

Exemple :

```

program declaration

implicit none
integer ::          i, j=5           ! type entier
real ::            var, x=2.5        ! type reel simple precision
double precision :: plus_precis      ! type reel double precision
logical ::         reussite          ! type logique
character (10) ::  mot               ! type caractere
complex ::         z = (1.2, 20)     ! type complexe

[ ... ]                               ! bloc d'instructions executables

end

```

Le type `logical` n'admet que deux valeurs `.true.` ou `.false.`

Remarque : Il est possible, voire recommande, d'écrire la déclaration des variables sur plusieurs lignes afin d'en faciliter la lisibilité et d'ajouter des commentaires.

```

program commentaires

implicit none
integer :: i, &           ! indice de boucle sur le temps
              j, &         ! nombre de niveaux
              k             ! indice de boucle sur les niveaux

[ ... ]                     ! bloc d'instructions executables

end

```

4.3 Afficher et lire des informations (entrée et sortie standard)

Pour pouvoir écrire des informations à l'écran, c'est-à-dire des commentaires ou le contenu de certaines variables, on utilise l'instruction `print`. L'affectation d'une variable par l'intermédiaire du clavier se fait en utilisant l'instruction `read`. Si on ne veut pas imposer le format d'écriture (on laisse faire l'ordinateur), on utilise le format par défaut symbolisé par une `*` (voir l'exemple du programme `ecriture-lecture`). Tous les caractères compris entre `' '` (ou entre `" "`) sont écrits à l'écran.

```

program ecriture_lecture
implicit none
real :: var, &
              lu           ! variable lue au clavier

var = 2.5

print*, 'La_variable_var_vaut:', var
print*, 'Entrez_une_valeur_au_clavier'
read*, lu
print*, 'La_valeur_entree_au_clavier_est', lu

end

```

4.4 Les expressions arithmétiques

On retrouve les opérateurs arithmétiques usuels : `« + »`, `« - »`, `« * »` et `« / »`. Ces opérandes ne sont définis a priori que lorsque les deux opérandes sont de même type. Le résultat est du même type que les opérandes.

Le compilateur convertit le type de l'un des opérandes, lorsque ces derniers sont différents, pour effectuer l'opération. Les conversions se font suivant la hiérarchie suivante : entier \rightarrow réel \rightarrow double précision. En présence d'un opérande entier et d'un opérande réel, l'entier est transformé en réel.

4.4.1 Cas de la division

Ainsi le quotient de deux entiers et un entier :

$$\frac{5}{2} = 2 \qquad \qquad \qquad \frac{3}{5} = 0 \qquad (4.1)$$

En revanche :

$$\frac{5.0}{2.0} = 2.5 \quad \frac{5.0}{2} = \frac{5}{2.0} = 2.5 \qquad (4.2)$$

4.4.2 L'opérateur d'élévation à la puissance

L'opérateur d'élévation à la puissance se note `***`. L'expression `a**b` correspond à la notation mathématique a^b .

Le résultat de l'expression `a**b` est entier si a et b sont entiers, sinon le résultat est réel.

Soit b un entier positif,

$$a ** b = a * a * \dots * a \text{ (} b \text{ fois)} \qquad (4.3)$$

$$a ** (-b) = 1 / (a ** b) \qquad (4.4)$$

Pour b réel quelconque et a positif,

$$a ** b = \exp(b * \ln(a)) \qquad (4.5)$$

4.5 Les expressions logiques

Pour comparer deux expressions, Fortran 90 dispose de 6 opérateurs de comparaison, `<`, `<=`, `>`, `>=`, `==`, `/=` qui signifient respectivement, inférieur à, inférieur ou égal à, supérieur à, supérieur ou égal à, égal à, différent de.

- Lorsque les deux expressions à comparer ne sont pas du même type, Fortran convertit le résultat de l'une des expressions dans le type de l'autre suivant les règles décrites précédemment.
- Il faut éviter d'utiliser la comparaison entre expressions non entières : l'expression logique `(a == 0.0)` avec a réel n'a pas grande signification !

Fortran dispose aussi d'opérateurs logiques permettant de combiner des opérateurs de comparaison qui sont, par ordre de priorité décroissante : `.not.`, `.and.`, `.or.`. Ils ont une priorité inférieure aux opérateurs précédents.

Par exemple :

$$y = (.not.(a < b)) \equiv y = (a >= b) \qquad (4.6)$$

La variable y est de type `logical`. Les parenthèses ne sont pas obligatoires mais facilitent la lecture (notez les points obligatoires de part et d'autre de `not`, `and` et `or`).

4.6 Les expressions constantes

Lorsqu'une constante est utilisée plusieurs fois dans un programme (par exemple π), il est utile (et recommandé) de la définir une seule fois en début de programme pendant la déclaration des variables.

Deux syntaxes sont possibles :

```
integer :: nb = 5
real    :: PI = 3.141593
```

Dans ce cas les variable nb et PI peuvent être modifiées dans le programme.

```
integer, parameter :: nb = 5
real, parameter    :: PI = 3.141593
```

nb et *PI* sont alors des constantes symboliques dont les valeurs ne peuvent pas être modifiées durant le programme (le compilateur affiche un message d'erreur s'il trouve dans le corps du programme l'instruction *nb = 12* par exemple).

Les déclarations de constante symbolique se font avant toute autre déclaration. On peut aussi utiliser une expression constante dans la mesure ou le compilateur peut la calculer.

implicit none

```
integer, parameter :: nb = 5
integer, parameter :: nb_max = 2*nb+4
integer, parameter :: nb_min = 2*nb-4
integer, parameter :: nb_elem = nb_max - nb_min + 1
```

```
[...]      ! bloc d'instructions executables
```

end

4.7 Les instructions de contrôle

4.7.1 L'instruction *if* structuré

La forme la plus générale du *if* structuré peut être schématisée comme suit :

```
if (exp_log1) then
    bloc1      ! bloc d'instructions
[ else if (exp_log2) then
    bloc2      ! bloc d'instructions
] ...
[ else
    blocn      ! bloc d'instructions
]
end if
```

où *exp_log* est une expression quelconque de type *logical* (par exemple : *if (a >= 0) then*), *bloc* est un bloc d'instructions, *[]* signifie que le contenu est facultatif, *[] ...* signifie que le contenu peut apparaître plusieurs fois. Dans l'exemple ci-dessus, si l'expression *exp_log1* est vraie alors la suite d'instructions *bloc1* est exécutée.

Sinon, si l'expression *exp_log2* est vraie alors c'est la suite d'instruction *bloc2* qui est exécutée (et ainsi de suite). Enfin, si toutes les expressions précédentes (*exp_log1*, *exp_log2*, ...) sont fausses et si l'instruction *else* est présente, la suite d'instructions *blocn* est exécutée. Si l'instruction *else* est absente, il est possible qu'aucune instruction ne soit exécutée par un bloc *if*.

Remarque : Il est recommandé d'indenter (décaler les blocs d'instructions vers la droite d'un certain nombre de caractères blancs) les différents *if* afin d'assurer cette lisibilité.

Un exemple d'utilisation du *if* structuré est donné dans l'exemple ci-après.

program nom_if

implicit none

integer :: i, j

read*, i, j

```
if (i < 0) then
    print*, 'i_est_negatif'
else if (i > 0) then
    print*, 'i_est_positif'
else
    if (j < 0) then
```



```

        print*, 'j_est_negatif'
    else if (j > 0) then
        print*, 'j_est_positif'
    else
        print*, 'i_et_j_sont_nuls'
    end if
end if

end

```

4.7.2 L'instruction *select case*

La syntaxe générale est la suivante :

```

select case (exp_scal)
[case (selecteur1)
    bloc1      ! bloc d'instructions
[case (selecteur2)
    bloc2      ! bloc d'instructions
]...
end select [nom]

```

où *exp_scal* est une expression de type **integer**, **logical** ou **character**. *selecteur* est une valeur, un intervalle de valeurs ou une liste de valeurs de même type que *exp_scal*.

Cette instruction permet d'exécuter la suite d'instructions *bloc1* lorsque la valeur de l'expression *exp_scal* est égale au *selecteur* (ou dans l'intervalle donné par le *selecteur*). Les intervalles sont de la forme suivante : [valeur1]:valeur2 ou valeur1:[valeur2] (par exemple **case(1:)** signifie que l'on s'intéresse aux valeurs entières comprises entre 1 et 2147483647. Les sélecteurs peuvent faire appel à des expressions constantes. Les valeurs figurant dans les différents sélecteurs d'une même instruction **select case** ne doivent pas se recouper (cela engendre une erreur à la compilation).

```

program case

```

```

implicit none

```

```

character(3) :: reponse

```

```

print*, 'Voulez-vous_continuer_le_programme_'
read*, reponse

```

```

select case (reponse)
    case ('oui')
        print*, 'OK,_ca_roule...'
    case ('non')
        print*, 'Au_revoir!'
        stop
    case default
        print*, 'Veuillez_repondre_par_"oui"_ou_"non"'
end select

```

```

end

```

Remarque : Pour écrire de *bons* programmes fortran, il faut :

- Que dans chaque **case** il y ait une seule valeur du paramètre
- **case default** est optionnel, mais il est conseillé de toujours en mettre un, comme ça on est sûr que quelque chose sera exécuté.
- **case default** est optionnel mais il vaut mieux le placer à la fin du **select case**, c'est plus logique et naturel.

4.7.3 La boucle for

La syntaxe générale est la suivante :

```
do var = debut , fin , [ pas ]
  bloc      ! bloc d'instructions
end do
```

La variable de contrôle *var* est de type **integer** ainsi que les expressions *debut*, *fin* et *pas*.

Cette instruction permet de répéter le bloc d'instructions *bloc* en donnant successivement à la variable *var* les valeurs *debut*, *debut+pas*, ..., *fin*. Si *pas* est absent, il est par défaut égal à 1. La valeur de *pas* peut être négative. Il faut alors que *debut* soit plus grand que *fin* sinon aucune instruction de *bloc* ne sera effectuée.



Il n'est pas possible de modifier, dans le bloc d'instructions de la boucle, la valeur de *var* (le compilateur envoie un message d'erreur) et les modifications éventuelles lors de l'exécution de la boucle de *debut*, *fin* et *pas* ne sont pas prises en compte.

Il est imprudent de chercher à exploiter la valeur de *var* après l'exécution de la boucle **do**. En effet, celle-ci ne prend pas nécessairement la valeur *fin* comme on pourrait le penser a priori.

4.7.4 La boucle tant que

La syntaxe générale est la suivante :

```
do while (exp_log)
  bloc      ! bloc d'instructions
end do
```

Cette instruction permet de répéter le bloc d'instructions *bloc* tant que l'expression logique *exp_log* est vraie. Si *exp_log* est fausse dès le début, le bloc n'est pas exécuté. Sinon, le bloc d'instructions doit modifier *exp_log* pour que la boucle puisse s'arrêter.

4.7.5 Les instructions *exit* et *cycle*

L'instruction **exit** permet de sortir d'une boucle de façon anticipée. Dans l'exemple suivant, les blocs *bloc1* et *bloc2* sont exécutés pour *var* allant de *debut* à *fin* tant que l'expression logique *exp_log* est fausse.

La boucle est interrompue si *var = fin* ou si *exp_log* est vraie. Dans le premier cas on passe au *bloc3*. Dans le second cas, *bloc1* est exécuté mais pas *bloc2*. Ensuite, on continue la boucle **do while** tant que *.not.fini* est vrai.

Comme on le voit sur cet exemple, lorsqu'une instruction **exit** apparaît dans une boucle qui est imbriquée dans une autre boucle, elle met fin à la boucle la plus interne.

```
do while (.not.fini)
  do var = debut , fin
    bloc1      ! bloc d'instructions
    if (exp_log) exit
    bloc2      ! bloc d'instructions
  end do
  bloc3      ! bloc d'instructions
end do
```

Lorsque *exp_log* est vraie, on sort de la boucle **do var**

Dans l'exemple suivant, l'instruction **exit** s'applique à la boucle **do while** grâce à l'utilisation de l'identificateur *boucle_principale*. Ainsi, si *exp_log* est vraie, ni *bloc2*, ni *bloc3* ne sont exécutés et on sort de la boucle **do while**.

```
do while (.not.fini)
  do var = debut , fin
    bloc1      ! bloc d'instructions
    if (exp_log) exit
    bloc2      ! bloc d'instructions
  end do
  bloc3
```

```

        bloc3          ! bloc d'instructions
end do

```

Lorsque *exp_log* est vraie, on sort de la boucle principale **do while**

L'instruction **cycle** permet de modifier le déroulement normal d'une boucle. Dans l'exemple suivant, *bloc1* et *bloc2* sont exécutés pour *var* allant de *debut* à *fin* par pas de 1.

Si l'expression logique *exp_log* est vraie, on passe à la valeur suivante de *var* sans exécuter le *bloc2*. Si *exp_log* est toujours vraie, seul *bloc1* est exécuté.

```

do var = debut, fin
  bloc1          ! bloc d'instructions
  if (exp_log) cycle
  bloc2          ! bloc d'instructions
end do

```

5 Les tableaux

5.1 Déclaration des tableaux

Un tableau est un ensemble ordonné d'éléments de même type. Chaque élément du tableau est repéré par un indice qui précise sa position au sein du tableau. Cet indice est entier. La déclaration des tableaux s'effectue comme suit :

implicit none

```

integer, parameter :: min = -5
integer, parameter :: max = 12
integer, parameter :: nb = 10
integer, parameter :: nb1 = 5, nb2 = 3
real, dimension (nb) :: vect_1      ! tableau de rang 1
integer, dimension (min:max) :: vect_2 ! tableau de rang 1
real, dimension (50) :: vect_3      ! tableau de rang 1
integer, dimension (nb1, nb2) :: t   ! tableau de rang 2
real, dimension (min:max, nb2) :: tab ! tableau de rang 2

```

```

[... ]          ! bloc d'instructions executables

```

end

Remarque : Quand les bornes ne sont pas spécifiées, la borne inférieure est égale à 1. Dans l'un des exemples précédents, seul *nb* est donné et les indices de *vect_1* vont de 1 à *nb*.

On peut aussi écrire : **real :: vect_1(nb)**.

- le nombre de dimensions est le **rang** du tableau (*vect_1* est de rang 1 et *tab* est de rang 2). Le nombre maximum de dimensions est égal à 7.
- le nombre de valeurs possibles pour l'indice d'une dimension donnée est l'**étendue** du tableau suivant cette dimension (*vect_2* est d'étendue 18). L'indice est compris entre la **borne inférieure** (*min* dans le cas de *vect_2*) et la **borne supérieure** (*max* dans le cas de *vect_2*).
- le nombre d'éléments du tableau est la **taille** du tableau; c'est donc le produit des étendues de chaque dimension (*tab* est de taille 54).
- la liste des étendues est le **profil** du tableau (*tab* est de profil (18, 3)).

5.2 Les opérations relatives aux tableaux

5.2.1 Affectation collective

Soit le tableau *mat* de rang 3, si on désire affecter la valeur 1 à tous les éléments du tableau *mat* on effectue, dans les principaux langages de programmation (*Pascal*, *C*, *Fortran*), la suite d'instructions du programme suivant. En Fortran 90, il est possible d'obtenir le même résultat en écrivant l'instruction : **mat = 1**

```
implicit none
```

```
integer :: i, j, k
integer, dimension (5, -3:2, 10) :: mat      ! tableau de rang 3
```

```
do i = 1, 5
  do j = -3, 2
    do k = 1, 10
      mat(i, j, k) = 1
    end do
  end do
end do
```

```
! est equivalent a
mat = 1
```

```
! ou mieux
mat(:, :, :) = 1
```

```
end
```

Remarque : Par soucis de lisibilité, il est souhaitable d'expliciter les opérations sur tableaux en utilisant

```
mat(:, :, :) = 1
```

au lieu de

```
mat = 1
```

afin de pouvoir distinguer les opérations sur tableaux des opérations sur variables simples. Ça permet d'ailleurs de voir directement le nombre de dimensions du tableau.

5.2.2 Les expressions tableaux

On peut affecter une expression à chaque élément d'un tableau. Par exemple, les deux blocs d'instructions du programme suivant sont équivalents.

```
implicit none
```

```
integer :: i
integer, parameter :: dim = 12
```

```
! tableaux de rang 1
integer, dimension (dim) :: a, b
integer, dimension (dim) :: som, prod, racine
```

```
! tableaux de type logique
logical, dimension (dim) :: compare
```

```
do i = 1, dim
  som(i) = a(i) + b(i)
  prod(i) = a(i)*b(i)
  prod(i) = 2*prod(i) + 1
  racine(i) = sqrt(real(a(i)))
  if (som(i) < prod(i)) then
    compare(i) = .true.
  else
    compare(i) = .false.
  endif
end do
```

*! En Fortran 90, les instructions precedentes
! se simplifient de la maniere suivante :*

! equivalent a : som = a + b

som(:) = a(:) + b(:)

prod(:) = a(:)*b(:)

prod(:) = 2*prod(:) + 1

! Attention les elements de a sont entiers

racine(:) = sqrt(real(a(:)))

compare(:) = (som(:) < prod(:))

end

5.2.3 Initialisation des tableaux à une dimension

L'initialisation d'un tableau de rang 1 à n éléments est possible en utilisant une liste de n éléments définie par *elem_1, ..., elem_n*. Le tableau correspondant s'écrit (/elem_1, ..., elem_n/).

Un tableau à plusieurs dimensions ne peut pas être initialisé directement. Il faut définir un tableau à une dimension et utiliser une fonction particulière qui n'est pas présentée dans ce cours : la fonction *reshape*.

program initialisation

implicit none

```
integer :: i
integer, parameter :: n = 5
integer, dimension(n) :: tab1, tab2, t           ! tableaux de rang 1
integer, dimension(n) :: tab3 = (/1,2,3,4,0/)
real, dimension(0:9) :: angle
```

```
tab1(1) = 3 ; tab1(2) = 5 ; tab1(3) = -2 ; tab1(4) = 4 ; tab1(5) = 202
```

```
tab2 = (/ 3, 5, -2, 4, 202/)
```

! les affectations suivantes sont equivalentes

```
do i = 0, 90, 10
```

```
    angle(i/10) = i*0.5
```

```
end do
```

```
angle = ((i*0.5, i=0, 90, 10)/)           ! boucle implicite
```

end

5.2.4 Les sections de tableau

Fortran 90 introduit une notion nouvelle par rapport aux langages tels que *C* ou *Pascal* qui est la section de tableau. L'écriture générale d'une section de tableau est la suivante :

```
tab(borne_inf : borne_sup : pas [, ...])
```

- *tab* est le nom d'un tableau,
- *borne_inf* est la borne inférieure de la section de tableau (c'est la borne inférieure de *tab* si elle est omise),
- *borne_sup* est la borne supérieure de la section de tableau (c'est la borne supérieure de *tab* si elle est omise),
- *pas* est le pas d'incrément (1 par défaut ; si le pas est négatif, la variation d'indice est rétrograde de *borne_sup* à *borne_inf*).

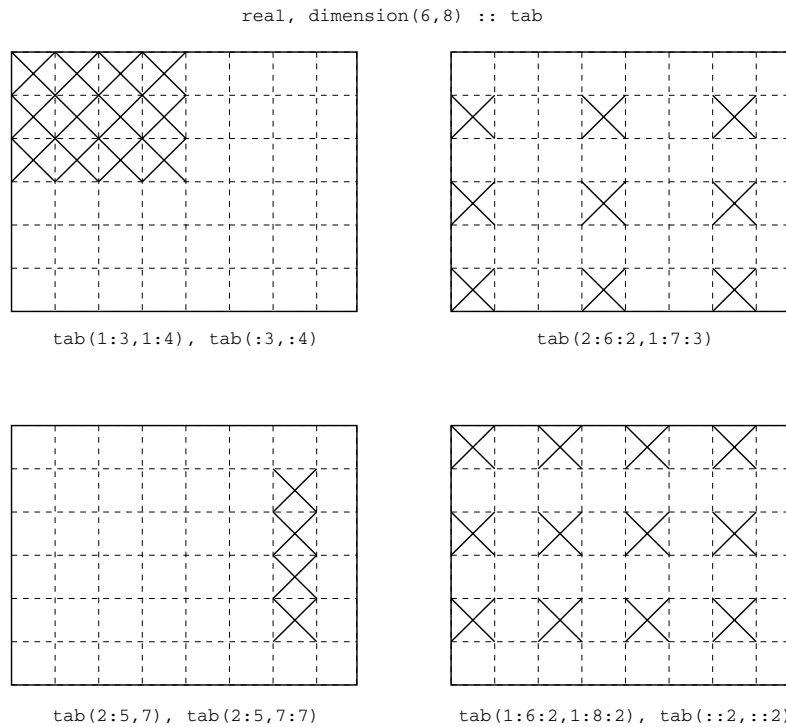


FIGURE 3 – Représentation de différentes sections de tableaux afin de montrer les possibilités de sélections.

Dans le programme suivant,

```
integer, dimension (n) :: w
integer, dimension (p) :: v

v(:) = (/ (i, i=1,p) /)      ! boucle implicite

! EXEMPLE 1
```

```
w(2:4) = v(7:9)
```

les valeurs $v(7)$, $v(8)$, $v(9)$ du tableau v sont affectées respectivement aux éléments $w(2)$, $w(3)$ et $w(4)$ du tableau w . Ainsi, la notation $w(i:j)$ signifie que l'on s'intéresse aux éléments $w(i)$, $w(i+1)$, \dots , $w(j)$ ($j > i$).

Remarque : Si on écrit $w(3:1) = 1$, aucune affectation ne sera effectuée.

Dans le second exemple, on remarque qu'il y a un recoupement, c'est à dire que le même terme apparaît à gauche et à droite du signe égal puisqu'on a les deux affectations « simultanées » :

$$\begin{aligned} v(2) &= (v(1) + v(3))/2 \\ v(3) &= (v(2) + v(4))/2 \end{aligned}$$

Que vaut $v(2)$ dans la seconde affectation ? !

La règle adoptée par *Fortran 90* est la suivante : la valeur d'une expression de type tableau est entièrement évaluée avant d'être affectée.

```
integer, dimension (n) :: w
integer, dimension (p) :: v, v1
```

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

```

! est equivalent a
do i = 1, p
  v1(i) = v(i)
end do
do i = 2, 9
  v(i) = (v1(i-1) + v1(i+1))/2
end do

```

Dans cet exemple, si on n'utilise pas les sections de tableau, on remarque qu'il est nécessaire d'utiliser un tableau tampon *v1* pour effectuer les calculs intermédiaires.

5.2.5 Les fonctions portant sur des tableaux

Il existe en *Fortran 90* des fonctions spécifiques aux tableaux. Les plus usuelles sont **sum** qui fournit la somme des éléments d'un tableau, **maxval** qui donne la valeur maximale d'un tableau, **minval** qui donne la valeur minimale et **product** qui donne le produits des éléments. Les fonctions **dot_product** et **matmul** permettent d'obtenir respectivement le produit scalaire et le produit matriciel de deux tableaux.

Exemple : Soit *A* un tableau à *m* ligne(s) et *n* colonne(s). On cherche la valeur maximale de l'ensemble formé par les éléments se trouvant à la ligne *j* pour les colonnes allant de *i* à *n*. Il suffira pour cela d'écrire l'instruction suivante :

```
maxval(A(j, i:n))
```

Soit *A* une matrice de rang 2 à *n* colonnes. L'instruction

```
sum(A, dim=1)
```

permet d'obtenir un tableau de rang 1 et d'étendue *n* dont chaque élément *i* est le résultat de la somme des éléments de la colonne *i* de *A*.

5.3 L'instruction *where*

Cette instruction permet de traiter les éléments d'un tableau vérifiant une certaine condition. La syntaxe est la suivante :

```

where (inst_log_tab)
  bloc1
[ elsewhere
  bloc2
]
end where

```

inst_log_tab est une instruction logique portant sur les éléments d'un tableau. Lorsque cette condition est vérifiée, le bloc d'instructions *bloc1* est exécuté, sinon le bloc d'instructions *bloc2* est exécuté.

Lorsque *bloc1* ne contient qu'une seule instruction et que *bloc2* est absent, on peut utiliser une forme simplifiée identique au **if** logique.

Ainsi, dans l'exemple suivant, tous les éléments négatifs du tableau *A* sont mis à zéro :

```
where (A < 0) A = 0.0
```

5.4 Les tableaux dynamiques

Quand on ne connaît pas à l'avance la taille des tableaux que l'on souhaite utiliser, on peut "surdimensionner" le tableaux en question au moment de la déclaration mais la méthode la plus élégante consiste à utiliser les tableaux dynamiques (tableaux à allocation différée). L'allocation sera effectuée lorsque les étendues du tableau seront connues (après lecture dans un fichier, au clavier ou après des calculs).

La déclaration d'un tableau dynamique s'effectue en précisant le rang du tableau et en utilisant l'attribut **allocatable** (allouable).

```

! declaration d'un tableau dynamique d'entiers de rang 2
real, dimension(:, :), allocatable :: matrice

```

L'allocation d'un emplacement se fait en utilisant l'instruction **allocate** en précisant chaque étendue :

```
! declaration d'un tableau dynamique d'entiers de rang 2
real, dimension (:,:), allocatable :: matrice
integer :: n, m, verif

[...]! bloc d'instructions executables

! lecture au clavier des etendues
read*, n ! suivant la premiere dimension
read*, m ! suivant la seconde dimension

allocate (matrice(n,m))
```

Remarque : Pour vérifier que l'allocation (ou la désallocation) s'est bien effectuée, on peut utiliser l'option **stat**.

```
allocate (matrice(n,m), stat=verif)
```

verif est une variable entière. Si **verif** = 0, l'allocation s'est bien effectuée, sinon une erreur s'est produite.

Lorsqu'un tableau dynamique devient inutile, il est recommandé de libérer la place allouée à ce tableau. Pour cela, on utilise l'instruction **deallocate** (désallouer).

```
deallocate (matrice) ! liberation
```

Il est possible de transmettre un tableau dynamique en argument d'une procédure sous certaines conditions :

- Le tableau dynamique devra être alloué et libéré dans le programme principal.
- Le programme principal doit contenir l'interface de la procédure. Cette condition n'est pas obligatoire si on utilise un module! (voir par exemple le programme *tableau_dynamique*).

```
program tableau_dynamique
```

```
implicit none
```

```
! declaration d'un tableau dynamique d'entiers de rang 2
integer, dimension (:,:), allocatable :: matrice
```

```
integer :: nb_lig, nb_col, i
```

```
read*, nb_lig ! lecture au clavier du nombre de lignes
read*, nb_col ! lecture au clavier du nombre de colonnes
```

```
! allocation d'un emplacement de profil nb_lig, nb_col
allocate (matrice(nb_lig, nb_col))
```

```
! affectations des elements de la matrice
```

```
do i = 1, nb_col
    matrice(:,i) = i
end do
call affiche(matrice)
```

```
! liberation de l'emplacement correspondant au tableau matrice
deallocate (matrice)
```

```
contains
```

```
subroutine affiche(t)
```



```

implicit none
integer, dimension(:, :), intent(in) :: t      ! dimensionnement automatique
integer :: i, j

do i = 1, size(t,1)
  print*, (t(i,j), j=1,size(t,2))
end do

end subroutine

end

```

Que la subroutine soit incluse dans le programme principal ou dans un module que ce dernier appelle, ça revient au même, il n'y a plus besoin de définir explicitement l'interface. Autrement il faut le faire, et ajouter

```

interface
  subroutine affiche(t)
    integer, dimension(:, :), intent(in) :: t
  end subroutine affiche
end interface

```

à la fin des déclarations de variables dans le programme principal.

6 Les Modules

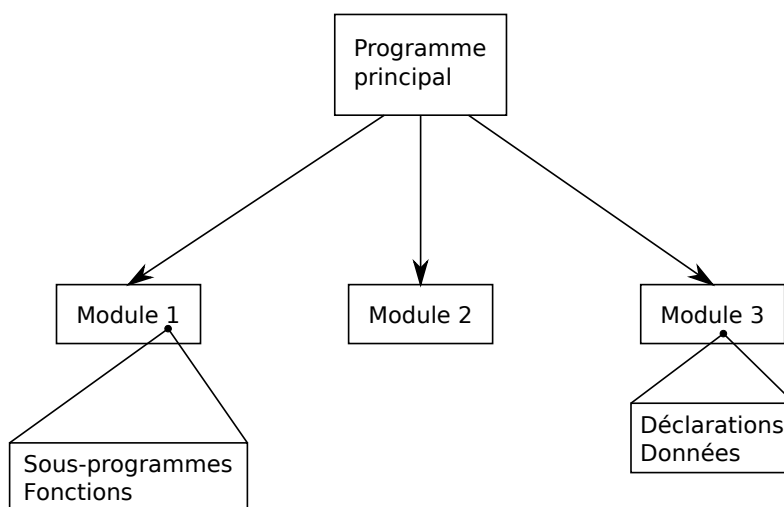


FIGURE 4 – Structure générale d'un programme Fortran 90

[FIGURE 4] présente la structure générale d'un programme *Fortran 90*. Nous allons voir qu'un programme principal peut aussi faire des appels à un ou plusieurs modules.

DÉFINITION 1

Le module se présente comme une unité de programme autonome permettant de mettre des informations en commun avec d'autres unités de programmes.

Ce programme est généralement écrit dans un fichier différent du fichier contenant le programme principal. Le module est compilé indépendamment des autres unités de programme mais ne peut pas être exécuté seul. Comme le montre [FIGURE 4], les modules peuvent contenir des procédures (sous-programmes et fonctions), des blocs de déclarations et des données.

Le module permet de fiabiliser les programmes en évitant la duplication des déclarations et des affectations utilisées par plusieurs unités de programme puisqu'il donne l'accès de son contenu à toutes les unités de programme qui en font l'appel.

Ils permettent :

- Une écriture des sources plus simples : en particulier ils évitent d’avoir à écrire des blocs **interface** qui sont assez lourds quand ils doivent être souvent répétés
- de remplacer avantageusement la notion de **COMMON**
- D’accéder à toutes les ressources du *Fortran 90* avec un maximum d’efficacité et de cohérence : gestion dynamique de la mémoire, pointeurs, généricité, surdéfinition des opérateurs, encapsulation. . .



Les unités **module** doivent être compilés *avant* de pouvoir être utilisées. Si le fichier source est unique, elles doivent être placées en tête.

6.1 Struture générale

Le module peut contenir un ensemble de déclarations et d’affectations et/ou une ou plusieurs procédure(s). Dans ce dernier cas, les procédures doivent être précédées par l’instruction **contains**. Un module commence par l’instruction **module** suivi du nom du module et se termine obligatoirement par **end module**.

```
module nom_module
```

```

implicit none
[ ... ]           ! bloc de declarations

contains         ! obligatoire si suivi de procedures

[ ... ]           ! suite de procedures
```

```
end module nom_module
```

6.2 Appel des modules depuis un programme principal

L’utilisation des modules est très simple; depuis le programme principal, l’appel du module se fait par l’instruction **use** suivi du nom du module. Il faut noter que le nom du module doit être différent de celui du programme principal. L’instruction **use** doit précéder toute autre déclaration.

```
program utilisation_module
```

```

use nom_module
use module_constantes
```

```

implicit none
[ ... ]           ! bloc de declarations

[ call ...
  call ... ] ! appels des procedures contenues dans le module nom_module
[ ... ]           ! bloc d'instructions executables
```

```
end
```

Remarque : Un module ne doit pas se référencer lui-même, même de manière indirecte. Par exemple, si le module *module1* contient **use module2**, ce dernier ne doit pas contenir l’instruction **use module1**.

6.3 Accès à tout ou partie d’un module

Une unité de programme qui appelle un module (via l’instruction **use**) à accès à toutes les entités de ce module (variables, procédures). Il est possible cependant de contrôler l’accès à ces entités pour empêcher les conflits entre différentes unités de programme.



Lorsqu'une unité de programme appelle un ou plusieurs modules, il peut y avoir un conflit entre les identificateurs (noms des variables et des procédures) de l'unité de programme et des modules. Si l'on ne souhaite pas modifier les identificateurs des modules (ce qui peut être laborieux si le module est long), il est possible de renommer ces identificateurs lors de l'appel des modules à partir de l'unité de programme.

```
use nom_module, nom_id_local => nom_id_module
```

- *nom_module* est le nom du module,
- *nom_id_local* est le nom attribué à l'identificateur dans l'unité de programme qui appelle le module,
- *nom_id_module* est le nom de l'identificateur public du module que l'on souhaite renommer.

Exemple :

```
use module_constantes, k_bol => k
```

va permettre d'utiliser la constante k du module *module_constantes* sous le nom k_{bol} dans le programme, afin de ne pas rentrer en conflit avec une variable k qui existe aussi dans le programme.

6.3.1 Protection interne

La première méthode pour contrôler l'accès à un module consiste à protéger certaines entités. Pour cela on utilise les instructions **private** et **public**. Par défaut, l'option d'accès au module est *public*. Toutes les instructions du module se trouvant après l'instruction **private** seront d'accès privé (et donc inaccessible par le programme qui appelle le module). Il est possible aussi d'ajouter l'attribut **private** lors de la déclaration d'une variable pour protéger son accès.

```
module module_atmosphere
```

```
use module_constantes, only : k, Na
```

```
real, private :: T = 300., &      ! temperature [K]
                  m = 28.0e-3, &   ! masse mol. moy. [gmol-1]
                  g = 9.81, &      ! pesanteur [ms-2]
                  Po = 1.01325e5 ! pression standard [Pa]
```

```
contains
```

```
function pression(z)
```

```
    real :: pression          ! pression a l'altitude z [Pa]
    real, intent(in) :: z     ! altitude [km]
    real :: h                  ! hauteur d'echelle [km]
```

```
    h = (k*T*Na)/(m*g)/1.0e3
    pression = Po*exp(-z/h)
```

```
end function pression
```

```
end module module_atmosphere
```

Et on accède au module de la façon suivante :

```
program acces
```

```
use module_atmosphere
```

```
implicit none
```

```
real :: p, &      ! pression [Pa]
        z          ! altitude [km]
```

```

z = 10
p = pression(z)

print *, p, g, h

end

```

Les variables T , m , g et Po ne sont pas accessibles par le programme *acces* malgré l'appel du module *module_atmosphere*. Elles ont, en effet, l'attribut **private** dans *module_atmosphere*. La variable h de la fonction *pression* n'est pas en conflit avec la variable h de *module constantes* grâce à la restriction d'accès via l'instruction **only**.

6.3.2 Protection externe

La seconde méthode de contrôle d'accès entre le module et l'unité de programme qui l'appelle consiste à restreindre l'accès à certaines entités depuis l'unité de programme. Cette restriction s'effectue en utilisant l'attribut **only** lors de l'appel du module (voir le module *module_atmosphere*).

```
use nom_module, only : liste_entites
```

où *liste_entites* est la liste des variables et procédures auxquelles on autorise l'accès lors de cet appel.

6.4 Partage de données et variables

Les modules peuvent être utilisés pour déclarer des variables susceptibles d'être communes à de nombreux programmes. Par exemple, un physicien est amené à utiliser, dans l'ensemble de ses programmes, les différentes constantes de la Physique. Plutôt que de déclarer ces constantes dans chaque programme, il suffit d'utiliser un module approprié dans lequel elles seront affectées une fois pour toute.

```

module module_constants

implicit none

! Constantes de la physique en unite S.I.

real, parameter :: c = 2.99792458e8 , &      ! vitesse de la lumiere [ms-1]
                  G = 6.6720e-11, &          ! constante de la gravitation [Nm2kg-2]
                  h = 6.626176e-34, &       ! constante de planck [JHz-1]
                  k = 1.380662e-23, &       ! constante de Boltzmann [JK-1]
                  Na = 6.022045e23, &      ! constante d'Avogadro [mol-1]
                  Rg = 8.3145 &           ! constante des gaz parfaits [Jmol-1K-1]

end module module_constants

```

7 Les Procédures (fonction et subroutine)

Il existe deux types de procédures : les sous-programmes et les fonctions (respectivement **subroutine** et **function** en anglais). Nous étudierons les fonctions plus loin dans le chapitre. Parmi les procédures, on distingue les procédures externes des procédures internes.

DÉFINITION 2

Fonction Une fonction, à l'instar de son homologue en informatique, est une application qui, à partir de variable d'entrée retourne *variable* de sortie qui peut être de n'importe quel type. La variable de retour est simplement une variable qui porte le même nom de la fonction et qui est retournée sans qu'on ait besoin de faire quelque chose en particulier.

DÉFINITION 3

Subroutine La subroutine, contrairement à la fonction, ne distingue pas, par défaut, les variables d'entrées et de sorties, une même variable peut à la fois être une entrée et une sortie (dans la pratique, avec l'attribut **intent()** lors de la déclaration des variables on peut faire des choses un peu plus propres).

De plus, on n'est pas limité à une variable de sortie. L'inconvénient est que, à part en regardant la déclaration ou le corps de la subroutine, on ne peut pas distinguer simplement les entrées et sorties qui sont simplement une suite de paramètres de la subroutine.

Les sous-programmes externes (**subroutine** ou **fonction**) sont des blocs de code en dehors du programme principal, soit après son instruction **end**, soit dans un fichier séparé qu'il faudra aussi compiler.

```

program nom_program

implicit none
[ ... ]           ! bloc de declaration

[ ... ]           ! bloc d'instructions executables

call nom(liste_arguments)      ! appel du sous-programme

[ ... ]           ! bloc d'instructions executables

end

subroutine nom(liste_arguments)

implicit none
[ ... ]           ! bloc de declaration (arguments et variables locales)

[ ... ]           ! bloc d'instructions executables

end subroutine nom

```

Les sous-programmes internes (**subroutine** ou **fonction**) sont des blocs de code qui vont être dans le corps du programme, à la suite de l'instruction **contains** et avant l'instruction **end**.

```

program nom_program

implicit none
[ ... ]           ! bloc de declaration

[ ... ]           ! bloc d'instructions executables
call nom(liste_arguments)      ! appel du sous-programme
[ ... ]           ! bloc d'instructions executables

contains
subroutine nom1(liste_arguments)
[ ... ]           ! bloc de declaration
[ ... ]           ! bloc d'instructions executables
end subroutine nom1

subroutine nom2(liste_arguments)
[ ... ]           ! bloc de declaration
[ ... ]           ! bloc d'instructions executables
end subroutine nom2

end

```

Dans le cas des procédures externes, on est en présence de domaines indépendants qui ne peuvent communiquer que par le biais des arguments.

Dans le cas des procédures internes, la procédure a accès à toutes les variables définies par son hôte. Ces variables sont dites *globales* et n'ont plus besoin d'être transmises en arguments. En revanche, toutes les variables déclarées dans la procédure interne sont locales à la procédure.

Ainsi, la déclaration de deux variables, l'une dans le programme hôte et l'autre dans la procédure interne avec le même nom provoque la création de deux variables différentes (bien qu'ayant le même nom!). Bien que pouvant paraître attrayante, cette méthode d'utilisation des procédures est à utiliser avec circonspection. En effet, l'utilisation dans le programme principal et le sous-programme interne d'un même nom pour 2 variables a priori différentes, risque de provoquer des erreurs de programmation.



Il est important de noter que les procédures internes ne peuvent être utilisées que par l'hôte qui les contiennent.

7.1 fonctions

En *Fortran 90*, les fonctions peuvent retourner n'importe quel type valide (tableau, type dérivé), mais une seule variable, que l'on attribue via le nom de la fonction elle-même. En définissant le type de la fonction, on définit le type de la variable que l'on va retourner.

```
function nom(liste_arguments)
```

```
implicit none
```

```
real :: nom
```

```
[...]           ! bloc de declaration
```

```
[...]           ! bloc d'instructions executables
```

```
nom = ...       ! expression arithmetique
```

```
end function nom
```

Ainsi, la fonction `nom` retourne un réel. Ce dernier est stocké comme donnée de retour via une ligne d'attribution

```
nom = 3e43
```

où `nom` est le nom de la fonction.

Pour appeler la fonction dans le programme principal, il suffit de faire :

```
program nom_program
```

```
implicit none
```

```
real :: resultat
```

```
[...]           ! bloc de declaration
```

```
[...]           ! bloc d'instructions executables
```

```
resultat = 1 + 3*nom(liste_arguments) ! exemple d'appel de la fonction nom
```

```
[...]           ! bloc d'instructions executables
```

```
end
```

Remarque : La variable dans laquelle on stocke le résultat de la fonction doit bien évidemment avoir le même type que celui déclaré pour le nom de la fonction (et donc sa variable de retour).

7.2 Subroutine

En *Fortran 90*, une subroutine est un bloc de programme avec des entrées et des sorties. Une subroutine `transfert` s'appelle de la façon suivante :

```
call transfert(a,b)
```

où `transfert` est une subroutine qui permet d'échanger les variables *a* et *b* via une variable temporaire définie dans la subroutine :

```
subroutine transfert(a,b)
```

```
implicit none
```

```
real, intent(inout) :: a,b
```

```
real :: c
```

```
c=a
```

```
a=b
```

```
b=c
end subroutine transfert
```

Les attributs `intent(in)`, `intent(out)` ou `intent(inout)` permettent de spécifier si un argument d'une subroutine est un paramètre d'entrée, une variable de sortie ou les deux. Ce n'est pas obligatoire, mais c'est fortement conseillé de s'en servir. Sans ça, il est beaucoup plus difficile de comprendre le code, et de le sécuriser (pour savoir si on a modifié une variable alors qu'on n'aurait pas dû par exemple).

7.3 transmission d'une procédure en argument

Supposons que nous ayons écrit un sous programme dont l'un des arguments représente une fonction (le sous-programme utilise la fonction passée en argument). On suppose que le sous-programme est inclus dans un module. Lors de l'appel du sous-programme depuis le programme principale, le compilateur ne pourra pas détecter que l'un des arguments est une fonction s'il n'a pas été déclaré comme tel. Pour résoudre ce problème, on utilise une interface contenant l'en-tête de la fonction ainsi que les déclarations relatives aux arguments.

```
program integration

! Calcul numerique d'integrales a l'aide des methodes
! composites des trapezes et de simpson

use mod_integrale

implicit none

real :: res1, res2, Pi

interface                                ! interface obligatoire
  function f1 (x)
    real, intent(in) :: x
    real :: f1
  end function
  function f2 (x)
    real, intent(in) :: x
    real :: f2
  end function
end interface

Pi = 4.0*atan(1.0)

! Appels de la subroutine sans utiliser les noms clés
! l'ordre des arguments est important
print*, '_'
call integrale (0.0, 1.0, 100, f1, res1, res2)
print*, 'Resultat_de_la_premiere_integrale:_:'
print*, 'Methode_des_trapezes:_:', res1
print*, 'Methode_de_simpson:_:', res2

! Appels de la subroutine en utilisant les noms clés
! l'ordre des arguments n'est pas important
print*, '_'
call integrale (fonction=f2, trapeze=res1, simpson=res2, &
               deb=Pi/3.0, fin=Pi/3.0, nb_int=200)
print*, 'Resultat_de_la_seconde_integrale:_:'
print*, 'Methode_des_trapezes:_:', res1
print*, 'Methode_de_simpson:_:', res2

end program integration
```

```

function f1 (x)                                ! premiere fonction a integrer
  real, intent(in) :: x
  real :: f1
  f1 = x*x+1
end function

function f2 (x)                                ! seconde fonction a integrer
  real, intent(in) :: x
  real :: f2
  f2 = sin(x)**2.
end function

  avec le module
module mod_integrale

implicit none

contains

subroutine integrale (deb, fin, nb_int, fonction, trapeze, simpson)

  implicit none
  real, intent(in) :: deb, fin                ! bornes d'integration
  real, intent(out) :: trapeze, simpson       ! methodes d'integration
  integer, intent(in) :: nb_int               ! nb d'intervalles
  real :: fonction

  real :: pas                                ! pas d'integration
  integer :: i

  pas = (fin - deb)/nb_int

  ! methode des trapezes
  trapeze = pas * ( 0.5*(fonction(deb)+fonction(fin)) + &
    sum( (/ (fonction(deb+i*pas), i = 1,nb_int-1) /) ) )

  ! methode de simpson
  simpson = pas/3.0 * ( fonction(deb) + fonction(fin) + &
    2.0*sum( (/ (fonction(deb+i*pas), i = 2, nb_int-2, 2) /) ) + &
    4.0*sum( (/ (fonction(deb+i*pas), i = 1, nb_int-1, 2) /) ) )

end subroutine integrale

end module mod_integrale

```

8 Vers les objets : les types dérivés

On appelle type dérivé la définition d'un nouveau type de variable, sorte d'objet contenant un nombre et un type de variable arbitraire.

Pour utiliser des variables du type dérivé *nom_type* précédemment défini, on déclare simplement :

```
type (nom_type) [,liste_attributs] :: nom_var
```

Remarque : Pour accéder aux champs de l'objet que nous venons de définir, nous utiliserons le caractère pourcent « % » (équivalent du point pour Python par exemple). En supposant qu'un champ *position* existe dans un objet *planete*, on y fait appel via


```

type (planete) :: terre
terre%position = 0.0

```

Exemple : Supposons qu'on veuille définir un nouveau type *animal* dans lequel seraient recensés la race, l'âge, la couleur et l'état de vaccination.

```

type animal
  character (len=20) :: race
  real :: age
  character (len=15) :: couleur
  logical, dimension(8) :: vaccination
end type animal

```

On peut ensuite manipuler globalement l'objet *animal* ou accéder à chacun de ses champs grâce à l'opérateur « % ».

9 Astuces et petits bouts de code

9.1 Lire un fichier de longueur inconnue

Pour lire un fichier dont on ne connaît pas le nombre de lignes :

```

implicit none

integer, parameter :: NMAX = 200 ! max number of messages

integer :: error
integer, dimension(NMAX) :: length
character(len=80), dimension(NMAX) :: message

open(14, file='message.in', status=old)
do
  read (14, '(i3,lx,i2,lx,a80)', iostat=error) j, length(i), message(i)
  if (error /= 0) exit
end do
close(14)

```

Remarque : La fonction `trim()` permet de supprimer les espaces en trop, pratique pour afficher ou écrire du texte proprement.

10 Optimisation

10.1 Comparaison f77/f90

10.2 Profiling

Avant d'essayer de rendre le code plus rapide, il faut en premier lieu trouver quelles parties du code le ralentissent et dans lesquelles il va être rentable de passer le temps d'optimisation.

Profilier le temps d'exécution d'un code fortran peut être fait en ajoutant l'option de compilation `-pg` :

```
gfortran -g -pg -o myprog myprog.f90
```

Remarque : Certains compilateurs, en particulier les vieux, ne vont pas autoriser les options d'optimisation si l'option `-g` est présente, et même s'ils optimisent, ça pourrait donner des résultats de profiling flous et difficiles à interpréter.

Il vaut donc mieux éviter les options d'optimisation quand on compile pour profiler.

Ensuite, exécutez normalement votre programme

```
./myprog
```

mais il faut quand même faire attention à avoir les droits d'écriture dans le dossier d'exécution afin de pouvoir écrire le fichier de résultat du profiling, appelé **gmon.out**.

Vous pouvez maintenant avoir les information de profiling en utilisant **gprof** :

```
gprof myprog gmon.out
```

Remarque : Si vous voulez une analyse ligne par ligne plutôt que fonction par fonction, il vous suffit (tant que vous avez compilé le programme avec l'option **-g**), d'utiliser l'option **-line** :

```
gprof --line myprog gmon.out
```

L'exécution de **gprof** va produire deux parties d'output, un *flat profile* et un *call graph*.

10.2.1 flat profile

Le flat profile ressemble à ça :

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
32.08	0.17	0.17	1488	0.11	0.14	__algo_mvs_MOD_mdt_mvs
28.30	0.32	0.15	1	150.01	530.02	MAIN__
16.98	0.41	0.09	477	0.19	0.34	__algo_radau_MOD_mdt_ra15
13.21	0.48	0.07	7254	0.01	0.01	__forces_MOD_mfo_grav
7.55	0.52	0.04	1634	0.02	0.02	__algo_mvs_MOD_mfo_mvs
1.89	0.53	0.01	1428	0.01	0.01	__drift_MOD_drift_kepu_lag
0.00	0.53	0.00	61641	0.00	0.00	__drift_MOD_drift_kepu_stumpff

Ce dernier liste les sections du programme (que ce soit les fonctions ou les lignes, selon que vous utilisez ou non l'option **-line**) par ordre du temps CPU utilisé, le premier étant celui qui en utilise le plus. Chaque ligne vous donne :

1. le pourcentage de CPU utilisé par cette section
2. le temps cumulé utilisé par cette section et toutes celles qui sont en dessous dans la liste
3. le nombre de seconde passés dans cette section

si cette section est une fonction :

- (a) le nombre d'appel de cette fonction dans le programme
 - (b) le temps moyen passé dans la fonction, par appel
 - (c) le temps moyen passé dans la fonction, ou une des fonctions qu'elle appelle, par appel
4. le nom de la fonction.

Remarque : Si vous avez compilé avec l'option **-g** et l'option **-line** pour **gprof**, alors le nom de la fonction inclura aussi le nom du fichier source et le numéro de ligne.

Le **flat profile** est suivi d'une explication de ce que signifie chaque champ.

10.2.2 call graph

Le **call graph** fournit les mêmes informations que le **flat profile**, mais organisation de façon à ce qu'il reflète la structure du programme.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 1.89% of 0.53 seconds

index	% time	self	children	called	name
				2	MAIN__ [1]
		0.15	0.38	1/1	main [2]
[1]	100.0	0.15	0.38	1+2	MAIN__ [1]
		0.17	0.05	1488/1488	__algo_mvs_MOD_mdt_mvs [3]
		0.00	0.16	1/1	mxx_sync.1545 [4]
		0.00	0.00	35/35	__algo_mvs_MOD_mco_mvs2h [13]
		0.00	0.00	1/1	__algo_mvs_MOD_mco_h2mvs [14]
		0.00	0.00	1488/1488	__dynamic_MOD_mce_cent [28]
		0.00	0.00	67/67	__mercury_outputs_MOD_mio_ce [35]
		0.00	0.00	54/54	__utilities_MOD_mioSpl [36]
		0.00	0.00	33/33	__mercury_outputs_MOD_mio_dump [37]
		0.00	0.00	32/32	__mercury_outputs_MOD_mio_out [38]
		0.00	0.00	14/15	__system_properties_MOD_mce_hill [41]
		0.00	0.00	14/14	__system_properties_MOD_mxx_ejec [42]
		0.00	0.00	6/6	__orbital_elements_MOD_mco_el2x [44]
		0.00	0.00	4/18	__system_properties_MOD_mxx_en [40]
		0.00	0.00	2/2	__mercury_outputs_MOD_mio_log [45]
		0.00	0.00	1/1	__system_properties_MOD_mce_init [47]
				2	MAIN__ [1]

					<spontaneous>
[2]	100.0	0.00	0.53		main [2]
		0.15	0.38	1/1	MAIN__ [1]

		0.17	0.05	1488/1488	MAIN__ [1]
[3]	40.6	0.17	0.05	1488	__algo_mvs_MOD_mdt_mvs [3]
		0.04	0.00	1490/1634	__algo_mvs_MOD_mfo_mvs [8]
		0.00	0.01	22320/25560	__drift_MOD_drift_one [10]

10.3 Des opérations équivalentes ne s'exécutent pas forcément avec la même rapidité

Dans l'exemple d'un programme qui fait X à la puissance Y , pour un entier Y , utilisons trois façons différentes de coder :

1. en utilisant la fonction power avec un Y constant

A=PI

```
call CPU_TIME(start_time)
do I=1,MAXLOOPS
  B=A**4
end do
call CPU_TIME(stop_time)
write(*,*) "A**4_Duration=",stop_time - start_time
```

2. en utilisant la fonction power mais avec une variable comme exposant.

A=PI

N=4

```
call CPU_TIME(start_time)
do I=1,MAXLOOPS
  B=A**N
```

```

end do
call CPU_TIME(stop_time)
write(*,*) "A**N_(N=4)_Duration=", stop_time - start_time

```

3. en multipliant X par lui même Y fois.

A=PI

```

call CPU_TIME(start_time)
do I=1,MAXLOOPS
    B=A*A*A*A
end do
call CPU_TIME(stop_time)
write(*,*) "A*A*A*A_Duration=", stop_time - start_time

```

\nInteger powers tests:

```

A**4 Duration=          1.3827890000000000
A**N (N=4) Duration=    2.9805470000000005
A**pN (pointer) Duration= 2.9005589999999994
A*A*A*A Duration=      1.2208150000000009

```

$$A^n \Rightarrow \underbrace{A * \dots * A}_{n \text{ fois}} \quad (10.1)$$

Il vaut donc mieux élever un nombre à une puissance entière par multiplication répétée (ou multiplication répétée suivie de division par 1.0 pour les puissances d'entiers négatifs.

Remarque : D'autres opérations pour lesquelles il y a de multiples façon de programmer auront le même comportement. Si on a un doute sur la technique à utiliser, on peut toujours modifier les sources fournis pour créer un set de tests pour un autre type d'opération.

10.4 Use Lookup Tables for Common Calculations

Le code ci-dessous consiste à comparer de manière intensive une valeur calculée de $\sin\left(\frac{\pi}{4}\right)$ dans une boucle, en utilisant une valeur pré-calculée stockée dans un tableau ("lookup table").

Tout d'abord on crée les tables de données :

```

do i=1,10
    pi_table(i)=pi/i
    sine_table(i)=sin(pi_table(i))
end do

```

utilisons trois façons différentes de coder :

1. en calculant intégralement à chaque fois :

```

call cpu_time(start_time)
do i=1,maxloops
    b=i*sin(pi/4.0)
end do
call cpu_time(stop_time)
write(*,*) "Repeated_pi/4_&sine, _Duration=", stop_time - start_time

```

2. en utilisant la table d'arguments :

```

call cpu_time(start_time)
do i=1,maxloops
    b=i*sin(pi_table(4))
end do
call cpu_time(stop_time)
write(*,*) "Repeated_sine, _Duration=", stop_time - start_time

```

3. en utilisant une table de sinus :

```
call cpu_time(start_time)
do i=1,maxloops
  b=i*sine_table(4)
end do
call cpu_time(stop_time)
write(*,*) "All_lookups ,_Duration=____",stop_time - start_time
```

On peut constater que le pré-calcul de $\pi/4$ fait peu de différence, mais éviter l'évaluation répétée de la fonction sinus (très consommatrice en temps) entraîne un temps de calcul 40 fois plus court. Dans cet exemple, le temps pris par la construction de la table de valeur est beaucoup moins grand que le temps économisé par le fait de se servir de la table.

```
Repeated pi/4 & sine, Duration=      2.99960000000000226E-002
Repeated sine, Duration=      0.16397499999999998
All lookups, Duration=      1.29980000000000095E-002
```

Remarque : Dans d'autres situation, la différence pourrait ne pas être aussi sensible. Cependant, il peut être parfois très intéressant de faire calculer à un autre programme une table de valeur que le programme pourra ensuite lire dans un fichier.

Quand c'est applicable, il peut être aussi intéressant de produire une table de valeur et ensuite d'interpoler la fonction entre deux valeurs tabulées pour calculer l'image par la fonction au lieu d'évaluer cette dernière à chaque fois.

10.5 Minimiser les sauts dans l'adressage mémoire

Cette section a pour but d'illustrer comment la manière dont est construit l'ordinateur peut influencer l'exécution de votre programme.

Tous les ordinateurs modernes ont trois types de mémoire :

- de la mémoire cache (quelques Mio)
- de la RAM (environ 4 Gio)
- de la mémoire tampon (*swap*) écrite sur disque dur (quasiment autant qu'on veut)

Chacun de ces types de mémoire peut être considéré un ordre de magnitude plus lent en terme de temps d'accès par rapport au type indiqué au dessus de lui dans la liste. Idéalement on veut donc utiliser la mémoire cache au maximum, et éviter autant que possible d'avoir recours à la mémoire tampon (ou *swap*). Malheureusement, vous n'avez aucun contrôle sur l'endroit où sont stockées les informations, c'est l'ordinateur qui gère les déplacements de fichiers de l'une à l'autre des types de mémoire.

À cause de ce comportement, on ne veut pas continuellement être en train de transférer des informations dans la mémoire cache parce que des informations logiques se trouvent à des emplacements physiques éloignés dans la mémoire. Il est donc important de comprendre comment est agencée la mémoire en fortran, en particulier dans les tableaux. C'est ce qu'illustre l'exemple suivant, par l'utilisation d'un grand tableau à deux dimension où on fait des opérations sur les éléments.

Même si **fortran** autorise l'utilisation de tableaux multi-dimensionnels, l'espace mémoire de l'ordinateur n'a qu'une seule dimension. [FIGURE 5] montre comment est représenté un simple tableau 3x3 $a(i,j)$.

On constate alors que c'est l'indice i qui varie le plus rapidement.

Si on a un tableau de grande dimension, alors faire varier le deuxième indice de 1 va entraîner la mise en mémoire cache d'une autre page de données. Il est donc important de mettre les boucles dans le bon ordre sous peine de ralentir considérablement les calculs.

Remarque : Dans la pratique, ce n'est pas si important que ça à partir du moment où on active les options d'optimisation qui vont se charger d'inverser l'ordre des boucles lors de la compilation. Mais c'est toujours bon de le savoir, ne serait-ce que pour certains cas particuliers.

On a donc les deux cas de figure suivant :

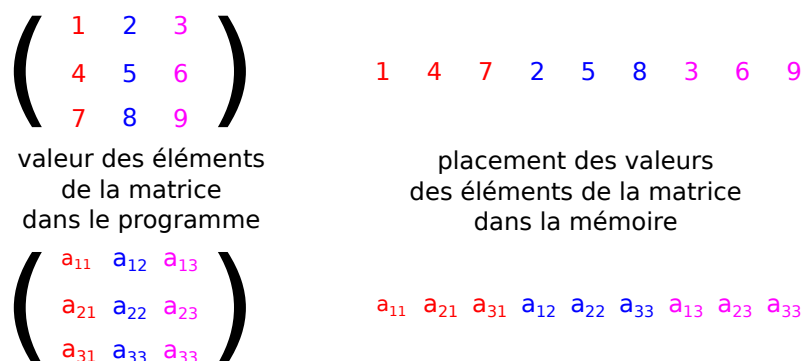


FIGURE 5 – Représentation de l'agencement en mémoire des valeurs associées aux éléments d'une matrice. L'élément a_{ij} (ligne i et colonne j) est noté $a(i, j)$ en **fortran 90**

- on fait varier d'abord l'indice des lignes puis pour chaque ligne on parcourt toutes les colonnes

```
call cpu_time(start_time)
do i=1,maxloops2
  do x=1,2000
    do y=1,2000
      matrix(x,y)=x*y
    end do
  end do
end do
call cpu_time(stop_time)
write(*,*) "x_outer, y_inner, Duration=", stop_time - start_time
```

- on fait varier d'abord l'indice des colonnes puis pour chaque colonne on parcourt toutes les lignes

```
call cpu_time(start_time)
do i=1,maxloops2
  do y=1,2000
    do x=1,2000
      matrix(x,y)=x*y
    end do
  end do
end do
call cpu_time(stop_time)
write(*,*) "y_outer, x_inner, Duration=", stop_time - start_time
```

```
x outer, y inner, Duration=      0.52391999999999994
y outer, x inner, Duration=      0.14897799999999994
```

Imbriquer les boucles dans le mauvais ordre ralentit le code par un facteur 3 environ. Il faut donc que l'indice qui varie le plus vite soit l'indice des lignes.

10.6 Utiliser les options d'optimisation du compilateur

L'astuce la plus simple pour augmenter la vitesse d'exécution d'un programme est de loin d'utiliser les options d'optimisation du compilateur.

L'option `-O2` est une sorte de meta option qui regroupe plein d'options, c'est celle qu'il faut activer en priorité. Dans mon cas, j'ai eu un gain de temps supérieur à 30% (mais ça dépend fortement du programme).

Remarque : Avec l'option `-O2`, il n'y a plus de différence de temps entre les deux ordres de boucles pour [§ 10.5].

Une autre option qui peut augmenter parfois la rapidité d'un programme est `-funroll-loops`

11 Avancé

11.1 Les pointeurs

Un pointeur, ce n'est ni plus ni moins qu'une variable qui contient une adresse mémoire.

Ceci étant dit, on définit différents types de pointeurs, selon la nature des objets vers lesquels on pointe. L'utilité d'une telle chose est de pouvoir faire de l'arithmétique avec les pointeurs, vu que l'on connaît la taille mémoire de chaque objet qu'on manipule.

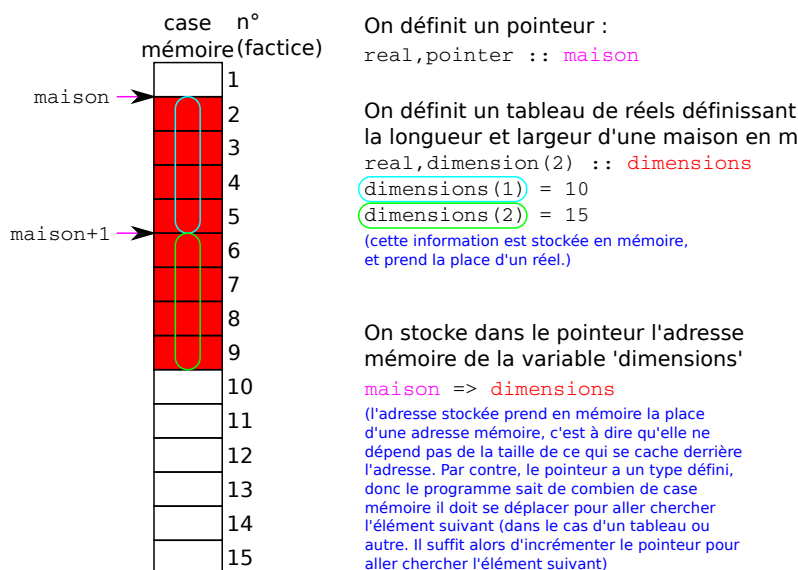


FIGURE 6 – Représentation schématique du fonctionnement d'un pointeur avec représentation de la mémoire. Les numéros pour les cases mémoire ne représentent pas la réalité, c'est juste pour montrer comment on fait référence à une case mémoire. L'idée est de montrer qu'en manipulant des adresses au lieu de manipuler les contenus, on peut faire des choses beaucoup plus puissantes.

On définit des pointeurs à l'aide d'attributs. Que ce soit pour définir les cibles que l'on pourra pointer

```
real, target :: a, b(1000), c(10,10)
```

ou pour définir les pointeurs eux mêmes (qui devront bien évidemment avoir le même type que les cibles auxquelles ils seront associés ultérieurement)

```
real, pointer :: pa, pb, pc
```

Pour attribuer une adresse au pointeur, il suffit ensuite de faire

```
pa => a
```

Après cette assignation, on peut alors faire

```
b(i) = pa * c(i,i)
```

qui est équivalent en terme de résultat avec

```
b(i) = a * c(i,i)
```

On peut aussi faire

```
pa = 1.23456
```

```
write(*,*) 'a = ', a
```

et avoir "1.23456" comme résultat affiché parce que changer *pa* change *a*

Le pointeur peut être ré-associé à n'importe quel moment. Il peut aussi être forcé à ne pointer sur rien en faisant :

```
nullify(pa)
```

11.2 Attributs des variables lors de leur déclaration

Lors de leur déclaration, il existe divers attributs que l'on peut donner aux variables et qui permettent de préciser à quoi elles vont servir notamment.

11.2.1 Une constante : `parameter`

On peut ainsi définir une variable comme étant un paramètre, c'est à dire que cette dernière ne pourra pas être modifiée dans le reste du programme. C'est pratique pour définir des constantes comme la valeur de π par exemple :

```
real, parameter :: pi = 3.14159
```

11.2.2 Entrée ou sortie : `intent`

Il est possible en Fortran 90 d'associer des attributs aux arguments des procédures pour fiabiliser les programmes. Les principaux attributs sont :

- **intent(in)**. Cet attribut signifie que la variable à laquelle il se rapporte est un argument d'entrée de la procédure; sa valeur ne peut pas être modifiée par la procédure.
- **intent(out)**. L'argument de la procédure a l'attribut sortie; la procédure ne peut pas utiliser sa valeur. Elle doit en revanche lui en attribuer une.
- **intent(in/out)**. L'argument correspondant est à la fois un argument d'entrée (la procédure peut utiliser sa valeur) et un argument de sortie (la procédure doit lui en attribuer une nouvelle).

Si les conditions précédentes ne sont pas respectées, une erreur surviendra à la compilation. La syntaxe des attributs est la suivante :

```
subroutine nom(a, b, c)

implicit none
integer, intent(in) ::    a           ! argument d'entree
real, intent(out) ::     b           ! argument de sortie
logical, intent(inout) :: c         ! argument d'entree et de sortie

[ ... ]                ! bloc d'instructions

end subroutine nom
```

11.3 Débuger des programmes fortran avec `gdb`

Afin d'utiliser un débugeur comme *gdb* pour suivre l'exécution d'un programme fortran, il est nécessaire de le compiler avec l'option `-g`, par exemple :

```
f77 -g foo.f -o foo
```

La commande suivante va créer un exécutable **foo** que vous pouvez exécuter normalement ou à travers **gdb** pour suivre ce qu'il fait au fur et à mesure.

Pour commencer l'exécution du programme **foo** avec **gdb** il faut :

1. faire précéder le nom du programme par **gdb** :

```
gdb foo
```

Vous aurez alors une ligne de commande de la forme

```
(gdb)
```

2. entrez ces commandes dans le prompt (gdb) :

```
break main
run
```

Ceci lancera l'exécution du programme, puis cette dernière sera mise en pause juste avant la première commande exécutable.

Une chose intéressante à connaître est la séquence exacte d'exécution du programme, en particulier à travers les boucles et les tests conditionnels. Si le programme n'est pas trop gros, vous pouvez suivre facilement ce chemin en exécutant les lignes de code une à une.

Pour exécuter la ligne suivante, il faut entrer dans le prompt (**gdb**) :

```
step
```

À chaque fois que vous entrez la commande **step**, **gdb** va alors afficher la ligne qui est sur le point d'être exécuter, avec le numéro de ligne à gauche. Ceci permet de savoir ce qu'il va se passer, avant que ça ne se passe réellement.

Pour quitter **gdb**, entrez la commande suivante dans le prompt :

```
quit
```

Vous aurez alors le message suivant :

```
The program is running.  Quit anyway (and kill it)? (y or n)
```

Entrez 'y' pour confirmez que vous souhaitez quitter **gdb**.

11.3.1 Savoir où on se trouve dans le programme

Pour savoir où on est, il suffit d'entrer dans le prompt (**gdb**) :

```
where
```

Cette commande affiche alors le numéro de ligne de la ligne courante. par exemple quelque chose du genre :

```
#0 foo () at foo.f:12
```

indique que l'exécution du programme est actuellement au niveau de la ligne 12 du code source du fichier **foo.f**

Vous pouvez afficher quelques lignes du code source autour de la position actuelle via

```
list
```

Il est aussi possible, via cette commande, de spécifier une liste de lignes à afficher. Par exemple pour lister les lignes 10 à 24 du programme courant, vous devez entrer dans le prompt (**gdb**) :

```
list 10,24
```

11.3.2 Afficher le contenu d'une variable fortran avec gdb

À n'importe quel moment de l'exécution pas à pas du programme, vous pouvez connaître les valeurs courantes de vos variables en utilisant la commande **print**. Par exemple, si vous avez une variable **density**, vous pouvez entrer la commande suivante afin de connaître la valeur stockée :

```
print density
```



Vous devez entrer les noms des variables en minuscules dans **gdb**, sans vous préoccuper de la casse de la variable dans votre code source.

11.3.3 Mettre le programme en pause à un endroit particulier

Au lieu d'entrer

```
break main
```

il faut entrer une commande du style

```
break [file:]function
```

où **[file:]** est un argument optionnel qui permet de spécifier dans quel fichier se trouve la fonction considérée (s'il y a plusieurs fichiers) et **function** est le nom de la fonction au début de laquelle on veut mettre l'exécution en pause.

11.3.4 Débuggage avancé

Pour exécuter le programme ligne à ligne, il existe **next** et **step**.

1. **step** permet d'exécuter la ligne suivante du programme tout en passant *au-dessus* de tout appel de fonction dans la ligne.
2. **next** permet d'exécuter la ligne suivante du programme, en exécutant aussi tous les appels de fonctions de la ligne.

Pour continuer l'exécution (jusqu'au prochain breakpoint je suppose) il faut utiliser la commande suivante dans le prompt (**gdb**) :

```
c
```

11.4 Erreurs de compilation

11.4.1 Utilisation de fonctions internes à un module

```
kepler_equation.o: In function '__kepler_equation_MOD_orbel_fhybrid':
kepler_equation.f90:(.text+0x25f): undefined reference to 'orbel_flon_'
collect2: ld a retourné 1 code d'état d'exécution
```

Le problème vient de la fonction **orbel_fhybrid** du module **kepler_equation**. j'ai en effet pris des fonctions pour en faire un module, et avant, des variables étaient définies dans la fonction, puisque celle-ci faisait appel à d'autres fonctions. Toutes ces fonctions faisant maintenant partie du même module, on peut et on doit enlever ces définitions.

Pour résoudre le problème, j'ai donc supprimé la ligne :

```
real*8 orbel_flon
```

dans la définition de la fonction **orbel_fhybrid**.

11.4.2 Utilisation de fonctions d'un module

```
mercury.f90:1140.22:
```

```
character*8 mio_re2c, mio_fl2c
1
```

```
Error: Symbol 'mio_re2c' at (1) already has basic type of CHARACTER
```

Il faut supprimer cette ligne parce que le type de la fonction est déjà défini dans le module, pas besoin de redéfinir le type de la fonction dans la subroutine qui importe le module.

11.4.3 Utilisation de subroutine en paramètre d'autres subroutines

```
mercury.f90:137.19:
```

```
external mco_dh2h,mco_h2dh
1
```

```
Error: Cannot change attributes of USE-associated symbol at (1)
```

En mettant ces subroutines (celles qu'on appelle en argument) dans un module, pas besoin de les définir via un **external**. Il faut donc supprimer cette ligne.

11.4.4 Function has no implicit type

```
test_mfo_user.f90:35.12:
```

```
f_p = get_F(p)
1
```

```
Error: Function 'get_f' at (1) has no IMPLICIT type
```

J'ai eu cette erreur parce que je n'avais pas rajouté le module dans la ligne de compilation

```
gfortran -o test test_mfo_user.f90 user_module.o
```

J'ai eu aussi cette erreur parce que dans le module `user_module`, la fonction `get_F` était une fonction privée.