

Table des matières

1	Préambule	3
2	Compilation	3
3	Transition fortran 77/fortran 90	4
3.1	Instructions obsolètes ou dépréciées	4
3.2	Comparaison f77/f90	4
4	Les bases	5
4.1	Éléments de syntaxe	5
4.2	Déclaration de variables	6
4.3	Afficher et lire des informations (entrée et sortie standard)	6
4.4	Les expressions arithmétiques	7
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	8
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>	9
4.7.4	La boucle <i>tant que</i>	10
4.7.5	Les instructions <i>exit</i> et <i>cycle</i>	10
4.8	Les formats de lecture et écriture	11
5	Les tableaux	13
5.1	Déclaration des tableaux	13
5.2	Les opérations relatives aux tableaux	13
5.2.1	Affectation collective	13
5.2.2	Les expressions tableaux	14
5.2.3	Initialisation des tableaux à une dimension	14
5.2.4	Les sections de tableau	15
5.2.5	Les fonctions portant sur des tableaux	16
5.3	L'instruction <i>where</i>	16
5.4	Les tableaux dynamiques	17
6	Les Modules	18
6.1	Struture générale	19
6.2	Appel des modules depuis un programme principal	19
6.3	Accès à tout ou partie d'un module	20
6.3.1	Protection interne	20
6.3.2	Protection externe	21
6.4	Partage de données et variables	21
7	Les Procédures (fonction et subroutine)	21
7.1	fonctions	22
7.2	Subroutine	23
7.3	transmission d'une procédure en argument	23
8	Vers les objets : les types dérivés	25
8.1	Définir un type dérivé	25
8.2	Initialisation lors de la déclaration	25

9	Astuces et petits bouts de code	26
9.1	Allouer une variable dynamique dans une subroutine	26
9.2	Lire un fichier de longueur inconnue	27
9.3	Lire un fichier de paramètres	27
9.4	Initialisation implicite d'un tableau	29
9.5	Remplissage d'un tableau dynamique de taille variable	29
10	Les fonctions intrinsèques en Fortran 90	30
10.1	Les fonctions numériques	30
10.1.1	Les fonctions numériques élémentaires	30
10.1.2	Les fonctions mathématiques élémentaires	32
10.1.3	Les fonctions d'interrogation	33
10.2	les fonctions relatives aux chaînes de caractères	34
10.2.1	Les fonctions élémentaires	34
10.2.2	Les fonctions de comparaison de chaînes	34
10.2.3	Les fonctions de manipulation de chaînes	34
10.3	Les fonctions relatives aux tableaux	36
10.4	Procédures diverses	38
11	Avancé	39
11.1	Les pointeurs	39
11.2	Attributs des variables lors de leur déclaration	40
11.2.1	Une constante : parameter	40
11.2.2	Entrée ou sortie : intent	40
11.3	Optimisation	41
11.3.1	Comparaison f77/f90	41
11.3.2	Profiling	41
11.3.3	Des opérations équivalentes ne s'exécutent pas forcément avec la même rapidité	42
11.3.4	Use Lookup Tables for Common Calculations	43
11.3.5	Minimiser les sauts dans l'adressage mémoire	44
11.3.6	Utiliser les options d'optimisation du compilateur	45
11.4	Débugger des programmes fortran avec gdb	45
11.4.1	Débuguer à l'aide d'un core dumped	46
11.4.2	Savoir où on se trouve dans le programme	48
11.4.3	Afficher le contenu d'une variable fortran avec gdb	48
11.4.4	Mettre le programme en pause à un endroit particulier	48
11.4.5	Débuggage avancé	48
11.5	Erreurs de compilation	49
11.5.1	Utilisation de fonctions internes à un module	49
11.5.2	Utilisation de fonctions d'un module	49
11.5.3	Utilisation de subroutine en paramètre d'autres subroutines	49
11.5.4	Function has no implicit type	49
11.5.5	Error : Rank mismatch in array reference at (1) (2/1)	50
11.5.6	Error : Rank mismatch in argument 'levels' at (1) (1 and 0)	50

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.

Ce tutoriel, même s'il n'en est pas qu'une pâle copie est largement inspiré du tutoriel de Michel Dobrijevic pour certaines parties où je ne pouvais de tout façon pas mieux expliquer qu'il ne l'avait déjà fait. C'est avec ce tutoriel que j'ai appris **fortran 90**

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, p3]

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

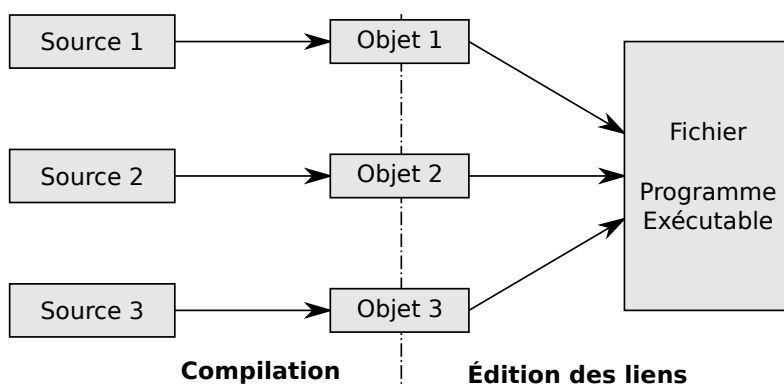


FIGURE 1 – La compilation de tous les fichiers source doit se faire avant l'édition des liens pour créer le fichier exécutable.

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, p4]


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



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 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

3.2 Comparaison f77/f90

En fortran 77, voici les temps d'exécution :

```

arguin.login> gfortran -o timings timings.f
arguin.login> ./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=          14.059861999999995
Y outer, X inner, duration=          5.0562309999999968

```

En fortran 90, en adaptant simplement le code :

```

arguin.login> gfortran -o timings timings.f90
arguin.login> ./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 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=      1.12283000000000004
\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 :

```

1 | print *, 'Montant HT :', montant_ht, &
2 |   'TVA:', tva, '&'
3 |   'Montant TTC :', montant_ttc

```

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

```

1 | print *, 'Entrez un nombre entier &
2 |   &compris entre 100 & 199'

```

Les commentaires commencent par le symbole « ! » :

```

1 | if (n < 100 .or. n > 199) ! Test cas d'erreur
2 | ! On lit l'exposant
3 | read *,x
4 | ! On lit la base
5 | read *,y
6 | if (y <=0) then ! Test cas d'erreur
7 |   print *, 'La base doit etre un nombre > 0'
8 | else
9 |   z = y**x ! On calcule la puissance
10 | 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 :

```
1 | 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 :

```
1 | program declaration
2 |
3 | implicit none
4 | integer ::          i, j=5           ! type entier
5 | real ::             var, x=2.5       ! type reel simple precision
6 | double precision :: plus_precis     ! type reel double precision
7 | logical ::          reussite        ! type logique
8 | character (10) ::   mot             ! type caractere
9 | complex ::          z = (1.2, 20)   ! type complexe
10 |
11 | [...]                          ! bloc d'instructions executables
12 |
13 | 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.

```
1 | program commentaires
2 |
3 | implicit none
4 | integer :: i, &           ! indice de boucle sur le temps
5 |           j, &           ! nombre de niveaux
6 |           k               ! indice de boucle sur les niveaux
7 |
8 | [...]                    ! bloc d'instructions executables
9 |
10 | 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.

```
1 | program ecriture_lecture
2 | implicit none
3 | real :: var, &
4 |       lu ! variable lue au clavier
5 |
6 | var = 2.5
7 |
8 | print*, 'La variable var vaut : ', var
9 | print*, 'Entrez une valeur au clavier'
10 | read*, lu
11 | print*, 'La valeur entree au clavier est', lu
```

```
12 |
13 | end
```

4.4 Les expressions arithmétiques

On retrouve les opérateurs arithmétiques usuels : « + », « - », « * » et « / ». Ces opérandes ne sont définis à 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 → réel → 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 :

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

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 :

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

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

```
1 integer, parameter :: nb = 5
2 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 où le compilateur peut la calculer.

```
1 implicit none
2
3 integer, parameter :: nb = 5
4 integer, parameter :: nb_max = 2*nb+4
5 integer, parameter :: nb_min = 2*nb-4
6 integer, parameter :: nb_elem = nb_max - nb_min + 1
7
8 [...] ! bloc d'instructions exécutables
9
10 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 :

```
1 if (exp_log1) then
2     bloc1 ! bloc d'instructions
3 [else if (exp_log2) then
4     bloc2 ! bloc d'instructions
5 ]...
6 [else
7     blocn ! bloc d'instructions
8 ]
9 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.

```
1 program nom_if
2
3 implicit none
4
5 integer :: i, j
6
7 read*, i, j
8
9 if (i < 0) then
```



```

10  print*, 'i est negatif'
11  else if (i > 0) then
12    print*, 'i est positif'
13  else
14    if (j < 0) then
15      print*, 'j est negatif'
16    else if (j > 0) then
17      print*, 'j est positif'
18    else
19      print*, 'i et j sont nuls'
20    end if
21  end if
22
23  end

```

4.7.2 L'instruction *select case*

La syntaxe générale est la suivante :

```

1  select case (exp_scal)
2  [case (selecteur1)
3    bloc1      ! bloc d'instructions
4  [case (selecteur2)
5    bloc2      ! bloc d'instructions
6  ]...
7  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).

```

1  program case
2
3  implicit none
4
5  character(3) :: reponse
6
7  print*, 'Voulez-vous continuer le programme ?'
8  read*, reponse
9
10 select case (reponse)
11   case ('oui')
12     print*, 'OK, ca roule...'
13   case ('non')
14     print*, 'Au revoir !'
15     stop
16   case default
17     print*, 'Veuillez repondre par "oui" ou par "non"'
18 end select
19
20 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 :

```

1 do var = debut, fin, [pas]
2   bloc      ! bloc d'instructions
3 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 :

```

1 do while (exp_log)
2   bloc      ! bloc d'instructions
3 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.

```

1 do while (.not.fini)
2   do var = debut, fin
3     bloc1      ! bloc d'instructions
4     if (exp_log) exit
5     bloc2      ! bloc d'instructions
6   end do
7   bloc3      ! bloc d'instructions
8 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**.

```

1 do while (.not.fini)
2   do var = debut, fin
3     bloc1      ! bloc d'instructions
4     if (exp_log) exit
5     bloc2      ! bloc d'instructions
6   end do
7   bloc3      ! bloc d'instructions
8 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.

```

1 do var = debut, fin
2   bloc1      ! bloc d'instructions
3   if (exp_log) cycle
4   bloc2      ! bloc d'instructions
5 end do

```

4.8 Les formats de lecture et écriture

Il est possible de lire ou écrire avec un format par défaut, « * ».

```
1 write(*,*) 'la valeur de var est : ',var
```

Mais on peut souhaiter utiliser des formats spécifiques, pour limiter le nombre de chiffres significatifs par exemple afin de rendre l'information plus lisible. Dans ce cas là, on utilisera plutôt :

```
1 write(*,'(a, es10.2)') 'la valeur de var est : ',var
```

où 'a' désigne une chaîne de caractère de longueur quelconque, et où 'es10.2' est un format d'affichage pour un réel. La signification exacte est expliqué dans [TABLE 1, p11].

Dans ce tableau, les lettres suivantes (en italique) ont une signification :

w : Le nombre de position à utiliser (l'espace à réserver pour l'affichage)

m : Le nombre minimum de position à utiliser

d : Le nombre de chiffres à utiliser à droite du point décimal.

e : Le nombre de chiffres à utiliser pour l'affichage de l'exposant.

Même si on peut afficher un nombre avec autant de positions qu'on veut, c'est juste une question d'affichage, ce nombre de positions n'a rien à voir avec la précision (i.e le nombre de chiffres significatifs).

But		Descripteurs de format	
Afficher/Lire des entiers		<i>Iw</i>	<i>Iw.n</i>
Afficher/Lire des réels	Format décimale	<i>Fw.d</i>	
	Format exponentielle	<i>Ew.d</i>	<i>Ew.dEe</i>
	Format Scientifique	<i>ESw.d</i>	<i>ESw.dEe</i>
	Format Ingénieur	<i>ENw.d</i>	<i>ENw.dEe</i>
Afficher/lire des booléens		<i>Lw</i>	
Afficher/lire des chaînes de caractères		<i>A</i>	<i>Aw</i>
Positionnement	Horizontal	<i>nX</i>	
	Tabulé	<i>Tc</i>	<i>TLc et TRc</i>
	Vertical	/	
Autre	Groupé	<i>r(...)</i>	
	Contrôle du format de scan	:	
	Contrôle du signe	<i>S, SP, et SS</i>	
	Contrôle des blancs	<i>BN et BZ</i>	

TABLE 1 – Liste des formats (je ne sais pas s'ils sont tous là, mais les plus courants en tout cas) d'affichage de variables. Ils sont utiles pour créer un format, qui doit être de la forme suivante '(F,F...)' où *F* est un format quelconque du tableau ci-dessus. Il est possible de dire qu'on plusieurs fois le même format en le préfixant par un nombre d'occurrence.

Voici ci-dessous un exemple d'utilisation dans un cas pratique, l'écriture d'un fichier de sortie **disk.out** qui écrit les différents paramètres d'un programme :

```

1 open(10, file='disk.out')
2
3 write(10,*) '_____ '
4 write(10,*) '| Properties of the disk |'
5 write(10,*) '_____ '
6 write(10,'(a,f4.2)') 'b/h = ',B_OVER_H
7 write(10,'(a,f4.2)') 'adiabatic index = ', ADIABATIC_INDEX
8 write(10,'(a,f4.2)') 'mean molecular weight = ', MEAN_MOLECULAR_WEIGHT
9 write(10,'(a,es10.1e2,a)') 'viscosity = ', viscosity, ' (cm^2/s)'
10 write(10,'(a,f6.1,a,f4.2 ,a)') 'initial surface density = ',INITIAL_SIGMA_0, ' * R^(-',
    ' ,INITIAL_SIGMA_INDEX,') (g/cm^2)'
11 write(10,*) ''
12 write(10,*) "Possible values : 'open', 'closed'"
13 write(10,*) 'inner boundary condition = ', trim(INNER_BOUNDARY_CONDITION)

```

```

14 write(10,*) 'outer boundary condition = ', trim(OUTER_BOUNDARY_CONDITION)
15 write(10,*) ''
16 write(10,'(a,f6.1,a)') 'inner edge of the disk = ', INNER_BOUNDARY_RADIUS, ' (AU)'
17 write(10,'(a,f6.1,a)') 'outer edge of the disk = ', OUTER_BOUNDARY_RADIUS, ' (AU)'
18 write(10,*) ''
19 write(10,*) 'Possible values : 0 for no dissipation, 1 for viscous dissipation and 2
    for exponential decay of the initial profile'
20 write(10,'(a,i1)') 'dissipation of the disk = ', DISSIPATION_TYPE
21 write(10,'(a,es10.1e2)') 'characteristic time for decay (only valid for exponential
    decay) = ', TAU_DISSIPATION
22 write(10,*) ''
23 write(10,*) '-----'
24 write(10,*) '| Interactions disk/planets |'
25 write(10,*) '-----'
26 write(10,*) 'when the inclination damping stops'
27 write(10,'(a,es10.1e2,a)') 'inclination cutoff = ', INCLINATION_CUTOFF, ' (rad)'
28 write(10,*) ''
29 write(10,*) "Possible values : 'real', 'mass_independant', 'mass_dependant'"
30 write(10,*) 'torque type = ', trim(TORQUE_TYPE)
31 write(10,*) ''
32
33 close(10)

```

qui donne le résultat suivant :

```

-----
| Properties of the disk |
-----
b/h = 0.40
adiabatic index = 1.40
mean molecular weight = 2.35
viscosity = 1.0E+15(cm^2/s)
surface density = 450.0 * R^(-0.50) (g/cm^2)

Possible values : 'open', 'closed'
inner boundary condition = closed
outer boundary condition = closed

inner edge of the disk = 1.0(AU)
outer edge of the disk = 100.0(AU)

Possible values : 0 for no dissipation, 1 for viscous
dissipation and 2 for exponential decay of the initial profile
dissipation of the disk = 1
characteristic time for decay (only valid for exponential decay) = -1.0E+00

-----
| Interactions disk/planets |
-----
when the inclination damping stops
inclination cutoff = 5.0E-04(rad)

Possible values : 'real', 'mass_independant', 'mass_dependant'
torque type = mass_dependant

```

Remarque : Vous pourrez remarquer que certaines lignes ont un espace au début, pas forcément très joli. C'est dû à l'utilisation d'un format *.

Afin de contourner ce problème, on doit spécifier explicitement le format. Au lieu d'utiliser :

```
write(10,*) '-----'
```

on doit donc utiliser

```
write(10,'(a)') '-----'
```

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 :

```

1 implicit none
2
3 integer, parameter :: min = -5
4 integer, parameter :: max = 12
5 integer, parameter :: nb = 10
6 integer, parameter :: nb1 = 5, nb2 = 3
7 real, dimension (nb) :: vect_1 ! tableau de rang 1
8 integer, dimension (min:max) :: vect_2 ! tableau de rang 1
9 real, dimension (50) :: vect_3 ! tableau de rang 1
10 integer, dimension (nb1, nb2) :: t ! tableau de rang 2
11 real, dimension (min:max, nb2) :: tab ! tableau de rang 2
12
13 [...] ! bloc d'instructions executables
14
15 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**

```

1 implicit none
2
3 integer :: i, j, k
4 integer, dimension (5, -3:2, 10) :: mat ! tableau de rang 3
5
6 do i = 1, 5
7   do j = -3, 2
8     do k = 1, 10
9       mat(i, j, k) = 1
10    end do
11  end do
12 end do
13
14 ! est equivalent a
15 mat = 1
16
17 ! ou mieux
18 mat(:, :, :) = 1
19
20 end

```

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

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

au lieu de

```
1| 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.

```
1| implicit none
2|
3| integer :: i
4| integer, parameter ::      dim = 12
5|
6| ! tableaux de rang 1
7| integer, dimension (dim) ::  a,b
8| integer, dimension (dim) ::  som, prod, racine
9|
10| ! tableaux de type logique
11| logical, dimension (dim) ::  compare
12|
13| do i = 1, dim
14|     som(i) = a(i) + b(i)
15|     prod(i) = a(i)*b(i)
16|     prod(i) = 2*prod(i) + 1
17|     racine(i) = sqrt(real(a(i)))
18|     if (som(i) < prod(i)) then
19|         compare(i) = .true.
20|     else
21|         compare(i) = .false.
22|     endif
23| end do
24|
25| ! En Fortran 90, les instructions precedentes
26| ! se simplifient de la maniere suivante :
27|
28| ! equivalent a : som = a + b
29| som(:) = a(:) + b(:)
30| prod(:) = a(:)*b(:)
31| prod(:) = 2*prod(:) + 1
32|
33| ! Attention les elements de a sont entiers
34| racine(:) = sqrt( real(a(:)) )
35| compare(:) = (som(:) < prod(:))
36|
37| 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*.

```
1| program initialisation
2|
3| implicit none
4|
5| integer ::      i
6| integer, parameter ::      n = 5
7| integer, dimension(n) ::  tab1, tab2, t      ! tableaux de rang 1
8| integer, dimension(n) ::  tab3 = (/1,2,3,4,0/)
```

```

9  real, dimension(0:9) ::      angle
10
11  tab1(1) = 3 ; tab1(2) = 5 ; tab1(3) = -2 ; tab1(4) = 4 ; tab1(5) = 202
12
13  tab2 = (/ 3, 5, -2, 4, 202/)
14
15  ! les affectations suivantes sont equivalentes
16  do i = 0, 90, 10
17      angle(i/10) = i*0.5
18  end do
19
20  angle = ((i*0.5, i=0, 90, 10)/)          ! boucle implicite
21
22  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 :

```
1 | 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'incréméntation (1 par défaut ; si le pas est négatif, la variation d'indice est rétrograde de *borne_sup* à *borne_inf*).

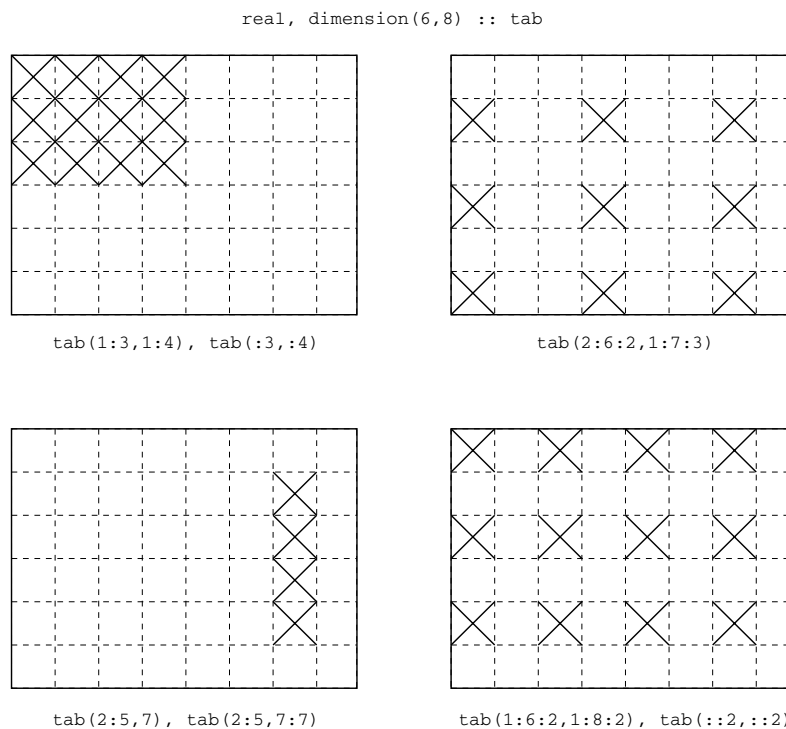


FIGURE 3 – Représentation de différentes sections de tableaux afin de montrer les possibilités de sélections.

Dans le programme suivant,

```

1 | integer, dimension (n) ::      w
2 | integer, dimension (p) ::      v
3 |
4 | v(:) = (/ (i,i=1,p)/)          ! boucle implicite
5 |
6 | ! EXEMPLE 1
7 |
8 | 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.

```

1 integer, dimension (n) :: w
2 integer, dimension (p) :: v, v1
3
4 v(2:9) = (v(1:8) + v(3:10))/2
5
6 ! est équivalent à
7 do i = 1, p
8   v1(i) = v(i)
9 end do
10 do i = 2, 9
11   v(i) = (v1(i-1) + v1(i+1))/2
12 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 :

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

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

```
1 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 :

```

1 where (inst_log_tab)
2   bloc1
3 [elsewhere
4   bloc2
5 ]
6 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 :

```
1 | 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).

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

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

```
1 | ! declaration d'un tableau dynamique d'entiers de rang 2
2 | real, dimension(:, :), allocatable :: matrice
3 | integer :: n, m, verif
4 |
5 | [...] ! bloc d'instructions exécutables
6 |
7 | ! lecture au clavier des étendues
8 | read*, n ! suivant la première dimension
9 | read*, m ! suivant la seconde dimension
10 |
11 | allocate(matrice(n, m))
```

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

```
1 | 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).

```
1 | deallocate(matrice) ! libération
```

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*).

```
1 | program tableau_dynamique
2 |
3 | implicit none
4 | ! declaration d'un tableau dynamique d'entiers de rang 2
5 | integer, dimension(:, :), allocatable :: matrice
6 |
7 | integer :: nb_lig, nb_col, i
8 |
9 | read*, nb_lig ! lecture au clavier du nombre de lignes
10 | read*, nb_col ! lecture au clavier du nombre de colonnes
11 |
12 | ! allocation d'un emplacement de profil nb_lig, nb_col
```

```

13 allocate (matrice(nb_lig, nb_col))
14
15 ! affectations des elements de la matrice
16 do i = 1, nb_col
17     matrice(:,i) = i
18 end do
19 call affiche(matrice)
20
21 ! liberation de l'emplacement correspondant au tableau matrice
22 deallocate (matrice)
23
24 contains
25
26 subroutine affiche(t)
27
28 implicit none
29 integer, dimension(:,,:), intent(in) :: t      ! dimensionnement automatique
30 integer :: i, j
31
32 do i = 1, size(t,1)
33     print*, (t(i,j), j=1,size(t,2))
34 end do
35
36 end subroutine
37
38 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

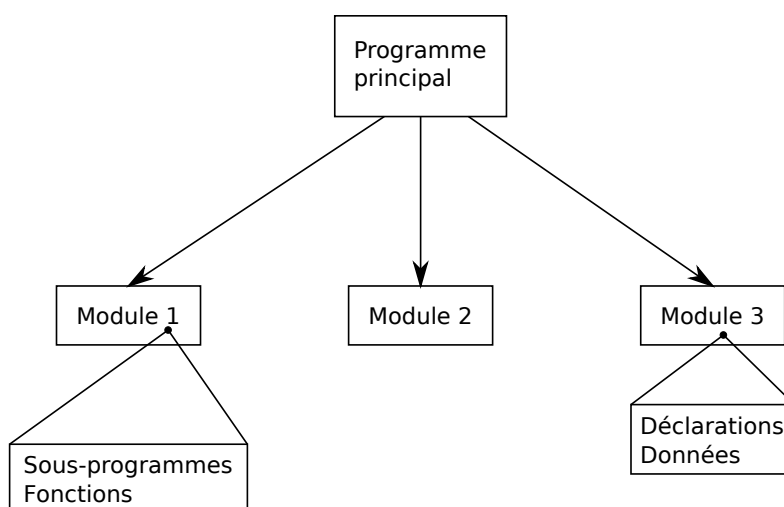


FIGURE 4 – Structure générale d'un programme Fortran 90

[FIGURE 4, p18] 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, p18], 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 Structure 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**.

```

1 module nom_module
2
3   implicit none
4   [...]           ! bloc de declarations
5
6   contains        ! obligatoire si suivi de procedures
7
8   [...]           ! suite de procedures
9
10 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.

```

1 program utilisation_module
2
3 use nom_module
4 use module constantes
5
6 implicit none
7 [...]           ! bloc de declarations
8
9 [call...]
10 [call...] ! appels des procedures contenues dans le module nom_module
11 [...]       ! bloc d'instructions executables
12
13 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.

```
1| 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 :

```
1| 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.

```
1| module module_atmosphere
2|
3| use module_constantes, only : k, Na
4|
5| real, private :: T = 300., &      ! temperature [K]
6|                m = 28.0e-3, &    ! masse mol. moy. [gmol-1]
7|                g = 9.81, &       ! pesanteur [ms-2]
8|                Po = 1.01325e5    ! pression standard [Pa]
9|
10| contains
11|
12| function pression(z)
13|
14|   real :: pression              ! pression a l'altitude z [Pa]
15|   real, intent(in) :: z        ! altitude [km]
16|   real :: h                    ! hauteur d'echelle [km]
17|
18|   h = (k*T*Na)/(m*g)/1.0e3
19|   pression = Po*exp(-z/h)
20|
21| end function pression
22|
23| end module module_atmosphere
```

Et on accède au module de la façon suivante :

```
1| program acces
2|
3| use module_atmosphere
4|
5| implicit none
6|
7| real :: p, &                  ! pression [Pa]
8|       z                      ! altitude [km]
```

```

9
10 z = 10
11 p = pression(z)
12
13 print*, p, g, h
14
15 end

```

Les variables T , m , g et Po ne sont pas accessibles par le programme *accés* 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_constants* 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*).

```
1 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.

```

1 module module_constants
2
3 implicit none
4
5 ! Constantes de la physique en unite S.I.
6
7 real, parameter :: c = 2.99792458e8 , &           ! vitesse de la lumiere [ms-1]
8                  G = 6.6720e-11, &               ! constante de la gravitation [Nm2kg
9                  h = 6.626176e-34, &               ! constante de planck [JHz-1]
10                 k = 1.380662e-23, &               ! constante de Boltzmann [JK-1]
11                 Na = 6.022045e23, &               ! constante d'Avogadro [mol-1]
12                 Rg = 8.3145 &                     ! constante des gaz parfaits [Jmol-1K
13                 -1]
14 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.

```

1 program nom_program
2
3 implicit none
4 [...]           ! bloc de declaration
5
6 [...]           ! bloc d'instructions executables
7
8 call nom(liste_arguments)           ! appel du sous-programme
9
10 [...]           ! bloc d'instructions executables
11
12 end
13
14 subroutine nom(liste_arguments)
15
16 implicit none
17 [...]           ! bloc de declaration (arguments et variables locales)
18
19 [...]           ! bloc d'instructions executables
20
21 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**.

```

1 program nom_program
2
3 implicit none
4 [...]           ! bloc de declaration
5
6 [...]           ! bloc d'instructions executables
7 call nom(liste_arguments)           ! appel du sous-programme
8 [...]           ! bloc d'instructions executables
9
10 contains
11 subroutine nom1(liste_arguments)
12 [...]           ! bloc de declaration
13 [...]           ! bloc d'instructions executables
14 end subroutine nom1
15
16 subroutine nom2(liste_arguments)
17 [...]           ! bloc de declaration
18 [...]           ! bloc d'instructions executables
19 end subroutine nom2
20
21 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.

```

1 function nom(liste_arguments)
2
3 implicit none
4 real :: nom
5 [...]           ! bloc de declaration
6
7 [...]           ! bloc d'instructions executables
8 nom = ...       ! expression arithmetique
9
10 end function nom

```

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

```
1 nom = 3e43
```

où `nom` est le nom de la fonction.

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

```

1 program nom_program
2
3 implicit none
4 real :: resultat
5 [...]           ! bloc de declaration
6
7 [...]           ! bloc d'instructions executables
8 resultat = 1 + 3*nom(liste_arguments) ! exemple d'appel de la fonction nom
9 [...]           ! bloc d'instructions executables
10
11 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 :

```
1 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 :

```

1 subroutine transfert(a,b)
2
3 implicit none
4 real, intent(inout) :: a,b
5 real :: c
6
7 c=a
8 a=b
9 b=c
10 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 inclu 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.

```

1 program integration
2
3 ! Calcul numerique d'integrales a l'aide des methodes
4 ! composites des trapezes et de simpson
5
6 use mod_integrale
7
8 implicit none
9
10 real :: res1, res2, Pi
11
12 interface                                ! interface obligatoire
13   function f1 (x)
14     real, intent(in) :: x
15     real :: f1
16   end function
17   function f2 (x)
18     real, intent(in) :: x
19     real :: f2
20   end function
21 end interface
22
23 Pi = 4.0*atan(1.0)
24
25 ! Appels de la subroutine sans utiliser les noms cles
26 ! l'ordre des arguments est important
27 print*, ' '
28 call integrale (0.0, 1.0, 100, f1, res1, res2)
29 print*, 'Resultat de la premiere integrale : '
30 print*, 'Methode des trapezes : ',res1
31 print*, 'Methode de simpson : ',res2
32
33 ! Appels de la subroutine en utilisant les noms cles
34 ! l'ordre des arguments n'est pas important
35 print*, ' '
36 call integrale (fonction=f2, trapeze=res1, simpson=res2, &
37               deb=-Pi/3.0, fin=Pi/3.0, nb_int=200)
38 print*, 'Resultat de la seconde integrale : '
39 print*, 'Methode des trapezes : ',res1
40 print*, 'Methode de simpson : ',res2
41
42 end program integration
43
44 function f1 (x)                                ! premiere fonction a integrer
45   real, intent(in) :: x
46   real :: f1
47   f1 = x*x+1
48 end function
49
50 function f2 (x)                                ! seconde fonction a integrer
51   real, intent(in) :: x
52   real :: f2
53   f2 = sin(x)**2.
54 end function

```

avec le module

```

1 module mod_integrale
2
3 implicit none
4
5 contains
6
7 subroutine integrale (deb, fin, nb_int, fonction, trapeze, simpson)
8
9   implicit none
10  real, intent(in) :: deb, fin                ! bornes d'integration
11  real, intent(out) :: trapeze, simpson        ! methodes d'integration
12  integer, intent(in) :: nb_int                ! nb d'intervalles
13  real :: fonction
14
15  real :: pas                                ! pas d'integration
16  integer :: i
17

```



```

18  pas = (fin - deb)/nb_int
19
20  ! methode des trapezes
21  trapeze = pas * ( 0.5*(fonction(deb)+fonction(fin)) + &
22                sum( (/ (fonction(deb+i*pas), i = 1,nb_int-1) /) ) )
23
24
25  ! methode de simpson
26  simpson = pas/3.0 * ( fonction(deb) + fonction(fin) + &
27                    2.0*sum( (/ (fonction(deb+i*pas), i = 2, nb_int-2, 2) /) ) + &
28                    4.0*sum( (/ (fonction(deb+i*pas), i = 1, nb_int-1, 2) /) ) )
29
30  end subroutine integrale
31
32  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.

8.1 Définir un type dérivé

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

```
1 | 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

```

1 | type (planete) :: terre
2
3 | 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.

```

1 | type animal
2 |   character (len=20) :: race
3 |   real :: age
4 |   character (len=15) :: couleur
5 |   logical, dimension(8) :: vaccination
6 | end type animal

```

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

8.2 Initialisation lors de la déclaration

On peut souhaiter définir une variable directement lors de la déclaration, notamment pour en faire une variable globale d'un module, donner une valeur par défaut, ou en faire un paramètre.

Il faut alors définir la variable de la façon suivante :

```

type COULEUR
  character(len=16) :: nom
  real, dimension(3) :: compos
end type COULEUR

```

```
type(COULEUR), parameter :: coul = couleur('rouge', (/ 1.,0.,0. /))
```

9 Astuces et petits bouts de code

9.1 Allouer une variable dynamique dans une subroutine

Voici un bout de code, qui ne s'intéresse pas à la rapidité d'exécution (il y a peut-être mieux) mais qui montre ce qu'il est possible de faire avec des variables dynamiques. Ici, on définit des tableaux dont on ne connaît pas la taille, on lit un fichier pour déterminer le nombre d'élément qu'il nous faut, et on définit les éléments du tableau. Tout ça est fait dans une subroutine qui passe ensuite les tableaux au programme principal :

```

1 program resultant_torque
2   use types_numeriques
3
4   implicit none
5
6   real(double_precision), dimension(:), allocatable :: semi_major_axis
7
8   call read_element_out(semi_major_axis=semi_major_axis)
9   write(*,*) semi_major_axis
10
11  contains
12
13  subroutine read_element_out(semi_major_axis)
14    ! subroutine that get properties of the planets from the file element.out
15
16    implicit none
17
18    character(len=80) :: line
19    character(len=8)  :: name
20    integer :: error ! to store the state of a read instruction
21    real(double_precision) :: value1
22    real(double_precision), dimension(:), allocatable, intent(out) :: semi_major_axis
23    integer :: nb_planets
24
25    logical :: isDefined
26
27    ! For loops
28    integer :: i
29    !-----
30
31    inquire(file='element.out', exist=isDefined)
32    if (isDefined) then
33
34        open(10, file='element.out', status='old')
35
36        ! We read the header. There the time in there, but we do not need it for the
37        ! moment
38        do i=1,5
39            read(10, '(a80)', iostat=error), line
40        end do
41
42        nb_planets = 0
43        do
44            read(10, *, iostat=error), line
45            if (error.eq.0) then
46                nb_planets = nb_planets + 1
47            else
48                exit
49            end if
50        end do
51
52        ! Once we have the number of planets, we rewind the file and read again
53        rewind(unit=10, iostat=error)
54
55        ! We allocate the array now we know its desired size
56        allocate(semi_major_axis(nb_planets))
57
58        ! We skip the header
59        do i=1,5
60            read(10, '(a80)', iostat=error), line
61        end do

```

```

62     i=1
63     do
64         read(10, *, iostat=error), name, value1
65         if (error /= 0) exit
66         semi_major_axis(i) = value1
67         i = i + 1
68
69     end do
70     else
71         write(*,*) 'Error: the file "element.out" does not exist'
72     end if
73
74     end subroutine read_element_out
75
76 end program resultant_torque

```

La **subroutine** `read_element_out` va lire dans un premier temps le fichier (sans tenir compte du header de 5 lignes) afin de déterminer le nombre de lignes utiles, et donc le nombre d'élément qu'aura notre tableau. Puis, il alloue la place nécessaire au tableau et enfin il stocke les éléments, un par un.

À noter que `read_element_out` affiche une erreur si le fichier nécessaire n'existe pas (dans le cas présent, le fichier `element.out`).

9.2 Lire un fichier de longueur inconnue

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

```

1  implicit none
2
3  integer, parameter :: NMAX = 200 ! max number of messages
4
5  integer :: error
6  integer, dimension(NMAX) :: length
7  character(len=80), dimension(NMAX) :: message
8
9  open(14, file='message.in', status='old')
10 do
11     read (14, '(i3,1x,i2,1x,a80)', iostat=error) j, length(i), message(i)
12     if (error /= 0) exit
13 end do
14 close(14)

```

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

9.3 Lire un fichier de paramètres

Ce n'est pas *la* bonne façon de faire, c'est juste ce que je fais moi.

Je défini un caractère qui autorise des commentaires (dans mon exemple, «! »). Les lignes blanches ou avec espaces sont ignorées quant à elle, vu que l'élément déclencheur est la présence du séparateur entre nom de paramètre et valeur.

Je défini aussi un séparateur pour les paramètres (dans mon exemple « = »). Chaque paramètre doit posséder un nom, sans espaces. L'ordre des paramètres n'a pas d'importance.

Remarque : Dans le reste du programme je défini des valeurs par défaut qui seront alors écrasées lors de la lecture du fichier si le paramètre y est défini. Je défini en effet les variables en tant que variables globales du module, vu qu'elles sont constantes dans le programme (même si le fait que je ne les connaisse pas à priori m'empêche de les définir en tant que `parameter`).

```

1  subroutine read_disk_properties()
2  ! subroutine that read the 'disk.in' file to retrieve disk properties.
3  ! Default value exist, if a parameter is not defined
4
5  implicit none
6
7  character(len=80) :: line
8

```

```

9      ! character that will indicate that the reste of the line is a comment
10     character(len=1) :: comment_character = '!'
11
12     ! the index of the comment character on the line.
13     ! If zero, there is none on the current string
14     integer :: comment_position
15
16     integer :: error ! to store the state of a read instruction
17
18     logical :: isParameter, isDefined
19     character(len=80) :: identificador, value
20     !-----
21
22     open(10, file='disk.in', status='old')
23
24     do
25         read(10, '(a80)', iostat=error), line
26         if (error /= 0) exit
27
28         ! We get only what is on the left of an eventual comment parameter
29         comment_position = index(line, comment_character)
30
31         ! If there are comments on the current line, we get rid of them
32         if (comment_position.ne.0) then
33             line = line(1:comment_position - 1)
34         end if
35
36         call get_parameter_value(line, isParameter, identificador, value)
37
38         if (isParameter) then
39             select case(identificador)
40                 case('b/h')
41                     read(value, *) b_over_h
42
43                 case('adiabatic_index')
44                     read(value, *) adiabatic_index
45
46                 case('temperature')
47                     read(value, *) temperature_0, temperature_index
48
49                 case default
50                     write(*,*) 'Warning: An unknown parameter has been found'
51                     write(*,*) "identificador='", trim(identificador), &
52                               "' ; value(s)='", trim(value), ""
53             end select
54         end if
55     end do
56
57     close(10)
58
59 end subroutine read_disk_properties

```

Dans cet exemple, je montre comment définir un paramètre ne contenant qu'une seule valeur, ou un paramètre contenant plusieurs valeurs (ici deux, mais il peut y en avoir plus).



Notez que je ne lis que les 80 premiers caractères d'une ligne. Il ne peut donc pas y avoir de valeur définie au delà du 80^e caractère. Par contre, la longueur des commentaires est arbitraire, y compris sur les lignes qui contiennent des paramètres en début de ligne.

Je défini ensuite la subroutine qui me permet de récupérer l'identificateur et la (ou les) valeur(s) associée(s) :

```

1 subroutine get_parameter_value(line, isParameter, id, value)
2 ! subroutine that try to split the line in two part,
3 ! given a separator value (set in parameter of the subroutine)
4 !
5 ! The routine return 3 values :
6 !
7 ! Return
8 ! isParameter : is a boolean to say whether or not there is a parameter
9 ! on this line. i.e if there is an occurrence of the separator
10 ! in the input line.

```

```

11 ! id : a string that contain the name of the parameter
12 ! value : a string that contains the value(s) associated with
13 ! the parameter name
14
15 implicit none
16
17 ! Input
18 character(len=80), intent(in) :: line
19
20 ! Output
21 logical, intent(out) :: isParameter
22 character(len=80), intent(out) :: id, value
23
24 ! Local
25 ! the separator of a parameter line
26 character(len=1), parameter :: sep = '='
27
28 integer :: sep_position ! an integer to get the position of the separator
29
30 ! -----
31
32 sep_position = index(line, sep)
33
34 if (sep_position.ne.0) then
35     isParameter = .true.
36     id = line(1:sep_position-1)
37     value = line(sep_position+1:)
38 else
39     isParameter = .false.
40 end if
41
42 end subroutine get_parameter_value

```

Le fichier de paramètre est quelque chose du genre :

```

! -----
! Parameter file for various properties of the disk.
! -----

adiabatic_index = 1.4
temperature = 510 1
b/h = 0.4

```

9.4 Initialisation implicite d'un tableau

Dans la section [§ 5.2.3, p14], une astuce est montrée pour une boucle implicite. Cette dernière est recopiée ici :

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

9.5 Remplissage d'un tableau dynamique de taille variable

Sous ce titre flou se cache une idée simple. On veut stocker des informations sur un nombre inconnu d'événements. Par exemple, je veux stocker la valeur des pas de temps jusqu'à un temps final. Les pas de temps étant non constants, je ne connais pas *a priori* leur nombre. Ainsi, je peux faire quelque chose comme ça :

```

1 real(double_precision), dimension(:), allocatable :: time, time_temp
2 real(double_precision) :: time_size ! the size of the array 'time'.
3 integer :: error ! to retrieve error, especially during allocations
4
5 time_size = 512 ! the size of the array.
6 allocate(time(time_size), stat=error)
7
8 [...]
9
10 ! If the limit of the array is reach,
11 ! we copy the values in a temporary array,
12 ! allocate with a double size, and
13 ! paste the old values in the new bigger array

```

```

14 if (k.eq.time_size) then
15   allocate(time_temp(time_size), stat=error)
16   time_temp(1:time_size) = time(1:time_size)
17   deallocate(time, stat=error)
18   time_size = time_size * 2
19   allocate(time(time_size), stat=error)
20   time(1:time_size/2) = time_temp(1:time_size/2)
21   deallocate(time_temp, stat=error)
22 end if

```

où `time` est mon tableau principal, contenant la valeur des pas de temps et `time_temp` le tableau tampon me permettant de stocker les valeurs de `time` pendant que je l'agrandis.

Je définis `time_size` (la taille du tableau `time`) comme étant égal à 512 au départ. Doubler la taille du tableau à chaque fois permet de limiter les réallocations qui sont coûteuses en temps.

10 Les fonctions intrinsèques en Fortran 90

Voici une liste de quelques fonctions intrinsèques du *Fortran 90*. Toutes les fonctionnalités de chaque fonction ne sont pas données. Pour plus de renseignements sur ces fonctions et les autres fonctions, consultez les ouvrages de référence (voir bibliographie).

10.1 Les fonctions numériques

10.1.1 Les fonctions numériques élémentaires

ABS *abs(a)* Fournit la valeur absolue de a ($|a|$), de type entier, réel ou complexe. Le résultat est du type de a . Quand a est complexe ($a = (x, y)$), le résultat est de type *real* et égal à $\sqrt{x^2 + y^2}$.

Exemple :

$$\text{abs}(-7.4) = 7.4 \quad (10.1)$$

$$\text{abs}((6.0, 8.0)) = 10.0 \quad (10.2)$$

AIMAG *aimag(z)* Fournit la partie imaginaire de z , de type complexe. Le résultat est de type *real*. Si $z = (x, y)$, le résultat est égal à y .

Exemple :

$$\text{aimag}((4.0, 5.0)) = 5.0 \quad (10.3)$$

AINTE *aint(a)* Fournit, dans le type *real*, la troncature de a (partie entière $E(a)$ pour $a > 0$ et $-E(-a)$ pour $a < 0$).

Exemple :

$$\text{aint}(3.678) = 3.0 \quad (10.4)$$

$$\text{aint}(-1.375) = -1.0 \quad (10.5)$$

ANINT *anint(a)* Fournit, dans le type *real*, l'arrondi de a à l'entier le plus proche.

Exemple :

$$\text{anint}(3.456) = 3.0 \quad (10.6)$$

$$\text{anint}(-2.798) = -3.0 \quad (10.7)$$

CEILING *ceiling(a)* Fournit l'entier (type *integer* standard) immédiatement supérieur à la valeur du réel a (voir *floor*).

Exemple :

$$\text{ceiling}(4.8) = 5 \quad (10.8)$$

$$\text{ceiling}(-2.55) = -2 \quad (10.9)$$

CMPLX *cmplx(x,y)* – Si y est absent : fournit le résultat de la conversion de la valeur x (de type numérique quelconque, complexe compris) en un complexe standard,
 – Si y est présent : fournit le résultat de la conversion du complexe (x,y) (x et y doivent être de type entier ou réel) dans le type complexe standard.

Exemple :

$$\text{cmplx}(-3) = (-3.0, 0.0) \quad (10.10)$$

$$\text{cmplx}(4.1, 2.3) = (4.1, 2.3) \quad (10.11)$$

FLOOR *floor(a)* Fournit l'entier (type *integer* standard) immédiatement inférieur à la valeur du réel a (voir *ceiling*).

Exemple :

$$\text{floor}(4.8) = 4 \quad (10.12)$$

$$\text{floor}(-5.6) = -6 \quad (10.13)$$

INT *int(a)* Fournit le résultat de la conversion en entier standard de la valeur de a qui peut être entière (le résultat est alors égal à a), réelle (le résultat est alors égal à la troncature de a , c'est-à-dire *aint(a)*) ou complexe (le résultat est alors égal à la troncature de la partie réelle de a).

Exemple :

$$\text{int}(-4.2) = -4 \quad (10.14)$$

$$\text{int}(7.8) = 7 \quad (10.15)$$

NINT *nint(a)* Fournit l'entier standard le plus proche du réel a .

Exemple :

$$\text{nint}(3.879) = 4 \quad (10.16)$$

$$\text{nint}(-2.789) = -3 \quad (10.17)$$

REAL *real(a)* Fournit le réel correspondant à a qui peut être de type numérique quelconque (s'il est complexe, on obtient la partie réelle).

Exemple :

$$\text{real}(-4) = -4.0 \quad (10.18)$$

Remarque : les fonctions suivantes (*conjg* à *sign*) fournissent un résultat ayant le même type que leur premier argument et lorsque plusieurs arguments sont prévus, ils doivent tous être du même type.

CONJG *conjg(z)* Fournit le complexe conjugué du complexe z .

Exemple :

$$\text{conjg}((2.0, 3.0)) = (2.0, -3.0) \quad (10.19)$$

DIM *dim(x, y)* Fournit le maximum de $x - y$ et de 0, x et y sont entiers ou réels.

Exemple :

$$\text{dim}(6, 2) = 4 \quad (10.20)$$

$$\text{dim}(-4.0, 3.0) = 0.0 \quad (10.21)$$

MAX *max(a1, a2, a3...)* Fournit la valeur maximale des valeurs reçues en arguments (entiers ou réels).

Exemple :

$$\max(2.0, -8.0, 6.0) = 6.0 \quad (10.22)$$

MIN *min(a1, a2, a3...)* Fournit la valeur minimale des valeurs reçues en arguments (entiers ou réels).

Exemple :

$$\min(2.0, -8.0, 6.0) = -8.0 \quad (10.23)$$

MOD *mod(a, p)* Fournit la valeur de $a - \text{int}(a/p) * p$; a et p doivent être de même type (entiers ou réels). Si p est nul, le résultat dépend de la machine (pas défini).

Exemple :

$$\text{mod}(7, 3) = 1 \quad (10.24)$$

$$\text{mod}(9, -6) = 3 \quad (10.25)$$

Cette fonction est souvent utilisée pour tester si un nombre est pair ou impair.

Exemple :

$$\text{mod}(4, 2) = 0 \quad (10.26)$$

$$\text{mod}(5, 2) = 1 \quad (10.27)$$

MODULO *modulo(a, p)* Fournit la valeur de a modulo p , c'est-à-dire $\mathbf{a-floor(a/p)*p}$ quand a et p sont réels et $\mathbf{a-floor(a:p)*p}$ (« : » représentant la division euclidienne quand a et p sont entiers). Si p est nul, le résultat dépend de la machine (pas défini).

Exemple :

$$\text{modulo}(7, 3) = 1 \quad (10.28)$$

$$\text{modulo}(9, -6) = -3 \quad (10.29)$$

$$\text{modulo}(156, 2) = 0 \quad (10.30)$$

SIGN *sign(a, b)* Fournit la valeur absolue de a multiplié par le signe de b (a et b doivent être de même type).

Exemple :

$$\text{sign}(4.0, -6.0) = -4.0 \quad (10.31)$$

$$\text{sign}(-5.0, 2.0) = 5.0 \quad (10.32)$$

10.1.2 Les fonctions mathématiques élémentaires

Remarque : toutes ces fonctions fournissent un résultat ayant le même type que leur premier argument. L'argument x peut être aussi un tableau. Dans ce cas, la fonction s'applique sur tous les éléments du tableau.

ACOS *acos(x)* Fournit la valeur en radians, dans l'intervalle $[0, \pi]$, de arccosinus de x , x étant un réel tel que $x \in [-1, 1]$.

ASIN *asin(x)* Fournit la valeur en radians, dans l'intervalle $[-\frac{\pi}{2}, \frac{\pi}{2}]$, de arcsinus de x , x étant un réel tel que $x \in [-1, 1]$.

ATAN *atan(x)* Fournit la valeur de arctangente de x , x étant réel, exprimé en radians, dans l'intervalle $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

ATAN2 *atan2(y, x)* Fournit la valeur principale de l'argument du nombre complexe (x, y) , exprimée en radians dans l'intervalle $[-\pi, \pi]$, x et y étant du même type réel.

Les valeurs de x et y ne doivent pas être toutes deux nulles.

Exemple :

$$\text{atan2}(2.679676, 1.0) = 1.213623 \quad (10.33)$$

COS *cos(x)* Fournit le cosinus de x pour x réel ou complexe, exprimé en radians.

COSH *cosh(x)* Fournit la valeur du cosinus hyperbolique de x , x étant un réel exprimé en radians.

EXP *exp(x)* Fournit la valeur de l'exponentielle de x pour x réel ou complexe.

LOG *log(x)* Fournit la valeur du logarithme népérien de x , réel positif ou complexe non nul (dans ce dernier cas, le résultat possède une partie imaginaire situé dans l'intervalle $[-\pi; \pi]$).

LOG10 *log10(x)* Fournit la valeur du logarithme à base 10 du réel positif x .

SIN *sin(x)* Fournit le sinus de x pour x réel ou complexe et exprimé en radians.

SINH *sinh(x)* Fournit la valeur du sinus hyperbolique de x , x étant réel et exprimé en radians.

SQRT *sqr(x)* Fournit la valeur de la racine carrée de x , réel positif ou nul ou complexe (dans ce cas, le résultat possède une partie réelle non négative; s'il s'agit de 0, la partie imaginaire du résultat n'est pas négative).

TAN *tan(x)* Fournit la tangente de x réel exprimé en radians.

TANH *tanh(x)* Fournit la tangente hyperbolique de x pour x réel et exprimé en radians.

10.1.3 Les fonctions d'interrogation

Remarque : Toutes ces fonctions peuvent recevoir en argument un scalaire (numérique) ou un tableau d'éléments numériques.

DIGITS *digits(x)* Fournit le nombre (entier standard) de chiffres significatifs du type de x (réel ou entier).

Exemple : si x est un réel codé sur 32 bits,

$$\text{digits}(x) = 24 \quad (10.34)$$

EPSILON *epsilon(x)* Fournit l'"epsilon machine" (le plus petit nombre ϵ tel que $1 + \epsilon$ soit différent de 1) du type correspondant au réel x . Le résultat a la valeur b^{1-p} .

Exemple : si x est un réel codé sur 32 bits,

$$\text{epsilon}(x) = 2^{-23} \quad (10.35)$$

HUGE *huge(x)* Fournit la plus grande valeur représentable dans le type de x . Le résultat est du même type que x qui peut être entier ou réel. Si x est réel, le résultat a la valeur $(1 - b^{-p}) * b^{e_{max}}$.

Exemple : si x est un réel codé sur 32 bits,

$$\text{huge}(x) \rightarrow (1 - 2^{-24}) * 2^{128} \quad (10.36)$$

MAXEXPONENT *maxexponent(x)* Fournit la plus grande valeur (entier standard) possible pour un exposant dans le type (réel) de x .

MINEXPONENT *minexponent(x)* Fournit la plus petite valeur (entier standard) possible pour un exposant dans le type (réel) de x .

PRECISION *precision(x)* Fournit le nombre (entier standard) minimal de chiffres significatifs dans le type (réel ou complexe) de x .

RANGE *range(x)* Fournit la valeur maximale e (entier standard) d'un exposant (en puissance de 10) dans le type de x (entier, réel ou complexe) telle que les nombres 10^e et 10^{-e} soient représentables dans le type en question.

Exemple : si x est un réel codé sur 32 bits,

$$\text{range}(x) = 37 \quad (10.37)$$

TINY *tiny(x)* Fournit la plus petite valeur représentable dans le type de x . Le résultat est du même type (variante comprise) que x qui peut être réel ou double précision. Le résultat a la valeur $b^{e_{\min}-1}$.

Exemple : si x est un réel codé sur 32 bits,

$$\text{tiny}(x) = 2^{-126} \quad (10.38)$$

(environ 10^{-38}).

10.2 les fonctions relatives aux chaînes de caractères

10.2.1 Les fonctions élémentaires

CHAR *char(i)* Fournit une chaîne de caractères de longueur 1 correspondant au caractère de code ASCII i (entier).

Exemple :

$$\text{char}(71) = 'G' \quad (10.39)$$

$$\text{char}(63) = '?' \quad (10.40)$$

ICHAR *ichar(c)* Fournit l'entier (type *integer* standard) correspondant au code ASCII du caractère c (chaîne de longueur 1).

Exemple :

$$\text{ichar}('Y') = 89 \quad (10.41)$$

$$\text{ichar}('%') = 37 \quad (10.42)$$

10.2.2 Les fonctions de comparaison de chaînes

LGE *lge(string_a, string_b)* Fournit la valeur *vrai* si la chaîne *string_a* apparaît après *string_b* ou lui est égale (dans l'ordre alphabétique donné par la table de caractères).

Exemple :

$$\text{lge}('TWO', 'THREE') = \text{true} \quad (10.43)$$

LGT *lgt(string_a, string_b)* Fournit la valeur *vrai* si la chaîne *string_a* apparaît après *string_b* (dans l'ordre alphabétique donné par la table de caractères).

LLE *lle(string_a, string_b)* Fournit la valeur *vrai* si la chaîne *string_a* apparaît avant *string_b* ou lui est égale (dans l'ordre alphabétique donné par la table de caractères).

Exemple :

$$\text{lge}('TWO', 'THREE') = \text{false} \quad (10.44)$$

LLT *llt(string_a, string_b)* Fournit la valeur *vrai* si la chaîne *string_a* apparaît avant *string_b* (dans l'ordre alphabétique donné par la table de caractères).

10.2.3 Les fonctions de manipulation de chaînes

ADJUSTL *adjustl(string)* Fournit en résultat la chaîne *string* « cadrée à gauche », c'est-à-dire débarrassée de tous ses blancs de début (et donc complétée à droite par autant de blancs supplémentaires).

Exemple :

```
adjustl('####summer') = ' summer####'
```

 (10.45)

Le caractère # représente ici un blanc.

ADJUSTR *adjustr(string)* Fournit en résultat la chaîne *string* « cadrée à droite », c'est-à-dire débarrassée de tous ses blancs de fin (et donc complétée à gauche par autant de blancs supplémentaires).

Exemple :

```
adjustl('summer####') = '####summer'
```

 (10.46)

Le caractère # représente ici un blanc.

INDEX *index(string, substring, back)* Fournit un entier (de type *integer* standard) correspondant au premier caractère de la chaîne *string* où apparaît la sous-chaîne *substring* (ou la valeur 0 si cette sous-chaîne n'apparaît pas). Si *back* (de type logique) n'est pas précisé ou s'il a la valeur *faux*, l'exploration se fait depuis le début de la chaîne; si *back* a la valeur *vrai*, cette recherche se fait depuis la fin de la chaîne (on a donc, en fait, la dernière « occurrence » de la sous-chaîne).

Exemple :

```
index('caractere','a',back = .true.) = 4
```

 (10.47)

LEN_TRIM *len_trim(string)* Fournit la longueur (type *integer* standard) de la chaîne *string*, débarrassée de ses blancs de fin.

Exemple :

```
len_trim('###C##D###') = 7
```

 (10.48)

Le caractère # représente ici un blanc.

SCAN *scan(string, set, back)* Fournit un entier (de type *integer* standard) correspondant au premier caractère de la chaîne *string* où apparaît l'un des caractères de la chaîne *set* (ou la valeur 0 si cette chaîne n'apparaît pas). Si *back* (de type logique) n'est pas précisé ou s'il a la valeur *faux*, l'exploration se fait depuis le début de la chaîne; si *back* a la valeur *vrai*, cette recherche se fait depuis la fin de la chaîne (on a donc, en fait, la dernière « occurrence » de l'un des caractères mentionnés).

Exemple :

```
scan('astring','st') = 2
```

 (10.49)

```
scan('astring','st',back = .true.) = 3
```

 (10.50)

VERIFY *verify(string, set, back)* Fournit un entier (de type *integer* standard) valant 0 si tous les caractères de la chaîne *string* figurent dans la chaîne *set* ou la position du premier caractère de *string* qui ne figure pas dans *set* dans le cas contraire. Si *back* (de type logique) n'est pas précisé ou s'il a la valeur *faux*, l'exploration se fait depuis le début de la chaîne; si *back* a la valeur *vrai*, cette recherche se fait depuis la fin de la chaîne.

Exemple :

```
verify('cdddc','c') = 2
```

 (10.51)

```
verify('cdddc','c',back = .true.) = 4
```

 (10.52)

LEN *len(string)* Entier (*integer* standard) correspondant à la longueur de la chaîne *string*. Si *string* est un tableau de chaînes, on obtient la longueur d'un élément d'un tel tableau.

Exemple : si *c* a été déclaré comme suit *character(15) :: c* alors

```
len(c) = 15
```

 (10.53)

REPEAT *repeat(string, n)* Fournit en résultat une chaîne obtenue en concaténant n fois la chaîne *string*.

Exemple :

$$\text{repeat}('s', 3) = 'sss' \quad (10.54)$$

TRIM *trim(string)* Fournit une chaîne obtenue en débarrassant la chaîne *string* de tous ses blancs de fin.

Exemple :

$$\text{trim}('###C##D###') = '###C##D' \quad (10.55)$$

Le caractère # représente ici un blanc.

10.3 Les fonctions relatives aux tableaux

DOT_PRODUCT *dot_product(vector_a, vector_b)* Fournit le produit scalaire de deux vecteurs (peut également porter sur des tableaux de type logique). Les arguments *vector_a* et *vector_b* doivent être des tableaux de rang 1 et de même taille, ayant soit tous les deux un type numérique (mais pas obligatoirement le même), soit tous les deux un type logique. Si *vector_a* est de type entier ou réel, on obtient la valeur de l'expression : `sum(vector_a*vector_b)`, avec le type correspondant à cette expression. Si *vector_a* est de type complexe, on obtient la valeur de l'expression : `sum(conj(vector_a)*vector_b)`, avec le type correspondant à cette expression. Si *vector_a* et *vector_b* sont tous deux de type logique, on obtient la valeur de l'expression : `any(vecteur_a.and.vecteur_b)` avec le type correspondant à cette expression.

Exemple :

$$\text{dot_product}((/1, 2, 3/), (/3, 4, 5/)) = 26 \quad (10.56)$$

MATMUL *matmul(matrix_a, matrix_b)* Produit scalaire de deux matrices, ou produit d'un vecteur (ligne) par une matrice ou produit d'une matrice par un vecteur (colonne). Les dimensions des tableaux sont soumises aux contraintes mathématiques habituelles. De plus, comme *dot_product*, la fonction *matmul* peut porter sur des tableaux de type logique. Les arguments sont donc des tableaux de rang 1 ou 2 qui doivent être soit tous deux de type numérique (mais pas nécessairement le même), soit tous deux de type logique.

Exemple : si $A = \begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$ et $X = (1, 2)$, alors

$$\text{matmul}(X, A) = (8, 11, 14) \quad (10.57)$$

Remarque : les fonctions suivantes s'appliquent à des tableaux de rang quelconque et fournissent un résultat scalaire du même type que les éléments du tableau. Elles peuvent toutes comporter l'argument optionnel *dim* (*dim=1* se rapporte à la première dimension).

ALL *all(mask)* Fournit la valeur *vrai* si tous les éléments du tableau logique *mask* ont la valeur *vrai*.

Exemple :

$$\text{all}((/.true., .false., .true./)) = \text{false} \quad (10.58)$$

ANY *any(mask)* Fournit la valeur *vrai* si l'un au moins des éléments du tableau logique *mask* a la valeur *vrai* et la valeur *faux* dans le cas contraire.

Exemple :

$$\text{any}((/.true., .false., .true./)) = \text{true} \quad (10.59)$$

COUNT *count(mask)* Fournit le nombre (entier standard) d'éléments du tableau logique *mask* ayant la valeur *vrai*.

Exemple :

$$\text{count}(/.true.,.false.,.true./) = 2 \quad (10.60)$$

MAXVAL *maxval(array)* Fournit la plus grande valeur du tableau *array*. Le résultat est du même type que les éléments de *array* qui peuvent être de type entier ou réel. Si le tableau *array* est de taille nulle, le résultat est égal à la plus petite valeur représentable dans le type concerné.

Exemple : si $A = \begin{pmatrix} 2 & 6 & 4 \\ 5 & 3 & 7 \end{pmatrix}$,

$$\text{maxval}(A, \text{dim} = 1) = (5, 6, 7) \quad (10.61)$$

(5 est la valeur max. dans la colonne 1, 6 est la valeur max. dans la colonne 2, etc.) et

$$\text{maxval}(A, \text{dim} = 2) = (6, 7) \quad (10.62)$$

(6 est la valeur max. dans la ligne 1, 7 est la valeur max. dans la ligne 2).

MINVAL *minval(array)* Fournit la plus petite valeur du tableau *array*. Le résultat est du même type que les éléments de *array* qui peuvent être de type entier ou réel. Si le tableau *array* est de taille nulle, le résultat est égal à la plus grande valeur représentable dans le type concerné.

Exemple : si $A = \begin{pmatrix} 2 & 6 & 4 \\ 5 & 3 & 7 \end{pmatrix}$,

$$\text{minval}(A, \text{dim} = 1) = (2, 3, 4) \quad (10.63)$$

(2 est la valeur min. dans la colonne 1, 3 est la valeur min. dans la colonne 2, etc.) et

$$\text{minval}(A, \text{dim} = 2) = (2, 3) \quad (10.64)$$

(2 est la valeur min. dans la ligne 1, 3 est la valeur min. dans la ligne 2).

PRODUCT *product(array)* Fournit le produit des valeurs des éléments du tableau *array*. Le résultat est du même type que les éléments de *array* qui peuvent être de type entier, réel ou complexe. Si le tableau *array* est de taille nulle, le résultat vaut 1.

Exemple : si $A = \begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix}$,

$$\text{product}(A, \text{dim} = 1) = (10, 18, 28) \quad (10.65)$$

$$\text{product}(A, \text{dim} = 2) = (24, 210) \quad (10.66)$$

SUM *sum(array)* Fournit la somme des valeurs des éléments du tableau *array*. Le résultat est du même type que les éléments de *array* qui peuvent être de type entier, réel ou complexe. Si le tableau *array* est de taille nulle, le résultat vaut 0.

Exemple : si $A = \begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix}$,

$$\text{sum}(A, \text{dim} = 1) = (7, 9, 11) \quad (10.67)$$

$$\text{sum}(A, \text{dim} = 2) = (9, 18) \quad (10.68)$$

ALLOCATED *allocated(array)* Fournit la valeur *vrai* si le tableau *array* (déclaré avec l'attribut *allocate*) est alloué et la valeur *faux* dans le cas contraire.

LBOUND *lbound(array,dim)* Si *dim* (entier) est présent, fournit la borne inférieure (entier standard) du tableau *array*, suivant sa dimension *dim*. Si *dim* est absent, Fournit un tableau d'entiers ayant

le rang de *array*, contenant les bornes inférieures de *array*, suivant toutes ses dimensions. Notez qu'il n'est pas possible d'appeler *lbound* au sein d'une procédure en lui fournissant un deuxième argument effectif qui soit lui-même un argument muet optionnel.

Exemple : si *A* a été déclaré comme suit *real* : : *A*(1 : 3, 5 : 8) alors

$$\text{lbound}(A) = (1, 5) \quad (10.69)$$

Voir aussi *ubound*.

SHAPE *shape(source)* Fournit un tableau d'entiers (standards) de rang 1 correspondant au profil de *source*. Si *source* est un scalaire, on obtient un tableau de taille 0.

Exemple : si *A* a été déclaré comme suit *real* : : *A*(1 : 3, 5 : 8) alors

$$\text{shape}(A) = (3, 4) \quad (10.70)$$

SIZE *size(array, dim)* Si *dim* est présent, fournit l'étendue (entier standard) de *array*, suivant sa dimension *dim*. Si *dim* est absent, fournit la taille de *array*.

Exemple : si *A* a été déclaré comme suit *real* : : *A*(1 : 3, 5 : 8) alors

$$\text{size}(A) = 12 \quad (10.71)$$

$$\text{size}(A, \text{dim} = 2) = 4 \quad (10.72)$$

UBOUND *ubound(array, dim)* Si *dim* (entier) est présent, fournit la borne supérieure (entier) du tableau *array*, suivant sa dimension *dim*. Si *dim* est absent, fournit un tableau d'entiers ayant le rang de *array*, contenant les bornes supérieures de *array*, suivant toutes ses dimensions. Notez qu'il n'est pas possible d'appeler *ubound* au sein d'une procédure en lui fournissant un deuxième argument effectif qui soit lui-même un argument muet optionnel.

Exemple : si *A* a été déclaré comme suit *real* : : *A*(1 : 3, 5 : 8) alors

$$\text{ubound}(A) = (3, 8) \quad (10.73)$$

Voir aussi *lbound*.

TRANSPOSE *transpose(matrix)* L'argument *matrix* doit être un tableau de rang 2, de type quelconque. Il fournit un tableau de même rang et dont les étendues sont celles de *matrix* inversées et dont l'élément d'indices *i, j* est *matrix(j, i)* (comme dans la transposée d'une matrice).

10.4 Procédures diverses

DATE_AND_TIME *call date_and_time(date, time, zone, values)* Fournit en sortie, dans les différents arguments dont le type est précisé ci-dessous les valeurs suivantes (lorsqu'elles ne sont pas disponibles, on obtient un espace pour les arguments de type *character* et la valeur *-huge(0)*, c'est-à-dire le plus petit entier négatif, pour les arguments de type numérique).

- *date (character)* : date sous la forme *aaaammjj* (4 caractères pour l'année, 2 pour le numéro de mois, 2 pour le numéro de jour),
- *time (character)* : heure sous la forme *hhmmss.sss* (2 caractères pour l'heure, 2 pour les minutes, 2 pour les secondes, un point et 2 caractères pour les millièmes de secondes),
- *zone (character)* : écart entre l'heure locale et le temps universel sous la forme *shhmm* (1 caractère pour le signe + ou -, 2 caractères pour les heures et 2 caractères pour les minutes).
- *values (tableau de rang 1 de 8 entiers)* : il fournit, sous forme numérique les différentes informations précédentes; ses éléments correspondent dans l'ordre à : année, numéro de mois, numéro de jour, différence en minutes avec le temps universel, heure, minutes, secondes et millièmes de seconde.

SYSTEM_CLOCK *call system_clock(count, count_rate, count_max)* Fournit dans les trois éléments de sortie, de type *integer*, les informations suivantes :

- *count* : valeur de l'horloge interne (*-huge(0)* si elle n'est pas accessible),

- *count_rate* : fréquence de l'horloge interne (0 si elle n'est pas accessible),
- *count_max* : valeur maximale de l'horloge interne (nombre de coups qu'elle peut compter avant de repasser à zéro, 0 si elle n'est pas accessible).

RANDOM_NUMBER *call random_number(x)* Fournit, dans x (réel) un nombre pseudo-aléatoire appartenant à l'intervalle $[0, 1[$. L'argument x peut être un tableau de réels; dans ce cas, on obtient une série de nombres aléatoires.

RANDOM_SEED *call random_seed(size, put, get)* Permet d'initialiser ou d'interroger le générateur de nombres aléatoires utilisé par la routine **random_number**. Un seul des trois arguments doit être spécifié.

- *size (integer de genre out)* : taille du tableau d'entiers utilisé comme « graines » pour la génération des nombres aléatoires,
- *put (integer de genre in)* : tableau d'entiers de rang 1 (de taille égale à la valeur fournie dans *size*) qui correspond aux valeurs qui seront utilisées comme graines pour la génération des nombres aléatoires.
- *get (integer de genre out)* : tableau d'entiers de rang 1 (de taille égale à la valeur fournie dans *size*) qui correspond aux valeurs utilisées pour la génération des nombres aléatoires. Si aucun argument n'est précisé, le générateur de nombres aléatoires est initialisé d'une manière dépendant de la machine.

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.

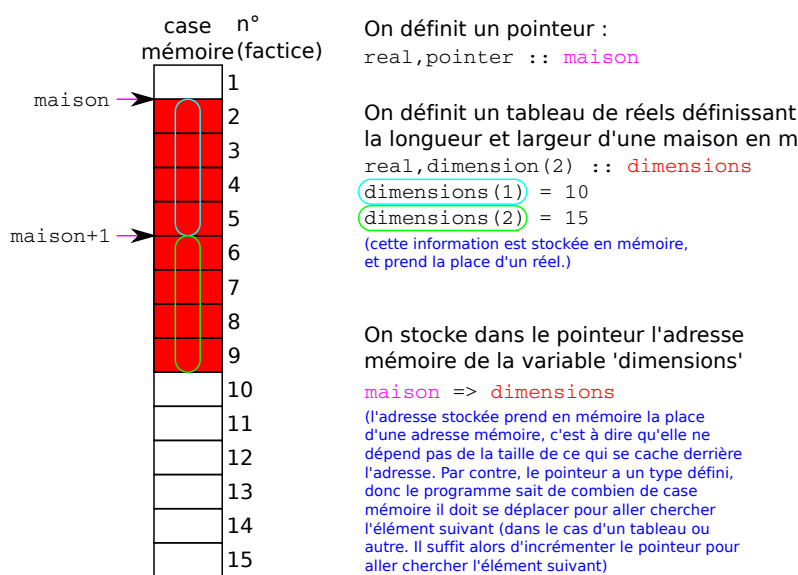


FIGURE 5 – Représentation schématique du fonctionnement d'un pointeur avec représentation de la mémoire. Les numéros pour les cases mémoire ne représentent pas la réalité, c'est juste pour montrer comment on fait référence à une case mémoire. L'idée est de montrer qu'en manipulant des adresses au lieu 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

```
1 | 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)

```
1 | 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 :

```
1 | 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 :

```
1 | subroutine nom(a, b, c)
2 |
3 | implicit none
4 | integer, intent(in) :: a           ! argument d'entree
5 | real, intent(out) :: b             ! argument de sortie
6 | logical, intent(inout) :: c       ! argument d'entree et de sortie
7 |
8 | [...]                             ! bloc d'instructions
9 |
10 | end subroutine nom
```


11.3 Optimisation

11.3.1 Comparaison f77/f90

11.3.2 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.

Profiler 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*.

flat profile Le flat profile ressemble à ça :

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self	calls	self	total	name
time	seconds	seconds		ms/call	ms/call	
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.

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_mio_spl [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]

11.3.3 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

```

1 A=PI
2
3 call CPU_TIME(start_time)
4 do I=1,MAXLOOPS
5     B=A**4
6 end do
7 call CPU_TIME(stop_time)
8 write(*,*) "A**4 Duration="      ",stop_time - start_time

```

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

```

1 A=PI
2 N=4

```

```

3
4 call CPU_TIME(start_time)
5 do I=1,MAXLOOPS
6   B=A**N
7 end do
8 call CPU_TIME(stop_time)
9 write(*,*) "A**N (N=4) Duration= ",stop_time - start_time

```

3. en multipliant X par lui même Y fois.

```

1 A=PI
2
3 call CPU_TIME(start_time)
4 do I=1,MAXLOOPS
5   B=A*A*A*A
6 end do
7 call CPU_TIME(stop_time)
8 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

```

$$A^n \Rightarrow \underbrace{A * \dots * A}_{n \text{ fois}} \quad (11.1)$$

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.

11.3.4 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 :

```

1 call cpu_time(start_time)
2 do i=1,maxloops
3   b=i*sin(pi/4.0)
4 end do
5 call cpu_time(stop_time)
6 write(*,*) "Repeated pi/4 & sine, Duration= ",stop_time - start_time

```

2. en utilisant la table d'arguments :

```

1 call cpu_time(start_time)
2 do i=1,maxloops
3   b=i*sin(pi_table(4))
4 end do
5 call cpu_time(stop_time)
6 write(*,*) "Repeated sine, Duration= ",stop_time - start_time

```

3. en utilisant une table de sinus :

```
1 call cpu_time(start_time)
2 do i=1,maxloops
3     b=i*sine_table(4)
4 end do
5 call cpu_time(stop_time)
6 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.

11.3.5 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'utilisant 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 6, p45] 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 :

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

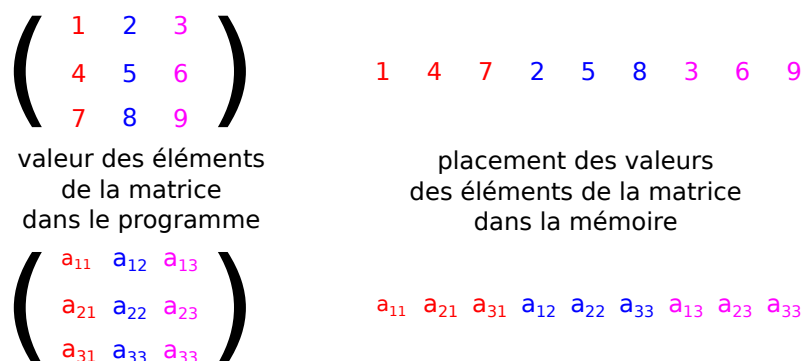


FIGURE 6 – Représentation de l'agencement en mémoire des valeurs associées aux éléments d'une matrice. L'élément a_{ij} (ligne i et colonne j) est noté $a(i,j)$ en **fortran 90**

```

1 call cpu_time(start_time)
2 do i=1,maxloops2
3     do x=1,2000
4         do y=1,2000
5             matrix(x,y)=x*y
6         end do
7     end do
8 end do
9 call cpu_time(stop_time)
10 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

```

1 call cpu_time(start_time)
2 do i=1,maxloops2
3     do y=1,2000
4         do x=1,2000
5             matrix(x,y)=x*y
6         end do
7     end do
8 end do
9 call cpu_time(stop_time)
10 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.

11.3.6 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 [§ 11.3.5, p44].

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

11.4 Débugger des programmes fortran avec gdb

Afin d'utiliser un débogueur 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**.

11.4.1 Débuguer à l'aide d'un core dumped

Parfois, il arrive qu'un programme plante et laisse un message du style

```
line 3: 37036 Abandon (core dumped) /home/login/bin/mercury/mercury
```

Il est possible de lire ce fichier **core.37036** (dans mon cas) à l'aide de **gdb** en lançant une commande de ce style :

```
gdb /home/login/bin/mercury/mercury -c core.37036
```

où **/home/login/bin/mercury/mercury** est le chemin absolu vers le binaire concerné, ici **mercury**.

Vous devriez obtenir des informations du style :

```
GNU gdb (GDB) Fedora (7.2-26.fc14)
```

```
[.
```

```
.
```

```
.]
```

```
Core was generated by '/home/cossou/bin/mercury/mercury'.
```

```
Program terminated with signal 6, Aborted.
```

```
#0 0x00000038fdc34085 in raise () from /lib64/libc.so.6
```

```
Missing separate debuginfos, use: debuginfo-install glibc-2.12.90-21.x86_64
```

```
libgcc-4.5.1-4.fc14.x86_64 libgfortran-4.5.1-4.fc14.x86_64
```

En entrant la commande **where** vous aurez, avec un peu de chance, des informations sur l'endroit du plantage :

```
(gdb) where
#0 0x00000038fdc34085 in raise () from /lib64/libc.so.6
#1 0x00000038fdc35a36 in abort () from /lib64/libc.so.6
#2 0x00000038fdc7156b in __libc_message () from /lib64/libc.so.6
#3 0x00000038fdc76e26 in malloc_printerr () from /lib64/libc.so.6
#4 0x000000000424aa6 in __algo_hybrid_MOD_mdt_hy ()
#5 0x000000000404810 in mal_hcon.1587.clone.3 ()
#6 0x00000000040db8d in MAIN__ ()
#7 0x00000000040f43f in main ()
```

Mais ces informations ne sont pas suffisantes. Pour avoir plus d'information, il suffit de recompiler votre programme avec l'option **-g** (pour **gfortran**), option pour donner plus d'information à **gdb**. Pas besoin de réexécuter la simulation. J'ai pour ma part compilé à coté, et je n'ai même pas quitté **gdb** pendant le processus.

Puis, il faut charger le fichier binaire compilé avec l'option pour **gdb** en faisant dans **gdb** :

```
(gdb) file /home/cossou/bin/mercury/mercury
warning: exec file is newer than core file.
Reading symbols from /home/cossou/bin/mercury/mercury...done.
```

J'ai eu droit à ce warning, c'est normal, vu que je viens de le compiler. Mais je n'ai rien changé par ailleurs. Et un nouveau **where** me donne les informations suivante :

```
(gdb) where
#0 0x00000038fdc34085 in raise () from /lib64/libc.so.6
#1 0x00000038fdc35a36 in abort () from /lib64/libc.so.6
#2 0x00000038fdc7156b in __libc_message () from /lib64/libc.so.6
#3 0x00000038fdc76e26 in malloc_printerr () from /lib64/libc.so.6
#4 0x000000000424aa6 in algo_hybrid::mdt_hy (time=313506736, h0=8,
    tol=9.9999999999999998e-13, en=..., am=<value optimized out>, jcen=...,
    rcen=0.0050000000000000001, nbod=30, nbig=30, m=..., x=..., v=..., s=...,
    rphys=..., rcrit=..., rce=..., stat=..., id=..., ngf=..., dtflag=2,
    ngflag=0, opflag=0, colflag=0, nclo=0, iclo=..., jclo=..., dclo=...,
    tclo=..., ixvclo=..., jxvclo=..., _id=8) at algo_hybrid.f90:96
#5 0x000000000404810 in mal_hcon (time=313506736, h0=8,
    tol=9.9999999999999998e-13, jcen=..., rcen=0.0050000000000000001, en=...,
    am=..., cefac=3, ndump=500, nfun=100, nbod=30, nbig=30, m=..., xh=...,
    vh=..., s=..., rho=..., rceh=..., stat=..., id=..., ngf=..., opflag=0,
    ngflag=0, onestep=0x423e60 <algo_hybrid::mdt_hy>,
    coord=0x422f20 <algo_hybrid::mco_h2dh>,
    bcoord=0x422d20 <algo_hybrid::mco_dh2h>, _id=<value optimized out>)
    at mercury.f90:1256
#6 0x00000000040db8d in mercury () at mercury.f90:262
#7 0x00000000040f43f in main (argc=<value optimized out>,
    argv=<value optimized out>) at mercury.f90:114
#8 0x00000038fdc1ee7d in __libc_start_main () from /lib64/libc.so.6
#9 0x0000000004010e9 in _start ()
```

Le point crucial est que dans mon cas, je peux récupérer le temps précis du plantage, et ainsi relancer la simulation (je peux repartir d'un fichier dump juste avant le crash, donc quasi immédiat), mais exécuter avec une version modifiée de mercury qui me donne plein d'informations sur la simulation uniquement quand le temps est égal au temps du crash et un petit peu avant.)

Le temps est ici un paramètre de la routine `mdt_hy (time=313506736 ...)` ce qui me permet de rajouter le bout de code suivant dans mon programme :

```
if (time.gt.313506725) then
  open(10, file='leak.out', status="old", access="append")
  write(10,*) time, planet, time_mig, time_ecc, time_inc
  close(10)
```

```
write(*,*)  
call print_planet_properties(p_prop)  
end if
```

11.4.2 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
```

11.4.3 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.

11.4.4 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.

11.4.5 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

Remarque : En compilant avec **gfortran** et l'option **-fbounds-check**, il faut lancer l'exécutable et les tests seront effectués au cours de l'exécution si j'ai bien compris, même si des tests sont effectués au cours de la compilation aussi.

11.5 Erreurs de compilation

11.5.1 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`.

11.5.2 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.

11.5.3 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.

11.5.4 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.

11.5.5 Error : Rank mismatch in array reference at (1) (2/1)

J'avais cette erreur parce que j'avais déclaré la variable comme

```
1| real, dimension(:), intent(in) :: matrix
```

au lieu de

```
1| real, dimension(:, :), intent(in) :: matrix
```

Il lui manquait donc une dimension. Et ça buguait quand je voulais faire :

```
1| temp = matrix(i,j) + 1.
```

11.5.6 Error : Rank mismatch in argument 'levels' at (1) (1 and 0)

Dans la routine, j'avais

```
1| real, dimension(:), intent(in) :: levels
```

et j'essayais de passer en argument

```
1| call contour(levels=0.)
```

Le problème vient du fait qu'on ne peut pas passer un scalaire pour un argument défini comme étant un tableau, il ne comprend pas que ça peut être un tableau à un élément. Je n'ai pas vraiment résolu le problème puisque je l'ai contourné en définissant l'argument comme scalaire. Je ne sais pas comment faire pour qu'on puisse passer une seule valeur. Peut-être définir un tableau à un seul élément via `dimension(1)`.