RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ « Dr. TAHAR MOULAY » DE SAIDA

FACULTÉ DES SCIENCES

Notes de Cours

Initiation au Langage de Programmation Fortran 90

(Première Partie)

Auteur : Habib BOUTALEB

Pour les étudiants de : Département de Physique, de Chimie et du Tronc-commun

Contact auteur : boutaleb.habib@gmail.com

Année Universitaire 2018 – 2019

Préface

L'objectif principal de ces notes de cours est de disposer les étudiants du département de Physique d'une formation en langage de programmation Fortran. Pour cela, nous avons élaboré un plan et qui consiste à diviser ce travail en deux parties. La première partie est de niveau débutant à intermédiaire et au bout de laquelle l'étudiant serait rapidement capable d'écrire des programmes simples et de manipuler les différentes instructions Fortran, c'est ce que nous avons appelé *« la voie à accès rapide à la programmation Fortran »*. Cela dit que le cours va tenir compte principalement de la version 90 du Fortran (F90). Pour des raisons pédagogiques, nous avons vu qu'il est bien utile de rappeler de temps à autre quelques notions algorithmiques et de mettre l'accent sur la version du Fortran 77 (F77). C'est la version qui a précédé la version F90 et elle a été utilisée pendant longtemps par une grande majorité de scientifiques.

La deuxième partie (présentée séparément) c'est *« le niveau avancé »* et dans lequel nous allons étudier avec plus de détails et de perfection les instructions du Fortran 90. En particulier, nous entamons la programmation modulaire et nous développons beaucoup plus les instructions d'entrée et de sortie de format

dirigées. Nous montrerons également les nouveautés apportées par les versions ultérieures (norme 95, 2003 et 2008).

Durant la période du cours, des séances « des ateliers » TD-TP seront programmés afin d'assimiler correctement et plus aisément les différentes notions vues dans le cours.

Finalement, nous tenons à préciser que ces notes de cours sont principalement inspirées des références suivantes :

- Claude Delannoy, Programmer en Fortran, 2 édition Fortran 90 et ses évolutions.
- Patrick Corde et Anne Fouilloux, Langage Fortran, Institut IDRIS.
- Luc Mieussens, Cours de Fortran 90, Université de Bordeaux

Nous terminons par remercier messieurs le Pr. Elkeurti Mohammed et le Dr. Zemouli Mostefa pour leurs conseils et recommandations dans le but de mener à bien ce travail.

Habib BOUTALEB

Kadda AMARA

INTRODUCTION

1 Introduction

1.1 Bref historique :

Le projet de création du langage Fortran remonte à l'année 1954 par John Backus de la société IBM : Mathematical **FOR**mula **TRAN**slating System d'où le nom FORTRAN. En 1957, la première version du langage (compilateur, FORTRAN II) est livrée. Cependant cette version ainsi que la suivante sont restées internes à la compagnie IBM. Depuis cette version, des évolutions croissantes ont été apportées au Fortran, comme l'introduction de l'instruction Format, les sous-programmes, les fonctions, etc.

En 1978, le Fortran 77 (ou Fortran V) est sorti et constitue réellement la première étape vraiment importante de son existence. Car cette version a introduit pour la première fois le type caractère ainsi que l'amélioration des instructions d'entrée et de sortie. Il faut dire qu'à cette époque, Fortran n'est pas le seul langage de programmation disponible, mais il est l'un des langages les plus utilisés par les scientifiques, les ingénieurs et les techniciens. Face à ce succès grandissant, les comités X3J3 accrédité par ANSI et le comité WG5 accrédité par ISO ont eu la mission pour moderniser au Fortran 77 et lui permettre d'être à jour. Ceci a fini par donner naissance en 1991 à la norme internationale ISO/ANSI Fortran 90. Cette norme a fait du langage Fortran un vrai langage structuré et elle a amélioré ses

possibilités de programmation modulaire, en particulier grâce à la notion d'interface. De plus, cette norme a facilité la manipulation des tableaux particulièrement bien adaptées au calcul vecto1iel ou parallèle ainsi que les tableaux allouables (gestion dynamique des tableaux).

Par la suite, d'autres normes ont été introduites, mais elles ne se mesurent pas à la révolution apportée par le Fortran 90. On peut citer par exemple, la norme Fortran 95 qui a introduit l'instruction *forall* spécifique au calcul parallèle, la norme Fortran 2003 qui est orienté vers la programmation objet et finalement le Fortran 2008 qui a amélioré le calcul parallèle et le développement des gros programmes.

Dans ce cours, nous allons nous intéresser le maximum possible à la norme Fortran 90. Cependant les exercices peuvent être (au choix de l'enseignant) programmés dans la version 77 ou 90. Nous allons également, et avant de donner la syntaxe Fortran des différentes instructions, rappeler sa syntaxe en langage algorithmique.

Ces notes de cours s'adressent particulièrement aux étudiants utilisateurs du langage de programmation Fortran (départements de physique, de Chimie et étudiants de Tronc-commum). Autrement dit, nous employons sans beaucoup d'attention certains termes informatiques proprement dit. Par exemple, on ne fait pas la différence entre un compilateur et un traducteur ou bien le code assembleur, etc. Toutefois certains prérequis informatiques sont nécessaires pur débuter facilement le module (notions d'algorithme, répertoire, fichier, mémoire, etc.)

1.2 Quelques Notions importantes : algorithme, programme, compilation, etc.

Souvent, lorsqu'on est confronté à un problème scientifique, comme la résolution d'une équation différentielle par exemple, on commence d'abord par voir si le problème admet une solution analytique ou bien il nécessite une résolution numérique. Dans le premier cas, la résolution du problème ne fait intervenir que l'utilisateur. Par contre dans le deuxième cas, et parce que c'est problème numérique, alors on a besoin de répéter des actions ou bien des instructions un bon nombre de fois (10, 100, ou 1000 fois par exemple ou plus); alors dans ce cas on a besoin de la machine (l'ordinateur) qui va répéter ces actions sans le risque de commettre des erreurs humaines comme l'oubli ou une faute de calcul par exemple. En réalité, la machine ne fait qu'exécuter des ordres, elle ne peut pas imaginer ou bien trouver la solution. Donc, l'utilisateur doit tout d'abord imaginer une solution mathématique au problème posé ensuite il doit organiser cette solution mathématique en des étapes bien ordonnées, simples, élémentaires et bien compréhensible par l'agent qui l'exécute qui est en fait l'ordinateur ou la machine. Cette première étape s'appelle *l'algorithme.* Alors celui-ci c'est comme un langage (d'ailleurs certains l'appelle pseudo-code) qui doit respecter une certaines formes et des règles syntaxiques, sauf qu'il demeure un langage humains qui ne permet pas de communiquer avec la machine, parce que cette dernière à son propre langage, c'est le langage machine et qui n'est pas forcement compris par l'utilisateur. D'où la notion de langage de programmation (programme). Donc le programme est la traduction de l'algorithme dans un langage de programmation, sauf que celui-ci n'est pas compris par la machine et on a besoin *d'un traducteur*

(compilateur) qui traduit le programme dans le langage machine. Cette étape est appelée *la compilation*. A la fin de cette étape, on aura généralement deux fichiers : un fichier objet (le programme écrit dans le langage machine, exemple.o) et un fichier exécutable (exemple.exe) et c'est celui-ci qui va nous fournir les résultats. On peut schématiser cette explication par le schéma ci-après.

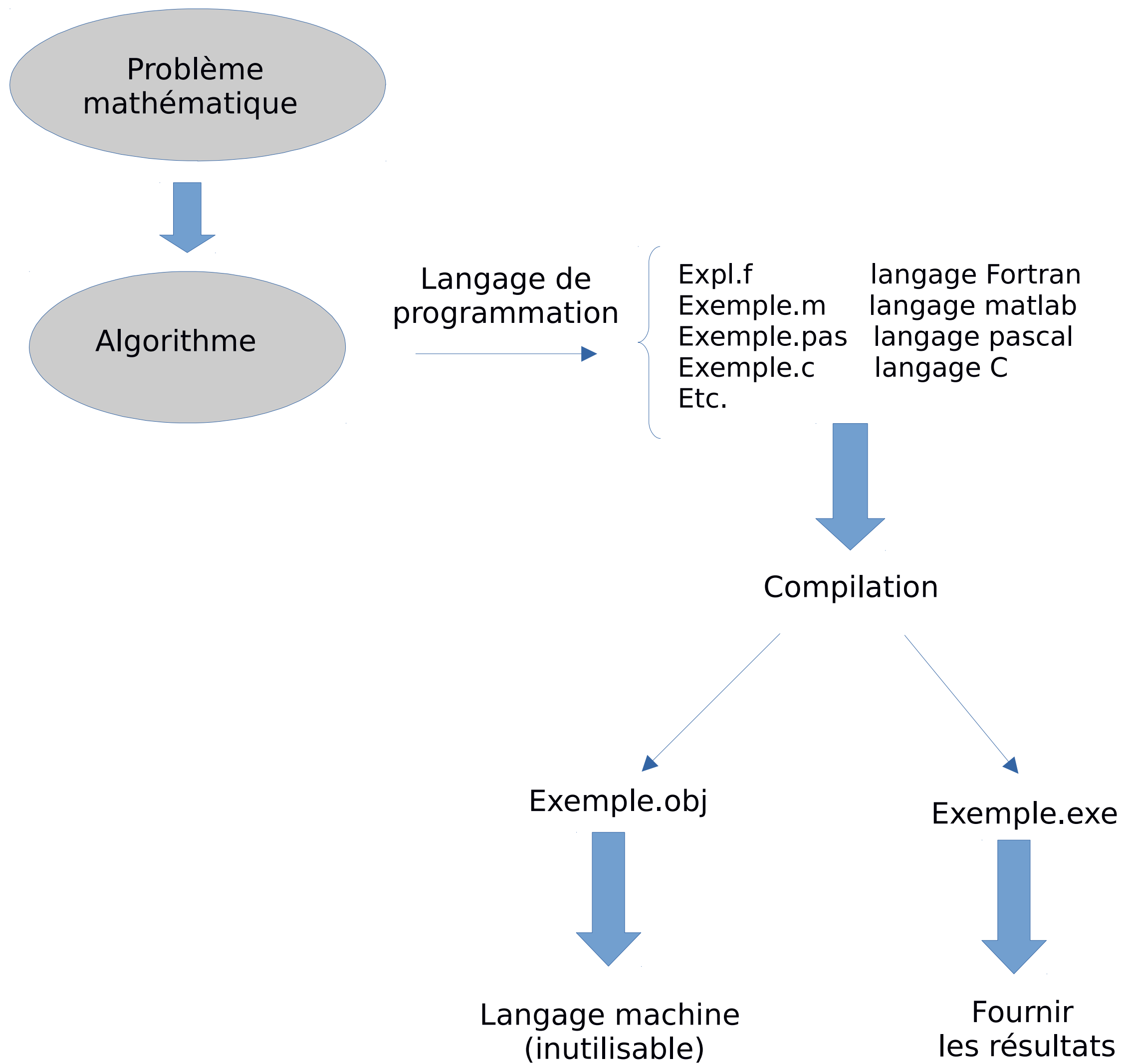

Schéma représentatif des différentes étapes de la conception d'un programme Fortran

1.3 Comment compiler un programme Fortran ?

Lorsqu'on travaille sous le système d'exploitation Windows, alors on bénéficie d'une interface graphique qui facilite grandement le travail avec Fortran. Alors après avoir installé le Fortran, une icône de celui-ci peut être placée sur le bureau. On fait un double clique sur cette icône et on voie apparaître la fenêtre Fortran. Cette fenêtre contient le menu presque ordinaire :

File, edition,……. Build … Help.

On suppose que le programme est écrit dans fichier qui porte le nom exemple.f ou exemple.f90. Après cela, on clique sur :

Build --- > Compile exemple.f

Dans cette étape, le compilateur fait une vérification syntaxique des différentes instructions utilisées par l'utilisateur, ensuite une vérification sémantique pour voir par exemple s'il y'a quelques choses qui n'est pas conforme avec les déclarations ou les affectations. Par exemple, déclarer une variable comme étant entière alors qu'on lui affecte une valeur réelle ou complexe. A la fin de cette étape, le compilateur fournit un rapport dans lequel sont notifiées les erreurs ainsi que leurs emplacements. On revient au programme source (exemple.f90) et on corrige ces erreurs. Il est fortement conseillé de corriger les erreurs diagnostiquées par le compilateur dans l'ordre d'apparition, car une première erreur peut en induire une quantité d'autres dans la suite du programme.

Il est aussi important de savoir que le fichier exécutable ne peut être créé qu'après la correction de toutes les erreurs (zéro erreurs), la présence des warning est acceptable. Après cela, on clique de nouveau sur Build ----- > build exemple.obj, c'est l'étape qu'on appelle étape de la création des liens avec des éventuels programmes ou subroutines. À la fin de cette étape et si tout va correctement, le fichier exécutable (exemple.exe) sera créé.

Par ailleurs, lorsqu'on travaille sous unix, la compilation se fait en utilisant des commandes tapées dans un terminal.

Si le programme Fortran est exemple.f90, alors la commande est : *gfortran -o exemple exemple.f90*, le fichier exemple est le nom du fichier exécutable, on peut mettre le nom qu'on souhaite. Pour exécuter le programme, on tape la linge de commande suivante : *./exemple*

Bien sûr ici la compilation et l'édition des liens est fait par une seule commande. On peut le faire sur deux étapes et ceci va être utile en cas de présence de sous programmes ou des subroutines et dans ce cas on doit les compiler séparément et créer les liens par la suite.

Compilation :*gfortran -c exemple.f90* : cette commande donne le fichier objet : exemple.o

Édition de liens : *gfortran exemple.o -o exemple.x*

Le fichier exemple.x est le fichier exécutable. Bien sûr dans la commande d'édition des liens, on doit mettre tous les fichiers objets des sous programmes ou des subroutines utilisés par les programme principal exemple.f90

(gfortran exemple.o subrout1.o subrout2.o - exemple.x)

II

Premiers Pas En Programmation

2 Premiers pas en programmation

Comme nous l'avons expliqué précédemment, pour résoudre un problème posé, on commence d'abord par résoudre mathématiquement le problème et organiser en des étapes sa résolution, c'est ce qu'on appelle l'algorithme. Par la suite, on traduit cet algorithme en un programme écrit dans un langage de programmation.

2.1 Définition d'un algorithme :

Un algorithme est une description complète et détaillée des actions à effectuer et ce dans un ordre bien déterminé. Ces actions doivent être simple, élémentaires (on ça ne nécessite pas qu'on les détaille davantage) et bien compréhensible par l'agent qui les exécute.

Pour quoi un algorithme? : C'est pour séparer entre l'analyse du problème et le codage. Donc pas de préoccupation de la syntaxe et c'est pour permettre de traiter le plus grand nombre de cas possibles.

On peut aussi parler de **l'organigramme** qui est une représentation graphique avec des symboles (carrés, losanges, etc.). Cette représentation offre une vue d'ensemble de

l'algorithme. Il faut noter que aussi bien l'algorithme que l'organigramme sont des langage humains (ne permettent pas de communiquer avec la machine) et universelles.

2.2 Définition d'un programme

Un programme est l'expression de l'algorithme dans un langage de programmation et ce dans le but de communiquer avec l'ordinateur (la machine). Un même algorithme peut être traduit dans plusieurs langages de programmation selon le besoin de l'utilisateur. Un langage de programmation est souvent menu d'un compilateur qui traduit le programme en question dans le langage machine et fournir un fichier exécutable, généralement séparé, pour l'obtention des résultats.

2.3 Notion d'unité de programme

Un programme Fortran est composé de parties indépendantes appelées unités de programme. Chaque partie est compilée de façon indépendante. Chacune admet son propre environnement. Cependant, il est possible que ces parties communiquent entre elles.

Les différentes unités de programme sont :

- le programme principal,
- les sous-programmes :
    - de type subroutine,
    - de type function,
- les modules,

- les block data.

Nous allons nous intéresser dans cette première partie à la description à l'unité du programme principal.

2.4 Structure générale d'un programme (programme principal)

program nom_du_programme

déclaration des variables

initialisation des variables

instructions

end program nom_du_programme

Les mots **program** et **end program** sont des mots clés réservés au langage Fortran. Le mot clé **program** sert à donner un nom au programme (pas au fichier). Cette instruction est facultatif. Par contre, le mot clé **end** (ou **end prgram**) est obligatoire, il marque la fin du programme principal.

Concernant, la structure de l'algorithme, elle est donnée par :

déclaration des variables

Début

initialisation des variables

instructions

Fin

2.5 Comment s'organise une page Fortran

En Fortran 90, on utilise généralement ce qu'on appelle le format libre (par opposition au format fixe utilisé en Fortran 77). Les lignes Fortran 90 comportent jusqu'à 132 caractères et peuvent comporter plus d'une instruction à condition de les séparer par un « ; » (point-virgule). Dans le cas ou une instruction comporte plus de 132 caractères alors celle-ci peut être coupé suivi du caractère « & » et on continue sur la ligne suivante. Dans le cas d'une chaine de caractères, le caractère « & » doit être mis à la fin de la ligne en-cours et au début de la ligne suite.

Le caractère «!» est utilisé pour écrire des commentaires. Tous les caractères qui viennent après ce caractère et à n'importe quel endroit sont ignorés par le compilateur. Les commentaires sont utilisés par l'utilisateur dans le but d'expliquer (pour lui ou un autre utilisateur) certaines instructions. **Remarque 1:** un bon programme est un programme bien commenté.

2.6 Exemple 1 :

Le but : approximation de $\sqrt{2}$ par la méthode de Newton.

Soit U_0 donné,

Calculer $U_{n+1} = \frac{U_n}{2} + \frac{1}{U_n}$pour n = 0 jusqu'à 10

Algorithme :

```
Variable n, entier
          u réel
Début
u =1.0
pour n =1 ,10
Faire
u=u / 2 . 0 + 1 . 0 / u
Fait
Ecrire (u)
fin
```

Traduction en Fortran :

```
program racine
implicit none

integer :: n  ! déclaration des variables
real :: u
u =1.0  ! – -----  initialisation

do n =1 ,10   ! début de la partie instruction – – boucle
u=u / 2 . 0 + 1 . 0 / u
```

```
end do

print* , ' approx . de sqrt( 2 ):' , u   ! — — affichage
end program racine
```

Dans cet exemple, nous découvrons les différentes parties de l'algorithme et du programme. On voie bien que ça commence par une partie déclaration, suivie d'une partie initialisation (si elle existe), ensuite une partie instructions (corps du programme).

Dans ce qui suit, nous allons détailler chacune de ces parties en commençant par la partie déclaration.

III

Les Déclarations

3. Déclaration

La partie déclaration est toujours placée au début de l'algorithme/programme. Dans cette partie, on « présente » tous « les objets » qu'on utilise dans l'algorithme/programme. Ces objets peuvent être des variables, des constantes symboliques, des procédures, des fonctions, noms de types, etc. La déclaration de ces objets consiste à les définir et ce en donnant leurs noms (identificateurs), leurs types et éventuellement leurs valeurs (initialisation). Ainsi, des règles s'imposent à l'élaboration des identificateurs ainsi que la syntaxe de la déclaration.

3. 1 Identificateur : on utilise un identificateur pour identifier (nommer) ou bien une variable, une constante (symbolique ou paramètre) ou une procédure, etc.

Il est constitué d'une suite de caractères alpha-numériques (lettres non accentuées), des chiffres et le signe underscore. Le premier caractère doit être une lettre et la longueur de l'identificateur (le nombre de caractère) ne doit pas dépasser 31 caractères *(63 à partir de Fortran 2003)*. On note qu'il n'y a pas de différence entre les lettres majuscules et les lettres minuscules.

Exemple 2

```
constante_gaz
pi2
Rk54
```

Identificateurs non valides

```
accentué
avec espace
Il_y_a_plus_de_trente_et_un_caracteres
_souligne_devant
1 _chiffre_devant
nom#alphanumerique
```

3.2 Les types d'une variable (objet) :

Le type d'une variable constitue la deuxième étape dans la déclaration. Lorsqu'on précise le type d'une variable, cela veut dire qu'on va lui associer ou plutôt lui réserver une place mémoire (un nombre d'octets), un mode de représentation interne, un intervalle de valeurs permises ainsi que l'ensemble des opérateurs agissant sur les variables ayant ce type.

déclaration	signification	ex. d'affectation
integer :: n,i	entier	n=10
real :: x,y	réel	x=1.0e-5
		y=1.0

```
complex :: z         complexe            z=cmplx(1.0,2.5e3)
logical :: b         booléen                   b=.true.
character :: c       caractère                 c='d'
CHARACTER(LEN=N) ch_car      ch_car une chêne de caractère
ou   CHARACTER*N ch_car       de longueur N ( N caractères)
```

Remarque 2 : la précision d'un réel simple est de 7 chiffres décimaux significatifs alors que celle d'un double est de 15.

Remarque 3 : Déclaration Implicit None

Il est bien prudent de déclarer toutes les variables pour éviter les oublis. Pour cela, on place au début du bloc déclaration l'instruction **implicit none.** Dans ce cas, les oublis sont alors détectés à la compilation. En absence de cette instruction, Fortran utilise une règle de typage implicite. En effet, il considère que toute variable dont le nom commence par une des lettres (i, j, k, l, m, n) est considérée comme entière et réelle dans le cas contraire.

Remarque 4 On peut imposer une règle de typage en utilisant l'instruction implicit comme suit, **exemple 3 :** implicit LOGICAL (z,x). Dans ce cas, toute variale qui commence par z ou x est considérée comme variable de type logique.

Remarque 5 : Les attributs

Chacun des types que nous avons mentionnés ci-haut peut être surchargé d'un des attributs suivants :

PARAMETER : pour désigner une constante symbolique (paramètre)

DIMENSION : pour montrer qu'il s'agit d'une variable pas simple, mais de type tableau.

SAVE : objet statique

EXTERNAL : procédure externe

INTRINSIC : procédure intrinsèque

Syntaxe :

type[, liste attributs ::] liste identificateurs

Bien sûr ici, la liste des attributs est optionnelle.

Exemple 4:

```
integer :: n   ! n est une variable de type entier.
real :: u   ! u est une variable de type réelle.
real , PARAMETER :: pi
integer , dimension ( -2:2) : : dim
real , dimension ( 1 : 3 , 0 : 4 , 1 : 2 0 ) : : tab
```

Remarque 6 : il faut dire que les deux « deux points :: » ne sont pas obligatoires sauf en cas d'initialisation pendant la déclaration. (a vérifier)

3.3 Cas de constantes littérales

Les constantes utilisées dans le programme ont aussi un des types mentionnés précédemment sauf que on ne les déclarent pas mais leurs écriture doit indiquer leurs types.

Constantes entières : 1 123 -28

Constantes réelles simple précision (point décimale ou bien la lettre E): 0. 1.0 1. 3.1415 31415E-4 1.6E-19
1E12 .001 -36.

Constantes réelles double précision :(obligatoirement la lettre D) : 0D0 0.D0 1.D0 1d0

3.1415d0 31415d-4 1.6D-19 1d12 -36.d0

Constantes complexes : 2.5+i s'écrira (2.5,1.) : (0.,0.) (1.,-1.) (1.34e-7, 4.89e-8) : un couple de deux réels.

Constantes chaines de caractères : c'est une suite de caractères encadrée par le délimiteur ' ou bien " :

```
CHARACTER(LEN=10) :: ch

ch = "Bonjour"

ch(4:7) = "soir"
```

3.4 Constantes symboliques : c'est lorsqu'on donne un nom symbolique à une constante littérale. Dans ce cas, cette constante doit être déclarée. Pour cela, on utilise l'attribut PARAMETER comme suit :

TYPE, PARAMETER :: n 1 = c 1 [, ..., n i = c i , ...]

ou encore PARAMETER (n 1 = c 1 [, ..., n i = c i , ...])

Exemple 5

```
PROGRAM constante
LOGICAL, PARAMETER :: VRAI=.TRUE., FAUX=.FALSE.
DOUBLE PRECISION :: PI, RTOD
PARAMETER (PI=3.14159265d0, RTOD=180.d0/PI)
...
END PROGRAM constante
```

IV

Opérateurs et expressions

4. Opérateurs et expressions

Comme c'est le cas en math, en Fortran on manipule plusieurs types d'opérateurs. Leurs rôles est d'agir sur les les objets (constantes, variables, expressions, etc) manipulés dans un programme pour fournir un résultat. Ces opérateurs sont regroupés selon leurs types. Nous commençons par :

4.1-a Les opérateurs arithmétiques :

Ce sont les opérateurs qui fournissent des résultats arithmétiques (numériques). Ces opérateurs sont :

(+) : addition, (-) : soustraction, (*) : multiplication, (/) : dévision, (**) : puissance.

Lorsqu'on écrit : A+B, alors A et B sont les opérandes et A+B est l'opération d'addition. La combinaison de ces opérateurs forme **une expression arithmétique.** Exemple : (A + B) * (C + D).

4.1.b Conversion implicite

Le type d'une expression arithmétique dépend des types de ses opérandes.

- si les 2 opérandes sont du même type alors l'expression arithmétique résultante sera de ce type.

- si les deux opérandes ne sont pas du même type alors

l'expression arithmétique sera évaluée dans le type le plus fort relativement à la hiérarchie suivante :

INTEGER < REAL < DOUBLE PRECISION < COMPLEX

Exemple 6

Expression	Valeur	type du résultat
99/100	0	INTEGER
7/3	2	INTEGER
(100*9)/5	180	INTEGER
(9/5)*100	100	INTEGER
99./100	0.99	REAL
99./100d0	0.99d0	DOUBLE PRECISION
(1.,2.)+1	(2.,2.)	COMPLEX

4.1.c Quelques pièges à éviter:

- soit l'expression arithmétique suivante : d = 1.d0 +5. **0.5

Alors dans cette expression 5. **0.5 est évalué en réel simple précision. Ensuite 1.d0+ 5. **0.5 est évalué en double précision. Donc on a déjà une perte de précision dans l'exponentiation si on veut que le calcul soit fait on double

précision. Pour cela, il est conseillé d'écrire : d = 1.d0 +5.d0 **0.5d0

- Opération puissance :

    - si r est un entier positif, alors

    x^r = x * x * ..... *x (r fois)

    x^{-r} = 1 / x^r

    - Pour r réel quelconque

    $x^r = e^{r \log x}$ si x est positif ou nul.

    X^r n'est pas calculable si x est négatif.

    - Ainsi, 2**3 est bien l'entier 8;
    - de même 2,5**2 sera un réel de valeur (approehée) 6,25.
    - 2**(-3)=1/8, c'est-à-dire O !
    - (-2)**3, celui-la est calculé par la formule $e^{3.\log(-2)}$ et il ne sera pas calculable.

4.1.d Conversion forcée par une affectation :

Bien que nous n'avons pas encore parler d'affectation, mais on peut dire que dans une affectation valeur = expression, le type de l'expression est forcément converti dans le type de la valeur.

Exemple 7:

Expression	Interprétation
x = 5	x = 5.0
N = 0.9999	N = 0
M = -1.9999	M = -1

Exemple 8

```
integer :: n=5, p = 10, q
real :: X = O.5, y = 1.25, z
y = n + p       !y reçoit la valeur réelle 15
p = X +y        !p reçoit la valeur entière 1 (conversion de
          1.75 en entier)
p = -X – y      ! p reçoit la valeur entière -1 (conversion
                 ! de -1.75 en entier)
```

4.1.e Ordre de priorité

Pour l'évaluation D'une expression arithmétique, les opérations sont effectuées selon l'ordre de priorité suivant : 1) la puissance **, 2) *, /, 3)+,-. Bien sûr on peut utiliser les parenthèses pour prioriser l'évaluation d'une certaine partie de l'expression.

4.2 Les Opérateurs relationnels

Ce sont des opérateurs qui permettent de comparer deux objets (variables, constantes ou expressions) entre eux. Le résultat de cette comparaison est de type logique. Ces opérateurs sont résumés dans le tableau ci-dessous.

Ancienne notation	nouvelle notation	interprétation
.LT.	<	inférieur
.LE.	<=	inférieur ou égal
.EQ.	==	égal
.NE.	/=	non égal
.GT.	>	supérieur
.GE.	>=	supérieur ou égal

Remarque 7: Ces opérateurs admettent des opérandes de type INTEGER, REAL ou CHARACTER (les caractères sont bien ordonnés, le code ascii par exemple). Seuls les opérateurs ==, /= peuvent s'appliquer à des expressions de type COMPLEX.

4.3 Les opérateurs logiques

ce sont des opérateurs qui agissent sur des opérandes de type logique et fournissent un résultat de même type. Ces opérateurs sont :

Opérateurs logiques	Interprétation
.NOT.	négation logique

.AND.	conjonction logique
.OR.	disjonction inclusive
.EQV.	équivalence logique
.NEQV.	non-équivalence logique

Remarque 8: Une **expression logique** est formée par des opérateurs relationnels, des opérateurs logiques ou les deux ensembles. Dans tous les cas, le résultat est de type logique.

4.4 Opérateur de concaténation :

C'est un opérateur qui agit seulement sur des variables ou des constantes caractères ou chaîne de caractères. Le résultat obtenu est une chaîne de caractères.

Expression	Interprétation
c 1 // c 2	concatène c 1 avec c 2

Exemple 9 :

```
'BON' // 'JOUR' --> 'BONJOUR'
CHARACTER(LEN=10) :: ch = 'BON'
ch = ch // 'JOUR'              ! <-- INVALIDE !!!
ch = TRIM(ch) // 'JOUR'        ! <-- VALIDE
! TRIM : retourne la chaıˆne de caractères transmise!!
! sans ses blancs de fin. TRIM('PICASSO^^^')= 'PICASSO'
```

4.5 Priorités des opérateurs

A la fin de cette partie, nous pouvons donner l'ordre de priorité selon lequel les opérateurs sont évalués. Dans l'ordre décroissant, cet ordre est donné par :

Opérateur	Associativité
**	D → G
* et /	G → D
+ et -	G → D
//	G → D
<, <=, ==	G → D
/=, >, >=	
.NOT.	G → D
.AND.	G → D
.OR.	G → D
.EQV. et .NEQV.	G → D

V

Les entrées et les sorties écran/clavier

5. Les entrées et les sorties écran/clavier

Dans n'importe quel algorithme ou programme on a besoin d'introduire les données qui nécessaires pour réaliser le calcul souhaité. C'est ce qu'on appelle les instructions d'entrées. A la fin du traitement, on a également besoin de visualiser les résultats obtenus, ou dans d'autres cas, afficher des messages de communications ou d'interactions. Pour cella, on utilise ce qu'on appelle les instructions de sortie. On note que nous allons dans cette première partie aborder les instrcuitions d'entrées/sorties seulement sur les supports écran/clavier et en format libre. Les instructions d'entrée/sortie sur les fichiers et avec un format dirigé serons abordé plus loin dans ce cours.

5.1 Lecture (entrée) au clavier des variables v1,v2,…vn

Algorithme : Lire(v1,v2,… vn)

Fortran : read*, v1,v2,… vn

ou **read**(*,*) v1,v2,… vn

5.2 Écriture (sortie) sur écran des expressions E1,E2,…En

Algorithme : Écrire (E1,E2,… En)

Fortran : Print*, E1,E2,… En

ou **write**(*,*) E1,E2,… En

Exemple 10

```
integer :: n
character ( len =15) :: pays
print*, ' quel age avez-vous ? '
read*, n
print* , ' vous ave z ' , n , ' ans '
print*, ' de q u e l pays e t e s vous ? '
read*, pays
print*, ' vous venez de', pays
```

5.3 Exercices avec solutions

Exercice 1

On considère en langage algorithmique les déclarations suivantes :

variables n,p **entier**

x **réel**

z **double précision**

n = 6;p=10;x=1.5;z=5.25

1. *Traduire cet algorithme en langage Fortran.*

```
integer :: n = 6, p=10
real :: x=1.5
double precision :: z=5.25
```

2. donnez le type et la valeur de chacune des expressions suivantes:

a) p*x-12 ! 10*1.5-12 = 15.0-12=13.0 real

b) z+n/p ! 5.25d0 + 6/10 = 5.25d0 double precision

c) x+(n+1.0)/p ! 1.5+(6+1.0)/10 = 1.5+7.0/10 = 2.2

d) (p – 7)**(n-4)**(p-8) !=3**2**2 = 81 integer

e) -p**(n-4) ! =-10**2 = -100

f) (-p+0.0)**(n-4) !n'importe quoi! , donc c'est incalculable

g) p>n! 10>6 ! logical .true.

h) z > x .and . p > n ! logical .true.

i) n == p .or . x > z ! logical .False.

j) .not. x == z .and . n==p ! logical .false.

k) .not. (x == z .and . n==p)! logical .true.

Exercice 3

Écrire un algorithme, puis le traduire en programme Fortran, qui lit une valeur réelle représentant la valeur d'un angle en degrés décimaux et qui affiche la valeur correspondante en degrés sexagésimaux (degrés, minutes, secondes). Rappelons que, par exemple:

50,26 degrés = 50 degrés. 15 minutes 36 secondes

solution 3

<u>Algorithme</u>

Variables angle,res_deg, res_min : **réel**

degres, minutes, secondes : **entier**

Début

Ecrire('donnez un angle (en degrés décimaux)')

lire(angle)

degres ---- > angle

res_deg ---- > (angle - degres) * 60.

minutes ---- > res_deg !ou minutes = int (res_degres)

res_min ---- >(res_deg - minutes)*60.

Secondes ---- > res_min

Ecrire(angle,'=',degres,'degres',minutes,'minutes',secondes,'secondes')

Fin

<u>Programme Fortran</u>

program conversion_angles

```
implicit none
real :: angle    !angles en degrés décimaux
integer :: degres, minutes, secondes !degrés,minutes et
                    !  secondes correspondantes
real :: res_deg, res_min
print*, 'donnez un angle (en degrés décimaux) '
read*, angle
degres = angle  ! ou degres = int(angle)
res_deg = (angle - degres) * 60.
minutes = res_deg  !ou minutes = int (res_degres)
res_min = (res_deg - minutes)*60.
secondes = res_min ! ou secondes = int (res_minutes)
print*,angle,'=',degres,'degres',minutes,'minutes',secondes,'seco
ndes'
end program conversion_angles
```

VI

Les instructions de contrôle

6. Les instructions de contrôle

Selon les problèmes traités par l'utilisateur, on est amené parfois à évaluer des situations ou des conditions et en conséquence choisir de réaliser telles ou telles instructions (**choix ou test)**. Dans d'autres situations, on est amené à répéter des instructions un certain nombre de fois ou dépendamment d'une condition **(itérations ou boucles)**. Ce type de problèmes est connu en informatique par ce qu'on appelle les instructions de contrôles. Nous allons dans un premier temps étudier les tests et qui sont exploiter en langage algorithmique par l'instruction **SI** et en Fortran par l'intruction **IF.** Dans la deuxième partie nous allons étudier les introductions répétitives et qui sont exploités en langage algorithmique par l'instruction **FAIRE** et en Fortran par l'instruction **DO.**

6.1 Les tests

6.1.1 Le choix simple, l'instruction IF

On considère l'exemple suivant : On veut calculer la valeur de y qui vaut xlog(x) si x#0 et zéro si x=0. Dans ce cas, nous remarquons que nous avons une condition sur x et que nous allons la tester. En fonction des résultats de ce test, on choisit la

valeur de y. Cela est réalisé en algorithme par l'instruction conditionnelle **SI** et qui a la syntaxe suivante :

```
Si (condition) alors
Bloc1 d'instructions
sinon
Bloc2 d'instructions
Fsi
```

- La syntaxe de cette instruction en Fortran est :

```
[nom]:IF (expression_logique)THEN
        Bloc1 d'instructions
        ! Exécuté si expression_logique est vraie
        ELSE nom
        Bloc2 d'instructions
        ! Exécuté si expression_logique est fausse
        ENDIF NOM
```

En réponse à l'exemple, on peut écrire :

```
if ( x /=0.0) then
y = x*log ( x )
else
```

```
y = 0.0
end if
```

- Dans le cas où il n'existe pas un traitement lorsque l'expression_logique est fausse, alors le Bloc 2 n'exsite pas et la syntaxe devient

```
IF (expression_logique)THEN
     Bloc1 d'instructions
     ! Exécuté si expression_logique est vraie
ENDIF
```

Lorsque l'expression_logique prend la valeur vraie, alors le Bloc1 est exécuté. Dans le cas contraire, alors se sont les instructions suivant le mot clé ENDIF qui seront exécutées.

- Le IF logique : Quand le bloc1 d'instructions ne comporte qu'une seule instruction alors la syntaxe se simplifie à :

```
IF (expression_logique)instruction
```

- **L'instruction elseif**

  Lorsque le bloc1 ou le bloc2 comporte à son tour des instructions de tests IF, alors il est possible d'utiliser une forme plus concise et dont la syntaxe est la suivante :

```
[nom :] if (exp_log_1) then
```

```
Bloc d'instructions
elseif (exp_log_2) then [nom]
Bloc d'instructions
ELSE [nom]
Bloc d'instructions
ENDIF [nom]
```

Exercice 4

Écrire un programme Fortran qui réalise une facturation avec remise. En effet, Il lit en donnée un prix hors taxes et il calcule le prix TTC correspondant (avec un taux de TVA constant de 19%). Il établit ensuite une remise dont le taux dépend de la valeur ainsi obtenue, à savoir:

- 0% pour un montant inférieur à 1000 dinars
- 1% pour un montant supérieur ou égal à 1000 dinars et inférieur à 2000 dinars
- 3% pour un montant supérieur ou égal à 2000 dinars et inférieur à 5000 dinars
- 5% pour un montant supérieur ou égal à 5000 dinars

Solution 4

```
program facturation_avec_remise
implicit none
real, parameter :: taux_tva = 19.0
real :: ht, ttc, net, tauxr, remise
print*, 'donnez le prix hors taxes :'
read*,ht
ttc = ht*(1. + taux_tva/100.)

If (ttc < 1000.)then
tauxr = 0.
     elseif(ttc < 2000.)then
tauxr = 1.
     elseif(ttc < 5000.)then
tauxr = 3.
     else
tauxr = 5.
ENDIF
remise =ttc*tauxr/100,
net=ttc-remise
print*,'prix ttc :    ',ttc
print*,'remise :',remise
```

```
print*,'net à payer :   ',net
end
```

6.1.2 Le choix multiple, l'instruction select case

C'est une instruction qui n'existe pas en Fortran 77 et elle a été introduite en Fortran 90. Cette instruction est préconisé lorsque l'évaluation de l'expression donne des résultats ou bien des discrets et qui appartient aux intervalles des instructions que nous allons réaliser dépendamment de cette évaluation. Avant de donner sa syntaxe, nous considérons l'exemple suivant :

Exemple 11

```
program exemple_select_case
implicit NONE
integer :: n
print*,'donnez un nombre entier'
read*,n
select case(n)!choisir un cas selon la valeur de n
     case(0)  ! n = 0
          print*,'il est nul'
     case(1,2) ! N = 1 ou 2
          print*,'il est petit'
     case(3:10) ! 3 ≤ N ≤ 10
          print*,'il est moyen'
```

```
        case(11 :) ! N ≥ 11
             print*,'il est grand'
        case default ! Tous les autres cas
             print*,'il est négatif'
    end select
    end
```

Dans cet exemple, en fonction de la valeur de n, nous choisissons une instruction parmi les instructions proposées. Si la valeur de « n » n'appartient pas à l'intervalle proposé, alors un traitement particulier est proposé par le mot clé **case default.**

D'une façon générale, on peut donner la syntaxe suivante

```
[ nom_bloc: ] SELECT CASE(expression)
[ CASE(sélecteurs) [ nom_bloc ]
bloc 1 ]
...
[ CASE DEFAULT[ nom_bloc ]
bloc n ]
END SELECT[ nom_bloc ]
```

- nom_bloc est identificateur pour identifier le bloc select case, il est facultatif,
- Expression est une expression scalaire de type **INTEGER**, ou une expression **de LOGICAL** ou **CHARACTER**,
- Sélecteurs est une liste de constantes du même type que

expression,

- Bloc i est une suite d'instructions Fortran.

Remarque 9: l'instruction select case trouve un intérêt particulier lorsque expression est de type caractère.

Exemple 12:

```
character(len=30) :: pays ! Pays est une variable
!chaine de caractères de longueur 30 caractères
...
langue : select case (pays) ! Expres. est de type caractères
case('france','alger','suisse','belgique')!les sélecteurs
print*,'bonjour'          ! instrcutions
case('royaume-uni','usa')
print*,'hello'
case default ! Si pays fait défaut
print*,'langue pas disponible'
end select langue ! Langue est le nom du bloc select case
```

6.2 Les itérations

Il est très fréquent de rencontrer dans des problèmes des opérations qui se répètent un certain nombre de fois ou dépendamment d'une instruction. C'est ce qu'on appelle les itérations. Ces itérations peuvent être réalisées ou bien avec des boucles avec compteur ou des boucles avec condition.

6.2.1 La boucle avec compteur : boucle DO

Exemple 13 : On veut calculer la somme y $=\sum_{i=1}^{n}\frac{x^i}{i}$

Algorithme	Programme
variables : i,n **entier**	integer : : i , n
x,y **réel**	**real** : : x , y
y -----> 0.0	y =0.0
pour i=1,n	**do** i=1,n
Faire	
y=y+x**i/i	y=y+x**i/i
fait	**enddo**
end	

Dans cet exemple, le problème tel qu'il est annoncé, ne montre pas que nous avons une répétition, d'ailleurs, on peut le résoudre sans avoir recours à une boucle DO (IF + GOTO). Mais l'utilisation de l'instruction d'affectation y=y+x ** i/i un certain nombre de fois peut effectivement résoudre le problème.

Ceci peut être abrégé par l'instruction DO et qui a la syntaxe générale suivante :

En Algorithme

Pour var=debut , fin [, pas]

Faire

instructions

Fait

En Fortran

[nom :] **do** var=debut , fin [, pas]

instructions

end do [nom]

var : identificateur d'une variable de type Integer (le type real est déconseillé à cause des problèmes liés à la précision). Var est souvent appelé variable de contrôle de la boucle. Elle doit être déclarée dans le programme au même titre que n'importe quelle autre. La valeur de var ne doit pas être modifié dans le bloc régit par la boucle.

Debut, fin et **pas** sont expressions scalaire de type Integer.

Si le **pas** est positif, alors début doit être inférieur à fin. Si le pas est négatif, alors c'est l'inverse. Si ces conditions ne sont pas vérifiées, alors la boucle est sauté et on

poursuit après le mot clé ENDDO. Dans le cas où le pas est omis, il est pris par défaut égal à 1.

Remarque 10 : La connaissance de ces trois paramètres **Debut, fin** et **pas** signifie que l'on connaît le nombre de répétition à effectuer. Dans certains problèmes le nombre de répétition n'est pas connu à l'avance mais il dépend d'une condition. C'est pour cette raison qu'il existe une autre façon d'exprimer les itérations, c'est le rôle de la boucle tant que en algorithme et la boucle DO WHILE en Fortran.

6.2.2 La boucle 'TANT QUE" : l'instruction DO WHILE

Dans ce cas, la syntaxe est donnée par :

En langage algorithmique :

Tant que (exp_logique)

Faire

bloc d'instructions

Fait

En Fortran, ceci se traduit par :

[nom :] **DO WHILE** (exp_logique)

bloc d'instructions

END DO [nom]

Remarque 11 :

- (exp logique) est une expression de type logique, donc elle doit avoir les deux valeurs .TRUE. Ou .FALSE.
- Le bloc d'instruction doit contenir une instruction qui doit modifier l'exp_logique, faute de quoi nous aurons une boucle infinie.
- Si au moment de rentrer dans la boucle, l'exp_logique prend la valeur .FALSE., alors le bloc d'instruction n'est exécuté aucune fois et ce sont les instructions suivant le mot clé ENDDO qui seront exécutées.

Exemple 14 : Écrire un programme qui calcule la somme de la série $y = \sum_{n\geq 1}\frac{1}{n^2}$. On s'arrête lorsque le terme général devient inférieur à ε fois la somme partielle courante.

Solution

```
PROGRAM iteration_while
INTEGER :: n
DOUBLE PRECISION :: terme, somme
DOUBLE PRECISION, PARAMETER :: epsilon = 1.d-3
LOGICAL :: fini
! Initialisation
n=0
somme=0.d0
fini=.FALSE.
```

```
DO WHILE (.not.fini)! exp_logique
n=n+1
terme = 1d0/n**2
somme=somme + terme
fini=(terme .LT. epsilon*somme)! Instruction qui
!Modifie l'exp_logique
END DO
print *,"Nombre d'itérations : ", n
print *,"Somme = ", somme
END PROGRAM iteration_while
```

6.2.3 Pour modifier le déroulement d'une boucle : les instructions exit et cycle

a) Sortie anticipée de boucle : l'instruction exit

C'est une instruction qui sert à interrompre le déroulement d'une boucle. Généralement, elle est utilisée avec l'instruction IF (IF logique simple). Sa syntaxe est la suivante :

EXIT [nom]

nom : nom d'une structure de boucle à l'intérieur de laquelle se trouve l'instruction **exit**

b) Bouclage anticipé : l'instruction cycle

L'instruction Cycle permet de rompre le déroulement "naturel" d'une boucle mais, cette fois, il s'agit simplement de passer "prématurément" à l'itération suivante. Sa syntaxe est :

cyle [nom]

nom : nom d'une structure de boucle à l'intérieur de laquelle se trouve l'instruction **cycle**

c) **L'instruction GOTO**

Cette instruction permet d'effectuer un branchement à un endroit particulier du programme, sa syntaxe est la suivante :

GOTO étiquette

étiquette : un numéro qui fait référence à une instruction

d) L'instruction STOP : C'est une instruction qui permet d'interrompre le programme.

6.2.4 Boucle sans la condition de contrôle (exp_logique)

C'est une forme de boucle introduite en Fortran 90 et qui paraît un peu curieuse parce qu'elle paraît infinie. Or dans ce type de boucle, on utilise généralement une des instrcutions permettant de modifier le déroulement d'une boucle, comme l'instrcution EXIT. Sa syntaxe est donnée dans l'exemple 15.

Exemple 15:

```
do
Bloc d'instructions
if (exp_log) exit
end do
```

ou sous une autre forme :

```
do                          ! cette forme est équivalente
if (.not. exp_log) exit     ! à une boucle DO WHILE
Bloc d'instructions
end do
```

VII

Les Tableaux

7 Les Tableaux

7.1 Tableaux statiques

Dans la vie quotidienne, nous ne manipulons pas seulement des variables simples, différentes et indépendantes les uns des autres, mais il arrive par fois que ces variables ont un lien qui les unissent, par exemple les notes des étudiants, les températures d'un matériau, les coefficients d'une série de Fourier, etc. Toutes ces variables vont subir par exemple un même traitement et donc il est plus avantageux de les regrouper dans un tableau. Afin de mieux comprendre comment les manipuler en Fortran, nous considérons l'exemple 16 suivant :

Exemple 16

```
program exemple_tableau
implicit none
integer, dimension (5) :: t !t est un tableau de 5 entiers
interger :: i
do i=1,5
print*,'donner un entier relatif'
```

```
read*, t(i) ! on lit l'élément de rang i du tableau t
end do
print*,' voici les nombres positifs que vous avez donné:'
do i=1,5
if (t(i) > 0) print*, t(i)
enddo
end
```

Dans cet exemple, nous utilisons un tableau nommé t et qui regroupe cinq éléments scalaires de type entier (Integer). Ces éléments, à part le fait qu'ils soient entiers, peuvent être des notes (de type Real) d'un élève, des températures relevées, etc. L'instruction **integer**, **dimension** (5) :: t est utilisée pour signifier en Fortran que nous avons un tableau et qui a les caractéristiques qu'on vient de mentionner. Donc c'est une instruction de déclaration de tableau. **Dimension**, en fait, c'est un attribut qui est employé ici avec le type Integer.

Par définition, un tableau est un ensemble ordonné d'éléments de même type désigné par un identificateur unique, t dans ce cas; chaque élément du tableau est repéré par un "indice" (nombre entier) précisant sa position au sein de cet ensemble, t(i) dans ce cas et il varie de 1 à 5.

D'une façon générale, pour déclarer un tableau, on utilise la syntaxe suivante :

TYPE, DIMENSION*(expr_1 ,...,expr_n) :: liste_tab*

avec :

- n ≤ 7 i.e un tableau peut avoir jusqu'à 7 dimensions
- expr_i sert à indiquer l'étendue dans la dimension correspondante. C'est une expression qui peut être spécifiée à l'aide :
  - d'une constante entière (peut être symbolique); dans ce cas, la borne inférieure du tableau est 1,
  - d'une expression de la forme cste_1 : cste_2 telle que cste_1 et cste_2 de type integer et $cste_1$ ≤ cste_2
- liste_tab est la liste de tableaux qu'on veut déclarer.

Dans cette première partie, nous allons utiliser principalement les tableaux statiques par opposition aux tableau dynamique. Ce dernier type a été introduit en Fortran 90 et il n'existait pas en Fortran 77.

Exemple 17

- integer, dimension (5) :: t

  Cette déclaration est équivalente à :

  integer, dimension (1:5) :: t
- real, dimension (-1: 12) :: v

  Dans ce cas, on spécifie que le premier élément est v(-1) et non pas v(1) comme s'était le cas pour « t »
- integer, parameter :: n_comp = 10, mini = -7, maxi = 15

  real, dimension (n_comp) :: vecteur

```
integer, dimension (mini:maxi) :: coef
integer, dimension (mini-1:maxi+1) :: ponder
```

- `real, dimension (3,5):: A`

  équivalente à : `real, dimension (1:3,1:5)::A`

- `real, dimension (-1:10, 0:10) :: B`

  B est un tableau de 132 (12 x 11) éléments, chaque élément étant repéré par deux indices, le premier allant de -1 à 10 et le second de 0 à 10.

Remarque 12

Il existe une autre façon, qui n'est pas recommandée, de déclarer un tableau sans l'attribut Dimension.

Syntaxe (exemple):

```
real, A(3,5)
```

7.2 Quelques définitions utiles pour les tableaux

- Le rang (rank) d'un tableau est son nombre de dimensions. Par exemple : REAL, DIMENSION (1:5,1:3) :: Y, Z. Alors Y et Z sont de rang égale à 2.
- L'étendue (extent) d'un tableau dans une dimension est le nombre d'éléments dans cette dimension. Par exemple : REAL, DIMENSION (15) :: X Alors l'étendue de X est 15. Y et Z ont une étendue de 5 et 3.

- Le profil (shape) d'un tableau est un vecteur dont chaque élément est l'étendue du tableau dans la dimension correspondante. Par exemple, le profil de X est le vecteur (/ 15 /), celui de Y et Z est le vecteur (/ 5,3 /)
- La taille (size) d'un tableau est le produit des éléments du vecteur correspondant à son profil. Par exemple, la taille des tableaux X, Y et Z est 15.
- Deux tableaux sont dits con formants s'ils ont le même profil. Les tableaux Y et Z sont conformants.
- L'instruction **where** s'emploie de la même façon que l'instruction IF mais seulement pour les tableaux. On verra ça en TD si l'occasion le permet.

  Syntaxe :

```
WHERE ( expression_logique_tableau)
   bloc1
   ELSEWHERE
   bloc2
End WHERE
```

7.3 Stockage des tableaux dans la mémoire et ordre des boucles

Lorsqu'on manipule de gros tableaux, il est très important de savoir comment ils sont stockés en mémoire. En effet, en mémoire la notion de tableau n'existe pas : les éléments sont rangés sous forme uni-dimensionnelle. Les éléments sont stockés ainsi, pour un tableau a(m,n) :

a(1,1), a(2,1), a(3,1), …, a(m,1), a(1,2), a(2,2), a(3,2), …, a(1,n), a(2,n), a(3,n), …, a(m,n).

Quand on parcourt les éléments d'un tableau, il faut que la boucle sur la première dimension soit la boucle la plus interne.

VIII

Les Sous programmes et les Fonctions

8. Les Sous programmes et les Fonctions

Il est très fréquent de rencontrer des programmes qui sont longs et peuvent facilement faire deux ou trois pages. Ainsi, il devient plus contraignant de le réviser pour détecter d'éventuelles erreurs ou les réadapter pour d'autres problèmes. Dans ce cas, il est plus judicieux d'organiser notre programme en plusieurs parties, qu'on appelle unités de programmes ou « procédures ». Ces unités sont composées d'un programme principal, des sous programmes, appelés également subroutines, et des fonctions. Dans le programme principal, on fait appel, au besoin, à ces subroutines et fonctions. L'avantage d'un tel découpage (nommé aussi programmation modulaire) est d'éviter la répétition des blocs d'instructions, ce qui permet d'avoir une meilleure lisibilité et plus de portabilité de nos programmes. Généralement, on peut distinguer deux types de sous-programmes (ou procédures) : les procédures externes et les procédures internes. Nous allons donc examiner chacun des deux types en commençant par des exemples. Concernant les fonctions, elles sont employées de la même façon que les procédures internes à quelques différences près. Nous les explicitons plus loin dans ce chapitre.

8.1 Procédures (sous-programmes) externes

Soit un sous-programme, on peut également dire une procédure, une subroutine ou parfois un module, nommé optimist. Sa structure ressemble à celle d'un programme, sauf que syntaxiquement, elle commence par le mot clé **subroutine** au lieu de **program** et se termine par **end subroutine** au lieu de **end program**.

Exemple 18

```
subroutine optimist (n_fois)
implicit none
integer,intent(in)::n_fois !déclaration de l'argument
                             !n_fois
integer :: 1 ! déclaration d'une variable locale
do i = 1, n_fois
print*,' il fait beau'
end do
end subroutine optimist
```

Dans cet exemple, on peut citer les points suivant :

- la subroutine comporte une déclaration un peu particulière : ***integer, intent (in) :: n_fois***. Il s'agit d'une déclaration d'un argument, en l'occurrence n_fois.Ces arguments n'ont d'importance qu'au sein de la définition du sous-programme

et servent à décrire seulement son travail. On les appelle arguments muets ou paramètres formels.

- L'utilisation de l'attribut ***intent(in)***. Cet attribut permet de specifier les parametres d'entree, de sortie ou les deux en même temps.
- Les variables locales au sous-programme, ici c'est la variable i. Ces variables ne sont connues que dans la subroutine où elles sont déclarées et ils n'ont d'intérêt que pendant leur exécution. On dit que leur "portée" est limitée à la subroutine en question.

Exemple 19

```
program exple_sous_progranme
implicit none
integer :: n = 2
print*,'appel optimist(n)'
call optimist (n)
print*,'appel optimist (2*n+1)'
call optimist (2*n+1 )
end
```

8.2 Appel de subroutine :

L'appel du sous-programme se fait en utilisant le mot clé **call**, mentionnant à la fois son nom et les arguments qu'on souhaite lui transmettre.

Les arguments utilisés lors de l'appel d'une subroutine sont appelés arguments "effectifs" (ou paramètres effectifs). Ici, nous avons pu utiliser une expression comme argument effectif; ceci était possible parce que l'argument correspondant a été déclaré comme paramètre d'entrée (intent(in)).

8.3 Procédure (sous-programme) interne

Dans ce cas, la subroutine se trouve dans le même fichier que le programme principal. Elle est située entre le mot clé **contains** qui annonce que le programme principal (son **hôte**) contient une subroutine et le mot clé **end.**

Exemple 20

```
program subroutine-interne
implicit none
integer :: n = 2
print*,'appel opttmist (n)'
call optimist (n)
print*,'appel optimist (2*n+l )'
call optimist (2*n+l)
contains
subroutine optimist (n_fois)
integer,intent (in) :: n_fois ! déclaration de l'
                    !argument n_fois
integer :: i     ! déclaration de la variable locale i
do i = 1, n_fois
print*,‘il fait beau’
end do
endsubroutine optimist
endprogram subroutine-interne
```

Exemple 21

```
program variables_globales
implicit none
real :: a=1.,b=2.,c=5.
real :: val1=1.5, val2=3.1
real :: res
call trinome(val1)
print*,'trirome (', val1,' )=', res
call trirome (val2)
print*,'trirome (', val2,')=', res
contains
subroutine trirome (x)
real, intent (in) :: x
res = a * x * x + b * x + c
end subroutire trirome
end program variables_globales
```

Sur ces deux exemples, on peut parler de variables locales et de variables globales. Dans l'exemple 20, la variable « i » est **une variable locale**, elle est définie (et n'a de portée) seulement dans la procédure interne qui la contient. En revanche, les variables a,b et c dans l'exemple 21 sont des **variables globales**. Elles sont définies dans le programme hôte et elles

sont également connues dans la procédure interne. Elle peut même en modifier la valeur.

8.4 Les arguments

Dans les exemples 20 et 21, les variables x et n_fois sont des arguments muets, elles déclarées à l'intérieur de la procédure interne. En Fortran 90, on peut utiliser l'attribut **intent (in)** pour dire qu'il s'agit des arguments d'entrées, donc leurs valeurs ne doivent pas être modifiées. **intent(out)** pour dire qu'il s'agit des variables de sorties, donc la procédure correspondante ne doit pas utiliser sa valeur du moment que cet argument doit recevoir une valeur. Enfin l'attribut **Intent(inout)** pour dire que les arguments en questions peuvent etre des arguments d'entrée et de sortie en même temps.

8.5 Les interfaces : Pour quoi écrire une interface ?

Généralement, il est préconisé d'écrire une interface pour pouvoir respecter les types des paramètres d'appels. Par exemple si dans une subroutine on déclare que n_fois est integer alors que pendant l'appel de la subroutine, on l'appelle pour une valeur réelle. Sans l'écriture d'une interface, la compilation ne signale aucune erreur, alors que si nous écrivons une interface la compilation nous signale bien une erreur.

Exemple 22

```
subroutine optimist (n_fois)

integer, intent (in) :: n_fois
```

```
. . . . . . . . . . .
call optimist (5.25)
```

Syntaxe d'une interface (ou bloc d'interface)

```
interface
subroutine optimist (n_fois)
integer, intent (in) :: n_fois
end subroutine optimist
subroutine sub2(z)!si on a une 2ième subroutine
real, intent (in) :: z
endsubroutine sub2
end interface
```

Remarque 13

Un bloc d'interface doit apparaître après une éventuelle déclaration **implicit none.**

8.6 Les Fonctions

L'utilisation des fonctions en Fortran se fait de la même façon que pour les sous-programmes. C'est-à-dire qu'on peut définir des fonctions internes et des fonctions externes (les deux cas sont des fonctions extrinsèques). Il existe par ailleurs des fonctions qu'on nomme des fonctions intrinsèques et qui sont reconnue par Fortran de façon automatiques, par exemple exp(x),

log(x), sin(x), etc. l'utilisation des ces fonctions intrinsèques se fait par son emploi que n'importe quel mot clé de Fortran. Quant aux fonctions extrinsèques, tout ce qui a été dit concernant les variables locales, les variables globales et les arguments restera valable. Pour la syntaxe, le mot clé **subroutine** va être remplacé par le mot clé **Function.**

Exemple 23

```
function trinome (a, b, c, x)
    ! ou encore real function trinome (a, b, c, x)
implicit none
real,intent(in) :: a,b,c,x  !déclaration des arguments
real :: trinome !  Déclaration du résultat
trinome = a * x * x + b * x + c
end function trinome
```

Remarque 14

- Dans cet exemple, le nom de la fonction, ici trinome, est utilisé pour fournir le résultat, d'ailleurs il a été déclaré comme real. On peut directement utiliser la déclaration mentionnée en commentaire (real function trinome (a, b, c, x)). Ce point là, on ne le trouve pas dans le cas d'une subroutine, parce que cette dernière ne retourne pas un résultat, mais des arguments de sortie. Donc pour résumer, une fonction retourne un résultat. Le Fortran 90 laisse la possibilité de donner un nom au résultat différent

de celui de la fonction. Nous verrons ceci quand l'occasion le permet.

- Afin de ne pas avoir un problème de type pour les arguments, il est possible d'utiliser une interface de la même façon que pour les subroutine, voir exemple 24.

Exemple 24

```
program exemle_fonction
implicit none
real :: a =1.,b = 2.,c = 5.
real :: x = 1.5
real :: y,z
interface
function trinome (a, b, c, x)
real, intent (in) :: a, b, c, x ! arguments
real :: trinome          ! resultat
end function trinome
end interface
y = trirome (a, b, c, x)
print*,'y = ', y
z = 2*(trinome(a, b, c, x)+trinome(a+1., b, 2*c, x+0.5))
print*,'z = ' , z
end program exemple_fonction
```

Table des matières

Préface 1

1 Introduction 3

1.1 Bref historique 3

1.2 Quelques Notions importantes : algorithme, programme, compilation 6

1.3 Comment compiler un programme Fortran ? 9

2 Premiers pas en programmation 12

2.1 Définition d'un algorithme 13

2.2 Définition d'un programme 14

2.3 Notion d'unité de programme 14

2.4 Structure générale d'un programme 15

2.5 Comment s'organise une page Fortran 16

2.6 Exemple 1 16

3. Déclaration 19

3. 1 Identificateur 20

3.2 Les types d'une variables 21

3.3 Cas de constantes littérales 24

3.4 Constantes symboliques 25

4. Opérateurs et expressions **26**

4.1-a Les opérateurs arithmétiques 27

4.1.b Conversion implicite 27

4.1.c Quelques pièges à éviter 28

4.1.d Conversion forcée par une affectation 29

4.1.e Ordre de priorité 30

4.2 Les Opérateurs relationnels 31

4.3 Les opérateurs logiques 31

4.4 Opérateur de concaténation 32

4.5 Priorités des opérateurs 33

5. Les entrées et les sorties écran/clavier **34**

5.1 Lecture au clavier des variables v1,v2,…vn 35

5.2 Écriture sur écran des expressions E1,E2,…En 36

5.3 Exercices avec solutions 36

6. Les instructions de contrôle **40**

6.1 Les tests 41

6.1.1 Le choix simple, l'instruction IF 41

6.1.2 Le choix multiple, l'instruction select case 46

6.2 Les itérations 49

6.2.1 La boucle avec compteur : boucle DO 49

6.2.2 La boucle 'TANT QUE" : l'instruction DO WHILE 51

6.2.3 Les instructions exit et cycle 53

6.2.4 Boucle sans la condition de contrôle (exp_logique) 54

7 Les Tableaux 56

7.1 Tableaux statiques 57

7.2 Quelques définitions utiles pour les tableaux 60

7.3 Stockage des tableaux dans la mémoire 61

8. Les Sous programmes et les Fonctions 64

8.1 Procédures (sous-programmes) externes 65

8.2 Appel de subroutine 66

8.3 Procédure (sous-programme) interne 67

8.4 Les arguments 69

8.5 Les interfaces 69

8.6 Les Fonctions 70