جامعة وهران للعلوم و التكنولوجيا محمد بوضياف

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie d'Oran « Mohamed BOUDIAF »
Faculté de Génie Mecanique
Département de Génie Maritime

Polycopié Pédagogique

Programmer en Fortran 77

Présenté par **:**

Dr ABED Bouabdellah

Année Universitaire : 2018/2019

Ce document pédagogique, est une introduction à la programmation par le langage **FORTRAN 77**, destiné aux étudiants du domaine Sciences et Technologies.

L'objectif de ce polycopié pédagogique est de fournir à l'étudiant une solide formation de base en informatique nécessaires à la programmation et à la mise en œuvre des algorithmes des différentes méthodes de résolution numériques en langage de programmation Fortran 77. Il traite essentiellement de la façon d'exprimer, dans ce langage scientifique, des algorithmes mathématiques simples.

Dr ABED Bouabdellah

Table des matières

Chapitre I Généralités 6
I.1. Le traitement de l'information 6
I.2. Présentation interne des données 8
I.2.1. Codage des entiers 9
I.2.2. Codage des réels 10
1.2.3. Codage des caractères 11
I.3. Introduction a FORTRAN 77 12
I.3.1. Historique 12
I.3.2. Définition et élément sémantique du Fortran 12
I.3.3. Elément syntaxique d'un programme Fortran 13
Chapitre II Constantes et Variables 16
II.1. Constantes 16
II.1.1. Constantes entières 17
II.1.2. Constantes réelles (simple précision) 17
II.1.3. Constantes double précision 18
II.1.4. Constantes complexes 19
II.1.5. Constantes logiques 19
II.1.6. Constantes chaînes 19
II.1.7. Déclaration des constantes 20
II.2. Les variables 20
II.2.1. Déclaration des variables 21
II.2.2 Déclaration des variables simples 22
Chapitre III Opérateurs et expressions 30
III.1. Instruction d'affectation 30

III.2. Opérateurs et fonctions mathématiques 33
III.2.1. Opérateurs arithmétiques 33
III.2.2. Ordre d'évaluation d'une expression arithmétique 34
III.2.3. Conversion des résultats obtenus par +, -, *, / 35
III.2.4. Fonctions Arithmétiques Intrinsèques 36
III.2.4.1. Fonctions Trigonométriques 36
III.2.4.2. Autres Fonctions 36
III.2.4.3. Minimum et Maximum 37
III.3 Expressions logiques 37
III.3.1. Définition 37
III.3.2. Opérateurs de comparaison arithmétique 38
III.3.3. Opérateurs logiques 39
III.3.4. Opérateur d'affectation 41
III.3.5. Opérateur de concaténation 42
III.3.6. Hiérarchie des opérateurs 42
Chapitre IV Instructions de contrôles 45
IV.1. But et fonctionnement des instructions de contrôles 45
IV.2. Les alternatives 45
IV.2.1. IF – Logique 46
IV.2.2. IF – Arithmétique 47
IV.2.3. Structure conditionnelle (IF/ELSE/ENDIF) 48
IV.2.4. Alternatives multiples 52
IV.3. Les ruptures des séquences 53
IV.3.1. Les ruptures internes 53
IV.3.1.1. GO TO - inconditionnel 54
IV.3.1.1. GO TO - calculé 55
IV.3.2. les ruptures externes 56
IV.3.2.1. Arrêt momentané 56
IV.3.2.2. Arrêt définitif 57
IV.4. les récurrences (répétition de séquences) 58
IV.4.1. Les récurrences fixes 58

IV.4.2. Les récurrences variables 60
Chapitre V Instructions d'Entrée et de Sortie 65
V.1. Généralités 65
V.2. Entrée/Sortie 65
V.2.1. Instruction PRINT 66
V.2.2. Instruction READ 67
V.2.3. Instruction WRITE 68
V.2.4. Cas des Variables Indicées 70
V.3. Instruction FORMAT 71
V.4. Spécificateurs 72
V.4.1. Spécificateurs d'édition de variables 73
V.4.1.1. Spécificateur I 73
V.4.1.2. Spécificateur F 74
V.4.1.3. Spécificateur E 75
V.4.1.4. Spécificateur D 76
V.4.1.5. Spécificateur G 77
V.4.1.6. Spécificateur logique 79
V.4.2. Spécificateurs de mise en page et d'édition 80
V.4.2.1. Spécificateur A 80
V.4.2.2. Spécificateur X 81
V.4.2.4. Spécificateur / (slash) 81
V.4.2.3. Spécificateur chaine de caractère ' ' 83
Chapitre VI SOUS-PROGRAMMES 84
VI.1. Généralités 84
VI.1.1. But des sous-programmes 84
VI.1.2. Différents types de Sous-Programmes en FORTRAN 77 86
VI.2. Sous-Programmes internes 86
VI.3. Sous-Programmes externes 88
VI.3.1. Sous-Programmes fonctions 88
VI.3.2 Sous-Programmes généraux 90
VI.4. Transmission de variables indicées en paramètres 93

VI.4.1. Dimensionnement Fixe 93
VI.4.2. Dimensionnement Variable 95
VI.5. Transmission de paramètres en zone commune 96
VI.6. Transmission d'un nom de sous-programme en paramètre 98
ANNEXES 101
Annexe A - Instructions 101
Annexe B - Fonctions intrinsèques 104
Références 106

Chapitre I Généralités

I.1. Le traitement de l'information

L'informatique : terme créé en 1962 par Philippe Dreyfus par la réunion des deux termes **information** et **automatique**. Il s'agit de la science du traitement automatique de l'information par ordinateurs. Elle mise en œuvre 3 composants principaux : les **algorithmes**, les **données**, et les **ordinateurs**.

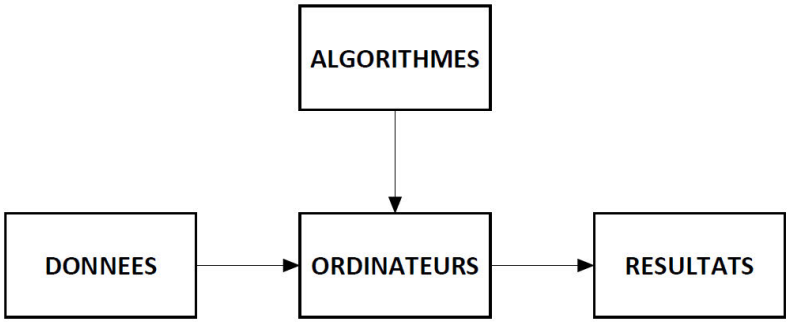
Les **ALGORITHMES** ce sont l'ensemble **d'instructions précises et ordonnées** qui vont être données à l'ordinateur pour exécution. Les algorithmes définissent les traitements appliqués aux données afin de résoudre une série de problèmes et d'obtenir les résultats souhaités.

Les **DONNÉES** ce sont l'ensemble d'informations qui vont être manipulées et traitées par les algorithmes.

Les **ORDINATEURS** se sont des dispositifs physiques de traitement qui vont exécuter automatiquement des informations selon les algorithmes sur les données pour obtenir les **RESULTATS** cherchés.

Il va donc falloir coder les données et les algorithmes pour les rendes compréhensibles par les ordinateurs.

Le codage de l'algorithme et de ces données associées se fait en utilisant un ***Langage symbolique*** indépendant de l'ordinateur utilisé. Cette opération s'appelle la ***programmation***, et le résultat de ce codage est un ***programme informatique***.



Un ***programme informatique*** est une suite d'instructions encodées, qui de manière très stricte respectent l'ensemble de conventions fixées par le langage informatique de programmation utilisé. Le programme est un fichier source écrit à l'aide d'un éditeur de texte.

Très souvent le ***langage de programmation*** informatique est normalisé (par Exemple le langage machine, le langage assembleur, le langage Pascal et le langage Python, le langage Fortran, le langage Pascal, le langage C, …). On en tire un avantage essentiel : la ***portabilité*** des programmes.

Le programme ainsi codé va donc être lu et traduit en langage internet par l'ordinateur pour ensuite exécuter l'ensemble des instructions en associant à chaque mot du langage informatique une action précise.

L'exécution sur un ordinateur d'un algorithme codé dans un langage symbolique impose une phase de traduction du langage symbolique dans le langage interne de l'ordinateur.

Pour exécuter cette transition il y a deux stratégies : la ***compilation*** et l'***interprétation***

La ***compilation*** consiste à traduire en langage interne l'ensemble du programme par le compilateur. Il produit une image du programme

compréhensible par l'ordinateur, et lui seul appeler programme objet. C'est ce programme objet qui sera exécuté ultérieurement par l'ordinateur.

L'***interprétation*** consiste à traduire et exécute immédiatement chaque ligne du programme. Cette façon de procéder présente l'avantage d'avoir une interprétation immédiate des instructions codées, donc éventuellement une correction tout aussi immédiate. Dans cette procédure, aucun programme objet n'est créé. L'interpréteur doit être utilisé chaque fois que le programme sera exécuté.

I.2. Présentation interne des données

Le seul langage que L'unité de codage Internet puisse véritablement comprendre est composé de bit (Un bit est un chiffre binaire 0 ou 1. C'est l'unité ´élémentaire d'information. En fait, on manipule rarement les bits un par un, ils sont regroupés en octets (8 bits). Pour des questions de performance les ordinateurs manipulent des ensembles octets appelée des mots (2 à 8 octets).

Dans le domaine scientifique les ordinateurs manipulent trois types de données élémentaires : les nombres entiers, les nombres réels et les caractères. Il est nécessaire de savoir comment sont codés ses différents types de données car cela induit pour certains d'entre des limitations.

Les techniques de codage utilisées sont multiples, elles varient entre particulier en fonction de type de l'application (scientifique ou de gestion pour des questions de performances.

Nous examinerons ici les techniques les plus couramment utilisées dans le domaine scientifique.

I.2.1. Codage des entiers

Les entrées représentent les entiers relatifs du langage mathématique usuel. Soit n le nombre de bits du champ de codage, le bit de gauche sert à codé le signe (0 → positif , 1 → négatif), les bits restant servent à coder le module du nombre.

- Les entiers positifs sont codés directement par leur équivalent binaire.

Exemple

Nombre	Code binaire
13	0 0001101
31	00011111
46	00101110

Nombre	Code binaire	Code inverse (complément à un)	Complément à deux
13	00001101	00001101	00001101
	00011111	00011111	00011111
46	00101110	00101110	00101110
-13	00001101	11110010	11110011

De façon générale, si on utilise un champ de codage de n bits, les entiers codés seront entre deux limites :

$$-2^{n-1} < \text{entier} < 2^{n-1} - 1$$

Exemple

Si n = 16 (codage sur 16 bits) :

-32768 < entier < 32767

Si n = 32 (codages sur 32 bits) :

-2 147 483 648 < entier < 2 147 483 647

I.2.2. Codage des réels

Les réels représentent en fait un sous-ensemble des rationnels du langage mathématique usuel. Par extension on assimile ces relations à l'ensemble des réels (du langage mathématique usuel)

Le codage des nombres réels se fait à partir de la représentation normalisée (notation scientifique ou virgule flottante), soit R un réel, on a :

$$R = (\text{mantisse}) \cdot 10^{(\text{exposant})} \quad \text{avec : } 0.1 < \text{mantisse} < 1$$

Le champ de codage sera donc partagé en deux sous-champs :

Un pour l'exposant (e), l'autre pour la mantisse (m). De plus, la base utilisée pour le codage et souvent prise égal à 16.

le codage des nombres réels se fait donc a partir de la forme :

$$R = (\text{mantisse}) \cdot 16^{(\text{exposant})} \quad \text{avec : } 1/16 < \text{mantisse} < 1$$

Pour le codage de l'exposant, on utilise la technique de codage des entiers.

Pour le codage de mantisse, on décompose la mantisse en une somme de la forme :

$$mantisse = \sum_{i=1}^{i=m/4} \frac{h_i}{16^i} \quad , \; 0 < hi < 15$$

1.2.3. Codage des caractères

on code chaque caractère dans un octet (8 bit), et on utilise une table de correspondance entre l'ensemble des caractères en leurs codes Internes. il existe deux principales table de correspondance la table utilisée sur les ordinateurs IBM et compatible. Il n'y a pas de limite inhérente au codage de caractère ceci appartenant déjà un ensemble fini.

I.3. Introduction a FORTRAN 77

I.3.1. Historique

Le langage de programmation ***FORTRAN*** a été conçu par *John Backus* en 1954 et la première version date de 1955. Avec les premiers ordinateurs commercialisés en 1956, le langage de programmation scientifique ***FORTRAN*** (FORmula TRANslator) est présenté au publique scientifique, et apparait le premier manuel de référence ***IBM*** qui définit la première version du langage : ***FORTAN I***. Par la suite, différentes versions sont apparues qui ont progressivement incorporé des améliorations et des extensions. En 1957, parution de la deuxième version ***FORTRAN II***. Ce FORTRAN reçoit les sous-programmes, les fonctions et les **COMMON**. En 1958, mise au point de la troisième version ***FORTRAN III***. En 1962, ***FORTRAN IV*** voit le jour avec en plus :

- Les déclarations explicites ;
- L'instruction **DATA** ;
- L'instruction **BLOCK DATA** ;

L'une des versions les plus importantes et les plus durables est le ***FORTRAN 77***, approuvé par l'***ANSI*** en 1977. Cette version a été acceptée dans le monde entier et est restée jusqu'à aujourd'hui en tant que FORTRAN universel. Ce dernier adopte un schéma d'alternative structuré (**IF-THEN-ELSE-ENDIF**) éliminant une bonne part des **GO TO** et il dispose des données de type **CHARACTER**.

I.3.2. Définition et élément sémantique du Fortran

Comme tout langage FORTRAN 77 utilise des mots. Ces mots peuvent représenter 4 types d'entités différentes, des constantes, des

variables, des fonctions ou des sous- programmes, des mots clés du langage.

Les données (constantes, variables, fonctions) peuvent être exprimées dans 6 types de modes différents

- le monde entier
- le mode réel
- le mode complexe
- le mode double-précision (réel, complexe)
- le mode logique
- le mode caractère

Ces mots sont rassemblés en instructions grâce à des règles syntaxiques.

Une instruction définie (ou codée) un certain traitement que l'on souhaite appliquer à des données. Un programme est constitué d'une suite destructions.

En fortran, l'exécution du programme et séquentiel c'est-à-dire que les instructions sont exécutés dans l'ordre où elles sont écrites, sauf directive contraire.

I.3.3. Elément syntaxique d'un programme Fortran

(a) l'alphabèt

Il se compose des :

- chiffres : **0 1 2 3 4 5 6 7 8 9**
- lettres : **a b c . . . x y z A B C . . . X Y Z**
- caractères : **- + / * , . ( ) " ' = espace**

(b) les mots

Ce sont des :

- mots clés : **DO , WRITE , IF , END ….**
- symboles : ensemble de **6 (six)** chiffre ou lettre maximum, le premier et en lettre.

Les symboles représentent des variables, des fonctions ou des sous-programmes, et sont créés par le programmeur en fonction de ses besoins.

L'INSTRUCTION

Elle s'écrit sur une ou plusieurs lignes. Elle comprend deux parties principales : l'étiquette (colonne 1 à 5) et l'instruction proprement dite (colonne 7 à 72).

1 5	6	7 72	73 ...
Zone étiquette	continuation	zone instruction	zone ignoré par le compilateur

L'étiquette est un nombre entier positif compris entre 1 et 99999.

La colonne 6 sert à indiquer qu'une ligne est la suite de la précédente. Il suffit pour cela qu'elle contienne un caractère quelconque sauf le 0 et le blanc.

Un programme écrit en langage Fortran 77 est l'ensemble des instructions comprises entre les mots clés : **PROGRAM** et **END** .

Exemple

```
PROGRAM
...
...
<Séquences d'instructions exécutables>
...
...
 END
```

Chapitre II Constantes et Variables

Dans un programme informatique, on va avoir en permanence besoin de manipuler et de stocker provisoirement des informations au cours d'un programme. Il peut s'agir de données fournies par l'utilisateur ou issues de l'ordinateur à partir des résultats intermédiaires ou définitifs obtenus par le programme. Ces données peuvent être de plusieurs types : Types numériques, Type alphanumérique, Type booléen.

Pour cette raison, on utilise des constances et des variables dans un programme informatique.

II.1. Constantes

Les constantes sont des données qui on ne peut pas les modifier pendant le déroulement d'un algorithme ou l'exécution d'un programme. Autrement dit une constante est une variable qui a une valeur définie et qui ne sera jamais modifiée dans le programme. Une erreur sera générée si la valeur d'une constante change dans le programme.

On peut par exemple utiliser une constante nommée ***pi*** pour stocker la valeur de **π**. Ainsi la constante ***pi*** contiendra toujours la valeur 3.141592.

```
pi = 3.141592
```

II.1.1. Constantes entières

Les Constantes entières représentent les nombres entiers relatifs. C'est une suite de chiffres précédée ou non d'un signe, ne pouvant comporter aucun autre caractère.

Exemple	Contre-exemple
123	3 14
-18	3.14
+4	2,71828

II.1.2. Constantes réelles (simple précision)

Les Constantes réelles représentent les nombres réels. Une constante de type réelle doit obligatoirement comporter :

- soit le point décimal, même s'il n'y a pas de chiffres après la virgule , pour la notation en virgule flottante (décimale).
- soit le caractère **E** pour la notation exponentielle (ou scientifique).

Pour les nombres écrits **0.xxxxx**, on peut omettre le **0** avant le point décimal.

Exemple

0.	0.0	.0		
1.	1.0			
0.001	.001			
-36.	-36.0	-36.00		
3.1415	31415E-4	31415E-04	314.15E-02	
1.E12	1.0E12	1.0E012	1.0E+12	1.0E+012
5.3E-8	5.30E-8	5.3E-08	-5.30E-08	-5.30E-008

II.1.3. Constantes double précision

Une constante **double précision** doit obligatoirement être écrite en natation exponentielle, la lettre **E** étant remplacée par un **D**.

Exemple

0. D 0	31415D-4	31415D-04	314.15D-02	
1.D12	1.0D12	1.0D012	1.0D+12	1.0D+012
5.3D-8	5.30D-8	5.30D-08	-5.30D-08	-5.3D-008

II.1.4. Constantes complexes

Une constante de type **complexe** est obtenue en combinant deux constantes réelles entre parenthèses séparées par une virgule. Les constantes complexes peuvent être de simple précision ou de double précision.

Le nombre complexe *2.5+i* **s'écrira en Fortran comme suit : (2.5 , 1.)**

Exemple

```
(0. , 0.)

(1. , -1.)

(1.34E-7 , 4.89E-8)

(1.34D-7 , 4.89D-8)
```

II.1.5. Constantes logiques

Une constante logique n'a que deux valeurs possibles :

```
.TRUE.

.FALSE.
```

II.1.6. Constantes chaînes

Elles sont constituées par une série de caractères encadrés par des apostrophes.

L'affichage d'apostrophes pose des problèmes.

Ce problème peut être réglé par le double apostrophage :

```
'Ceci est une chaîne'

'/home/louisnar'

'L''apostrophe doit être doublé'
```

II.1.7. Déclaration des constantes

Dans un programme, la déclaration des constantes permet de référencer une constante à l'aide d'un symbole. Elles ne peuvent être modifiées au milieu du programme et sont affectées une seule fois dans la section de 'déclaration' avec le mot-clé **PARAMETER** .

Syntaxe

```
PARAMETER (const1=valeur1, const2=valeur2, ...)

PARAMETER (max = 1000, q = 1.6D-19)
```

II.2. Les variables

Une variable est un emplacement en mémoire référencé par un nom, dans lequel on peut lire et écrire des valeurs au cours du programme.

Les variables permettent (entre autres) de :

- manipuler des symboles ;
- programmer des formules.

Avant d'utiliser une variable, il faut :

- définir son type ;

- lui donner un nom.

C'est la *déclaration*. Elle doit être écrite dans la première partie (la partie déclaration) d'un bloc fonctionnel *(programme principal, subroutine ou fonction)* dans lequel intervient la variable.

II.2.1. Déclaration des variables

FORTRAN est un langage permettant l'utilisation de 5 types de variables intrinsèques :

REAL	réels
INTEGER	entiers
LOGICAL	logiques
COMPLEX	complexes
CHARACTER	chaînes de caractères

La déclaration ce fait entre le mot-clé : **PROGRAM, SUBROUTINE ou FUNCTION** et la première instruction exécutable.

Syntaxe

```
TYPE VAR1, VAR2, VAR3, .....
```

(On peut déclarer plusieurs variables du même type sur une même ligne.)

Exemple

```
INTEGER     i,j,k
REAL        alpha, beta
DOUBLE PRECISION    x,y
COMPLEX     z
```

II.2.2 Déclaration des variables simples

2.2.2.1 Variables entières

La déclaration **INTEGER** indique que les variables et les fonctions nommées sont des données de type entières, et peuvent être définies par :

```
INTEGER, INTEGER*1, INTEGER*2, ou INTEGER*4.
INTEGER(1), INTEGER(2), ou INTEGER(4).
```

Syntaxe

```
INTEGER VAR1, VAR2, VAR3, .....
```

La table suivante présente les différents types des données entières, la mémoire occupée par chaque type, et le domaine d'utilisation de chaque type.

Exemple

```
INTEGER x
```

```
INTEGER x, y, z
```

type	Octet(s)	Rang
INTEGER(1)	1	-128 à 127
INTEGER(2)	2	-32,768 à 32,767
INTEGER(4)	4	-2,147,483,648 à 2,147,483,647

Table 2.1 Domaine d'utilisation des variables entières

2.2.2.2 Variables réelles

La déclaration **REAL** indique que les variables et les fonctions nommées sont des données de type réelles simple ou double précision, et peuvent être définies par :

```
REAL ou REAL*4                    simple précision
DOUBLE PRECISION ou REAL*8 double précision
```

Syntaxe

```
REAL VAR1, VAR2, VAR3, .....
```

Exemple

```
REAL x

REAL x, y, z
```

La table suivante montre les différents types des variables réelles la mémoire occupée par chaque type, la précision décimale, et le domaine d'utilisation de chaque type.

Type	octets	Précision	Rang
REAL(4)	4	6 chiffres	Nombres négatifs approximativement de -3.4028235E+38 à -1.1754944E-38 Le Zéro est un nombre positif approximativement de +1.1754944E-38 à +3.4028235E+38
REAL(8) or DOUBLE PRECISION	8	15 chiffres	Nombres négatifs approximativement de -1.797693134862316D+308 à -2.225073858507201D-308 Zéro est un nombre positif approximativement de +2.225073858507201D-308 à +1.797693134862316D+308

Table 2.2 Variables réelles

2.2.2.3 Variables complexes

La déclaration **COMPLEX** indique que les variables sont des données de type complexes simple ou double précision, et peuvent être définies par :

COMPLEX ou COMPLEX(4)	simple précision
DOUBLE COMPLEX ou COMPLEX(8)	double précision

Un nombre complexe est représenté par sa partie réelle et sa partie imaginaire. On le définit sous la forme suivante :

```
[Signe](Partie réelle, Partie imaginaire)
```

Exemple z =(5.0,1.0)

Syntaxe

```
COMPLEX VAR1, VAR2, VAR3, .....
```

Exemple

```
COMPLEX z

COMPLEX z1, z2, z3
```

2.2.2.4 Variables chaine de caractères

Syntaxe de déclaration :

```
CHARACTER*n VAR1, VAR2, VAR3, .....
```

où *n* représente la longueur de la chaîne. Cette déclaration réserve *n* octets en mémoire pour y stocker *n* caractères.

Exemple

```
CHARACTER*15 nom

CHARACTER*100 nomfichier
```

- Si l'on ignore la longueur de la chaîne, on pourra écrire la déclaration sous la forme :

```
CHARACTER*(*) chaine
```

- NOM et PRENOM sont deux chaînes de caractères pouvant comporter au plus 20 caractères..

```
CHARACTER *20 NOM,PRENOM
```

- NOM1, NOM2 ET NOM3 sont des chaînes de caractères d'une longueur maximale de 8 caractères, et ORIGIN est défini comme étant de longueur 15.

```
CHARACTER*8 NOM1, NOM2, ORIGIN*15, NOM3
```

2.2.2.5 Variables logiques

La déclaration **LOGICAL** indique que les variables et les fonctions nommées sont des données de type logiques , et peuvent être définies par :

```
LOGICAL, LOGICAL(1), LOGICAL(2), ou LOGICAL(4).

LOGICAL*1, LOGICAL*2, ou LOGICAL*4.
```

Syntaxe

```
LOGICAL VAR1, VAR2, VAR3, .....
```

Exemple

```
LOGICAL a

LOGICAL a, b
```

Le type de données logique est constitué de deux valeurs, **.TRUE.**et **.FALSE. (vrai et faux)**

Les variables entières et les variables réelles sont distinguées par le compilateur à partir de leur lettre initiale. Par défaut, toute variable dont le nom commence par la lettre ***I, J, K, L, M, N*** est considérée comme entière et les autres comme étant réelles.

On peut toutefois modifier cette règle en utilisant la déclaration **IMPLICIT.**

Syntaxe

```
IMPLICIT type (lettre1-lettre2), type (lettre3, ....)
```

type peut être **INTEGER, REAL, CHARACTER, COMPLEX**

Toute variable commençant par une lettre comprise entre lettre1 et lettre2 ou par lettre3 sera par défaut du type indiqué.

A noter que cette déclaration doit être écrite avant toute instruction exécutable.

Exemple

```
IMPLICIT REAL (a-c,e,w-z)
```

Tout ce qui commence par ***a,b,c,e,w,x,y,z*** sera **REAL**

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

tout ce qui commence par ***i,j,k,l,m,n*** sera **INTEGER**, tout le reste **DOUBLE PRECISION**

```
IMPLICIT COMPLEX (Z)
```

Tout ce qui commence par z est **COMPLEX** par défaut.

2.2.2.6 IMPLICIT NONE

IMPLICIT NONE spécifie que tous les variables et fonctions doivent être déclarés (type).

La déclaration **IMPLICIT NONE** doit précéder la déclaration **PARAMÈTER dans** le cas où les deux instructions apparaissent dans le même programme.

Chapitre III Opérateurs et expressions

III.1. Instruction d'affectation

L'instruction d'affectation consiste à calculer une expression et de stocker le résultat dans une variable (ou symbole).

L'expression définit le traitement appliqué aux opérandes (constantes, variables, fonctions) par les opérateurs.

L'expression et la variable affectée doivent être de modes compatibles (arithmétique, logique, caractère) pour que l'affectation ait un sens.

L'affectation se fait par le signe = , et elle se réalise à la compilation.

La syntaxe générale d'une instruction d'affectation est la suivante :

```
Variable = Expression
```

L'expression peut être formée d'une constante, d'une variable, ou formée d'une combinaison de constantes, de variables, de fonctions et d'opérateurs, qui définit une expression évaluable.

```
Variable = constante
```

La constante peut être d'un type quelconque : entier, réel, booléen, chaine de caractères, tableau, matrice.

Exemple

```
I     =  1
X     =  5.
Y(1)  =  24.0E-02
NAME1 = "ABED"
VRAI  = .TRUE.
FAUX  = .FALSE.
```

```
Variable = expression
```

Une expression est une formule pour calculer une valeur. L'expression peut être une opération logique, une opération arithmétique, un appel de fonction ou toute autre combinaison d'un ou plusieurs expressions évaluables.

Exemple

```
Aire    = a*b
Volume  = Aire * Hauteur
Delta   = b**2 - 4. * a * c
pivot   = A(I,K)/ A(K,K)
Name    = "ABED"    // " BOUABDELH"
ExpLog1 = a1 .OR. (b1 .AND. g1)
ExpLog2 = ((x1*x2)+x3).GT.x4).AND.x5
```

L'évaluation d'une expression s'effectue de gauche à droite, en respectant l'ordre des priorités. L'ordre d'évaluation pouvant être modifié par des parenthèses.

La variable peut varier au cours du temps.

Les noms attribués aux variables sont des identificateurs arbitraires. Il est judicieux de les choisir courts et explicites, de manière à exprimer clairement leurs références.

Les noms des variables doivent obéir à quelques règles simples :

- Un nom de variable peut être formée d'une suite de lettres (a. . .z , A. . .Z) et de chiffres (0. . .9), qui doit toujours commencer par une lettre.
- Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits.
- Les mots réservés par le langage ne peuvent pas être utilisés comme noms de variable.
- Le fortran ne distingue pas les lettres majuscules des minuscules.

Les opérateurs spécifient les actions à effectuer sur les opérandes.

III.2. Opérateurs et fonctions mathématiques

III.2.1. Opérateurs arithmétiques

Les expressions numériques exprimées dans Fortran sont formée d'opérandes et d'opérateurs, combinés d'une manière qui suit les règles de la syntaxe Fortran.

Une expression numérique est une expression dont les opérandes sont l'un des trois types numériques **entier**, **réel** et **complexe** de simple ou double précision.

Les opérateurs arithmétiques de fortran comme suite :

+	pour l'addition,
-	pour la soustraction,
*****	pour la multiplication,
/	pour la division,
******	pour l'élévation à la puissance (***I**2*** vaut I au carré, ***X**Y*** avec ***X*** et ***Y*** réels, vaut ***exp. (Y*Log X)*** si ***Log X*** a un sens).

Les Opérandes peuvent être :

- des constantes,
- des variables simples ou indicées,
- des fonctions,
- des Expressions complexes.

Exemple

```
3.14159

K

A(I)

SIN(A + B)

-1.0 / X + Y / Z ** 2
```

III.2.2. Ordre d'évaluation d'une expression arithmétique

L'évaluation d'une expression arithmétique se fait selon l'ordre établit suivant :

1. expressions secondaires entre parenthèses (nous commençons par la plus internes)
2. Fonctions numériques standard
3. élévation à une puissance, c.-à-d. augmentant à une puissance
4. multiplication et division (nous commençons de droite à gauche)
5. addition, soustraction, ou négation.

Exemple (l'ordre d'exécution)

```
(1)  A+B*C**2    1) C**2     2) B*C**2   3) A+B*C**2

(2)  A/(B*C)     1) (B*C)    2) A/(B*C)
```

III.2.3. Conversion des résultats obtenus par +, -, *, /

Les types sont classés par ordre croissant du rang :

1. **INTEGER*2** rang le plus faible
2. **INTEGER*4 (INTEGER)**
3. **REAL*4 (REAL)**
4. **REAL*8 (REAL DOUBLE PRECISION)**
5. **COMPLEX*8 (COMPLEX DOUBLE PRECISION)**
6. **COMPLEX*16** rang le plus élevé

C'est l'opérande de rang le plus élevé qui fixe le type du résultat, c.-à-d. au cours d'un calcul, si les opérandes sont de types différents, l'opérande du type le plus faible (occupant le moins de place mémoire) sera converti dans le type de l'opérande le plus fort.

Exemples

1. Pour les opérations d'addition de soustraction et de multiplication, et si les opérandes sont des entiers, le résultat sera un entier :

   *3 + 4= 7 , 4 * 3 = 12 , 3 - 4 = -1*

2. pour la division le résultat donnera la partie entière du quotient :

   4 / 3 = 1 , 3 / 4 = 0 , - 9 / 2 = -4

3. pour le mode mixte il faut faire intention :

   (9/2)+6.2=10.2 (9./2)+6.2=10.7

III.2.4. Fonctions Arithmétiques Intrinsèques

III.2.4.1. Fonctions Trigonométriques

ASIN(X) : arc sinus

ACOS(X) : arc cosinus

ATAN(X) : arc tangent

SIN(X) : sinus angle en radians

COS(X) : cosinus angle en radians

TAN(X) : tangente angle en radians

SINH(X) : sinus hyperbolique

COSH(X) : cosinus hyperbolique

TANH(X) : tangente hyperbolique

III.2.4.2. Autres Fonctions

ABS(X) : valeur absolue

SQRT(X) : racine carrée

LOG(X) : logarithme népérien

LOG10(X) : logarithme décimal

EXP(X) : exponentielle

III.2.4.3. Minimum et Maximum

MIN(X1,X2) minimum de 2 réels
MAX(X1,X2) maximum de 2 réels

III.3 Expressions logiques

III.3.1. Définition

Une expression logique permet de faire une comparaison entre deux expressions numériques (ou de type chaîne de caractères). Une expression logique élémentaire est composée de deux grandeurs arithmétiques liées par un opérateur de relation logique.

Constante arithmétique	**.opérateur de relation logique.**	Constante arithmétique
Variable arithmétique		Variable arithmétique
Expression arithmétique		Expression arithmétique

Le résultat d'une comparaison logique est de type logique ayant l'une des valeurs **.True.** ou **.False.** (.vrai. ou .faux.)

III.3.2. Opérateurs de comparaison arithmétique

Ces opérateurs admettent des opérandes de type `INTEGER`, `REAL` ou `CHARACTER`. Seuls les opérateurs `.EQ.` , `.NE.` peuvent s'appliquer à des expressions de type `COMPLEX` .

Leur résultat est un logique. Ils s'utilisent sous la forme :

```
OpérandeNumérique1 .OpérateurComparaison. OpérandeNumérique2
```

`.GT.` Pour strictement supérieur.

`.GE.` Pour supérieur ou égal.

`.EQ.` Pour égal.

`.NE.` Pour non égal c'est a dire différent.

`.LE.` Pour inférieur ou égal.

`.LT.` Pour strictement inférieur.

en Maths	en FORTRAN	en Anglais
$x = y$	X.EQ.Y	Equal
$x \neq y$	X.NE.Y	Not Equal
$x > y$	X.GT.Y	Greater Than
$x < y$	X.LT.Y	Less Than

$x \geq y$	X.GE.Y	Greater or Equal
$x \leq y$	X.LE.Y	Less or Equal

Exemple

```
X .EQ. Y

X est-il égal à Y ? Le résultat est logique : .TRUE. ou .FALSE.

I .NE. 10

B**2 -4.*A*C .GT. 0.
```

III.3.3. Opérateurs logiques

Leur résultat est un logique.

Syntaxe

```
.NOT. Opérande1
```

ou

```
OpérandeLogique1 .OpérateurLogique. OpérandeLogique2
```

Exemples

(a) L1 .AND. L2

Deux résultats logiques :

.TRUE. si L1 et L2 sont vrais ou

.FALSE. si l'un au moins des deux est faux.

(b) L1 .OR. L2

Deux résultats logiques :

.FALSE. si L1 et L2 sont faux ou

.TRUE. si l'un au moins des deux est vrai.

Tables de vérité

Opérateur de négation

L	.NOT. L
.TRUE.	.FALSE.
.FALSE.	.TRUE.

Autres opérateurs

L1	L2	L1 .AND. L2	L1 .OR. L2	L1 .EQV. L2	L1 .NEQV. L2
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

Exemple

(a) L1 .AND. L2

Deux résultats logiques :

.TRUE. si L1 et L2 sont vrais ou

.FALSE. si l'un au moins des deux est faux.

(b) L1 .OR. L2

Deux résultats logiques :

.FALSE. si L1 et L2 sont faux ou

.TRUE. si l'un au moins des deux est vrai.

III.3.4. Opérateur d'affectation

Elle est faite par l'instruction = :

nomVar = Valeur	*x = 2.*
nomVar = Expression Arithmétique	*x = y ou x = b**2. - 4.* a * c*
nomVar = Caractères	*nom = 'DJAMEL'*

Généralement, si le type de la variable est différent de celui de la valeur, la valeur subit un changement de type ''naturel'' (Valeur entière d'un réel, ...) pour la transformer dans le type de la variable.

Si la variable est de type CHARACTER alors la chaîne valeur doit être encadrée de '.

Exemple

```
nom = 'Djamel'

titre = 'L"Arbre'
```

Il faut doubler l'apostrophe pour qu'elle retrouve sa signification de caractère.

III.3.5. Opérateur de concaténation

Le seul opérateur des chaînes de caractères est l'opérateur (//) dite de concaténation. Il est employé pour enchaîner, ou joindre, deux chaînes de caractères pour n'en former qu'une.

Exemple:

```
CHARACTER*8   FIRST
CHARACTER*10  SECOND

FIRST  = 'GENIE '
SECOND = 'MARITIME'

PRINT*,  FIRST // SECOND
```

Le résultat est une chaîne à 18 caractères :

```
GENIE MARITIME
```

III.3.6. Hiérarchie des opérateurs

Dans une expression non parenthèse, l'ordre d'exécution des opérateurs suit l'ordre de présentation suivent :

1. les Opérateurs arithmétiques.
2. Les opérateurs de comparaison tous a égalité de priorité
3. **.NOT.**
4. **.AND.**
5. **.OR.**
6. **.EQV.** et **.NEQV.**

Dans le tableau suivant, les opérateurs sont donnés par ordre de priorité décroissante :

Opérateur	Associativité
**	Droite → Gauche
* et =	Gauche → Droite
+ et −	Gauche → Droite
==	Gauche → Droite
< , < = , = = / = , > , > =	Gauche → Droite
.NOT.	Gauche → Droite
.AND.	Gauche → Droite
.OR.	Gauche → Droite

.EQV. et **.NEQV.**	Gauche → Droite

Chapitre IV Instructions de contrôles

IV.1. But et fonctionnement des instructions de contrôles

Un algorithme décrivant un traitement se compose de deux types d'informations :

- celles concernant la nature des traitements à effectuer.
- celles indiquant quel l'enchaînement de traitements.

Les instructions de contrôles appartiennent à cette deuxième catégorie.

Ce sont elle qui vont permettre de donner une structure au programme et indiquer sous quelle condition on effectue ou pas telle ou telle séquence.

Un des apports clés de la recherche sur les langages de programmation se situe à ce niveau.

Il est donc essentiel, si l'on veut garder la lisibilité d'un programme, d'expliquer clairement dans l'algorithme et dans le programme, la structure de contrôle.

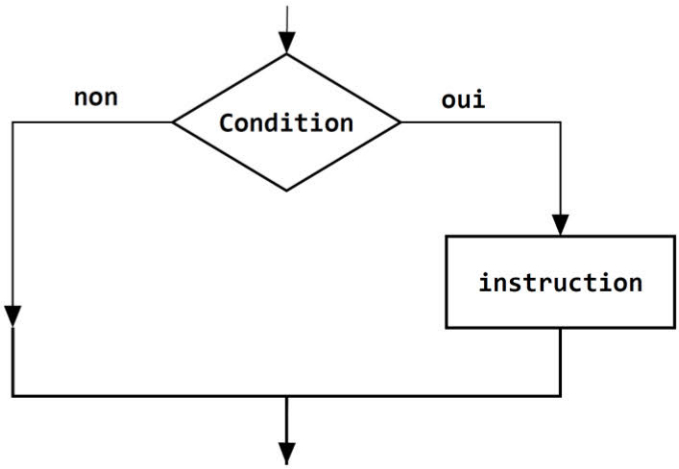
IV.2. Les alternatives

Par défaut, les instructions sont exécutées séquentiellement et une seule fois, dans l'ordre spécifié dans le programme. L'utilisation des

structures de contrôles permet de modifier le déroulement de l'exécution des instructions (le flux d'instructions).

IV.2.1. IF – Logique

Instruction exécute conditionnellement une seule instruction, selon la valeur d'une expression logique. Dans ce cas, la séquence se réduit à une seule instruction.



Syntaxe

```
IF (<expression logique>) <instruction>
```

L'instruction n'étant effectuée que si l'expression logique à la valeur `.TRUE.` , et le contrôle passe à l'instruction suivante dans le programme.

Toutefois, si l'expression logique est `.FALSE.`, l'instruction est ignorée et le contrôle passe à l'instruction suivante dans le programme.

L'instruction exécutable doit être différente des instructions DO, DO WHILE, ELSE, ELSE IF, END, END DO, END IF, END FUNCTION, ou END SUBROUTINE.

Exemple

```
A = 4.
B = -2.
C = 12.
DELTA = B**2 - 4. * A * C
IF (DELTA .LT. 0.0) PRINT*,' IL N'Y A PAS DE SOLUTION DANS R'
IF (DELTA .EQ. 0.0) PRINT*,' IL Y A UNE SEULE SOLUTION DANS R'
IF (DELTA .EQ. 0.0) PRINT*,' IL Y A DEUX SOLUTIONS DANS R'
```

IV.2.2. IF – Arithmétique

Instruction transfère le contrôle à une des trois instructions étiquetées, selon le signe de l'expression arithmétique qu'elle est positive, nulle ou négative.

Syntaxe

```
IF (<expression>) <etiq1> , <etiq2> , <etiq3>
```

<expression> : est une expression **arithmétique** de type entière ou réelle simple - ou à double précision.

<etiq1> , <etiq2> , <etiq3> : Étiquettes d'instructions exécutables dans le même programme.

Une même étiquette d'instructions peut être utilisée plus qu'une fois.

Exemples

a) L'instruction **IF** suivante :

```
IF (n-10) 10 , 20 , 30
```

transfert le contrôle à l'instruction :

1 - **10** pour ***n < 10***, c.-à-d. ***n – 10 < 0***
2 - **20** pour ***n = 10***, c.-à-d. ***n – 10 = 0***
3 - **30** pour ***n > 10***, c.-à-d. ***n – 10 > 0***

b) L'instruction IF suivante transfert le contrôle :

```
IF (n-10) 10 , 10 , 30
```

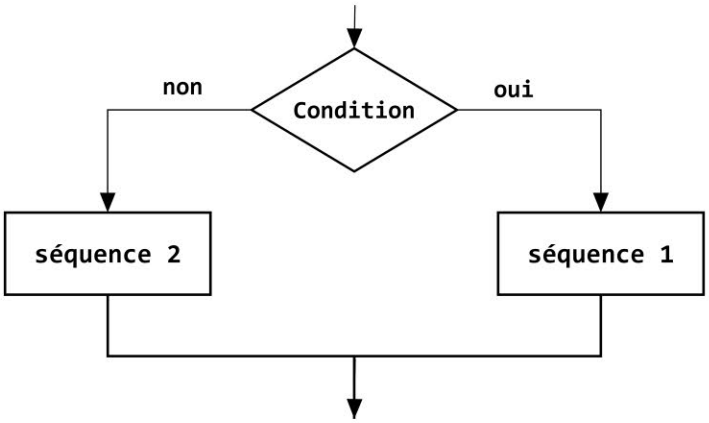
L'instruction IF suivante

1 - à l'instruction **10** pour **n <= 10**,

2 - à l'instruction **30** pour **n > 10**.

IV.2.3. Structure conditionnelle (IF/ELSE/ENDIF)

La structure conditionnelle binaire est la base de toute structure de contrôle. Elle correspond à l'organigramme suivant :



Si la condition est vraie on exécute la séquence 1, si elle est fausse on exécute la séquence 2.

Les séquences 1 et 2 pouvant être, soient de simples instructions élémentaires, soient des instructions plus complexes.

Syntaxe

```
IF (<expression logique>) THEN
      <Séquence 1 d'instructions exécutables>
  ELSE
      <Séquence 2 d'instructions exécutables>
ENDIF
```

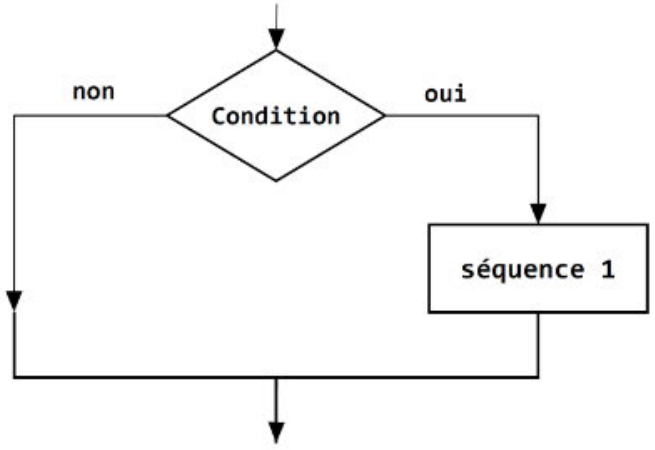
pour améliorer la lisibilité du programme, les instructions correspondantes aux séquences 1 et 2 sont décalées (tabulées) par rapport à celle de la structure de contrôle (`IF ..THEN..ELSE..ENDIF`).

Exemple

```
PROGRAM EQUADR
  IMPLICIT NONE

  REAL  A, B, C
  REAL  DELTA

  PRINT *, ' ENTREZ LES COEFFICIENTS A, B ET C :'
  READ *, A, B, C

  DELTA = B**2 - 4*A*C
  PRINT *, ' DISCRIMINANT = ', DELTA

  IF(DELTA .GE. 0.0) THEN
    PRINT*,' EQUATION ADMIT 2  RACINES REELLES'
   ELSE
    PRINT*, ' EQUATION ADMIT 2 RACINES COMPLEXES CONJUGUEES'
  END IF

STOP
END
```

Si les séquences 1 et 2 comportent aussi des alternatives on réapplique une tabulation pour chaque alternative.

Si l'une des alternatives et vide, on a l'organigramme suivant :



Syntaxe

```
IF (<expression logique>) THEN
      <Séquence d'instructions exécutables>
ENDIF
```

Exemple

```
      PROGRAM EQUADR
C     EQUADR : PROGRAMME POUR RESOUDRE L'EQUATION
C              DU DEUXEME DEGRE: A*X^2 + B*X + C = 0 DANS R
        IMPLICIT NONE
        REAL  A, B, C
        REAL  DELTA
        REAL  X1, X2 , X12
        PRINT *, ' ENTREZ LES COEFFICIENTS A, B ET C :'
        READ *, A, B, C
        DELTA = B**2 - 4*A*C
        PRINT *, ' DISCRIMINANT = ', DELTA
        IF ( DELTA .GT. 0.0 ) THEN
            X1 = ( -B + SQRT(DELTA)) / (2 * A)
            X2 = ( -B - SQRT(DELTA)) / (2 * A)
            PRINT *, " RACINES DISTINCTES : "
            PRINT *, X1, X2
        ENDIF
        IF ( DELTA .EQ. 0.0 ) THEN
             X12 = - B / (2 * A)
             PRINT *, " RACINE DOUBLE : "
             PRINT *, X12
        ENDIF
        IF ( DELTA .LT. 0.0 ) THEN
          PRINT *, " PAS DE RACINES REELLES "
        END IF
      STOP
      END
```

IV.2.4. Alternatives multiples

Ce cas peut être résolu en utilisant des alternatives binaires ; cependant on dispense, comme pour les organigrammes d'une structure particulière qui a l'avantage d'être plus simple et éviter des tabulations excessives.

Syntaxe

```
IF (condition 1) THEN
.. <Séquence 1>
ELSE IF (condition 2) THEN
.. <Séquence 2>
ELSE IF (condition 3) THEN
.. <Séquence 3>
ELSE
<Séquence else>
ENDIF
```

Le programme exécute la séquence 1 si la condition logique 1 est **.TRUE.** , puis reprend après ENDIF. Sinon, il exécute la séquence 2 si la condition logique 2 est **.TRUE.** , puis reprend après ENDIF. Sinon, il exécute la séquence 3 si la condition logique 3 est **.TRUE.** , puis reprend après ENDIF. Sinon, il exécute la séquence else, puis reprend après ENDIF.

Exemple

```
      PROGRAM EQUADR
C     EQUADR : PROGRAMME POUR RESOUDRE L'EQUATION
C              DU DEUXEME DEGRE: A*X^2 + B*X + C = 0 DANS R
C
      IMPLICIT NONE
      REAL  A, B, C
      REAL  DELTA
      REAL  X1, X2 , X12
      PRINT *, ' ENTREZ LES COEFFICIENTS A, B ET C :'
      READ *, A, B, C
      DELTA = B**2 - 4*A*C
      PRINT *, ' DISCRIMINANT = ', DELTA
      IF ( DELTA .GT. 0.0 ) THEN
          X1 = ( -B + SQRT(DELTA)) / (2 * A)
          X2 = ( -B - SQRT(DELTA)) / (2 * A)
          PRINT *, " RACINES DISTINCTES : "
          PRINT *, X1, X2
        ELSE IF ( DELTA .EQ. 0.0 ) THEN
          X12 = - B / (2 * A)
          PRINT *, " RACINE DOUBLE : "
          PRINT *, X12
       ELSE
       PRINT *, " PAS DE RACINES REELLES "
      END IF
      STOP
      END
```

IV.3. Les ruptures des séquences

Ces directives provoquent une rupture inconditionnelle dans l'enchaînement séquentiel des instructions du programme.

IV.3.1. Les ruptures internes

Cette instruction de ruptures internes se présente sous deux formes :

1. GO TO - inconditionnel,

2. GO TO - calculé (conditionnel).

IV.3.1.1. GO TO - inconditionnel

SYNTHAXE

```
GOTO <étiquette>
```

Quand on arrive sur cette instruction, on continue l'exécution à partir de l'instruction portant l'étiquette indiquée.

En FORTRAN 77, cette instruction est justifiée que dans trois cas :

- Le traitement des exceptions dans les ordres d'entrée-sortie.
- L'interruption prématurée d'une récurrence fixe
- Le re-bouclage d'un programme sur lui-même

Dans tous les autres cas, cette instruction est à proscrire car elle peut être une source d'erreurs, de non lisibilité, de non vérifiabilité du programme et en définitive de fatigue mentale excessif pour le résultat obtenu.

Exemple

```
      PRINT*, 'PROGRAMME RESOUT EQUATION DU SECOND DEGRE DANS R :'
      PRINT*, ' A*X^2 + B*X + C  '
      PRINT*, 'SAISIR LES 3 COEFFICIENTS REELS A , B ET C'
      READ *, A , B , C
      DELTA = B**2 - 4. * A * C
       IF (DELTA .LT. 0.) GOTO 10
       IF (DELTA .EQ. 0.) GOTO 20
       IF (DELTA .GT. 0.) GOTO 30
10    PRINT*, 'PAS DE SOLUTION REELLE'
       GOTO 100
20    PRINT*, 'SOLUTION DOUBLE :'
      PRINT*, ' X12 = ', -B/(2*A)
       GOTO 100
30    PRINT*, 'DEUX SOLUTIONS DISTINCTES'
      PRINT*, ' X1 = ', (-B + SQRT(DELTA))/(2*A)
      PRINT*, ' X2 = ', (-B - SQRT(DELTA))/(2*A)
100    CONTINUE
      STOP
      END
```

IV.3.1.1. GO TO - calculé

Instruction transfère le contrôle à une étiquette indiquée dans l'ordre des étiquettes.

Syntaxe

GOTO (<*étiq*>) [,] <*n*>

<*étiq*> : Une ou plusieurs étiquettes d'instructions exécutables dans le même programme, séparé (es) par des virgules. Une même étiquette peut apparaître plusieurs fois dans la liste <*étiq*>.

<*n*> : Un nombre entier.

Les éléments de la liste <*étiq*> représentent l'étiquette à laquelle le contrôle doit être transféré. Par exemple, si <*n*> est **4**, le contrôle est transféré à la 4ème étiquette dans la liste des <*étiquettes*>.

Remarques

- Si <*n*> est inférieure à 1 ou plus grand que le nombre d'étiquettes dans la liste, GOTO calculée agit comme l'instruction CONTINUE.
- En ne peut transférer le contrôle de l'extérieur vers l'intérieur d'un bloc DO, IF, ELSE IF, ou ELSE.

Exemple

```
      j = 2

      GOTO (10, 20, 30) , j

10     PRINT*,'10'
      GOTO 100

20     PRINT*,'20'
      GOTO 100

30     PRINT*,'30'
      GOTO 100

100   CONTINUE
```

IV.3.2. les ruptures externes

Elles correspondent à un arrêt momentané ou définitif de l'exécution du programme.

IV.3.2.1. Arrêt momentané

Syntaxe

```
PAUSE '<constante caractère>'
```

Cette instruction provoque l'arrêt de l'exécution du programme, elle est utile en phase de mise au point d'un programme, elle permet, après avoir inséré des instructions **PAUSE**, de suivre "à la trace" son exécution.

et l'apparition sur l'écran du message suivant :

```
<constante caractère>
```

Ou le message suivant si on omet <constante caractère> .

```
Pause - Please enter a blank line (to continue) or a DOS command.
```

Pour redémarrer le programme, il suffit d'appuyer sur la touche chariot du clavier.

IV.3.2.2. Arrêt définitif

Cette instruction est la dernière à exécuter. Elle doit donc logiquement apparaître au moins une fois dans un programme. Elle impose l'arrêt immédiat du programme.

Syntaxe

```
STOP '<constante caractère>'
```

Elle provoque l'apparition du message suivant à l'écran :

```
<constante caractère>
```

Ou le message suivant si on omet <constante caractère> .

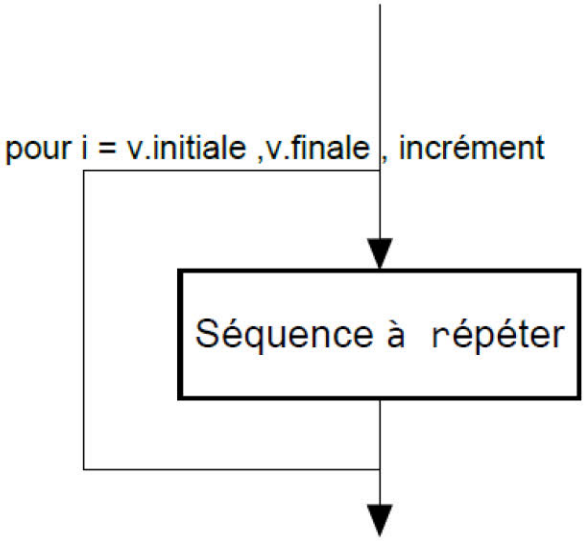
```
Stop - Program terminated.
```

IV.4. les récurrences (répétition de séquences)

Très fréquent ton traitement scientifique de récurrence correspond à des répétitions de séquences. Si le nombre de répétitions est connu à l'avance, la récurrence est dite fixe, sinon elle est dite variable.

IV.4.1. Les récurrences fixes

Elles sont utilisées dans les traitements de variables indicées (vecteurs, matrices ..), chaque fois que le nombre de répétitions (de boucles) est connu à l'avance, on a l'organigramme suivant :



Syntaxe

```
DO <compteur> = <Valeur initiale> , <valeur finale> , [<incrément>]
   .
   .
  Séquence à répéter
   .
   .
ENDDO
```

<*Compteur*> = <variable ou expressions entières> | <variable ou expression réelle>

<*valeur initiale*> = valeur initiale du compteur

<*valeur finale*> = valeur finale du compteur

<incrément> = incrément qui est rajouté au compteur à chaque boucle.

Exemple

```
DO I = 1, N + 1 , 2
    X1 = A(I + J)
    X2 = A(I + J)
    A(I) = AMAX (X1 , A2)
ENDDO
```

Remarques sur l'utilisation des boucles DO

- Une boucle DO et toujours effectué au moins une fois
- Il est interdit de modifier le compteur de la boucle dans la boucle (il ne doit jamais apparaître à gauche d'un signe égal).
- on peut arrêter une boucle avec un "GOTO" avant sa fin normale, mais il est interdit d'y arriver autrement que par l'instruction DO (sinon le compteur ne serait pas initialisé).
- en sortie d'une boucle le compteur a une valeur indéfinie.
- on tabule pour les questions de lisibilité, les instructions de la boucle par rapport aux instructions de contrôle (DO .. ENDDO).
- Si l'incrément est égal à 1, on peut l'omettre.

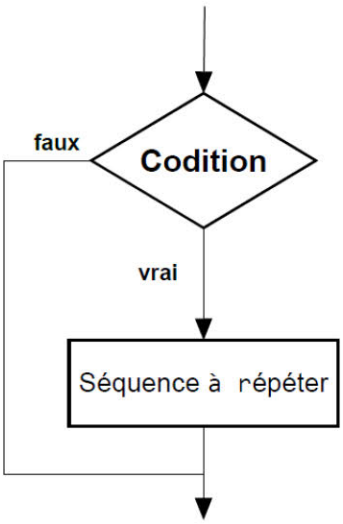
  ainsi: `DO INDICE = I + J , N , 1`

  est équivalente à : `DO INDICE = I + J , N`

La modification des bornes d'une boucle DO dans celle-ci, et sans effet sur le fonctionnement de cette boucle.

IV.4.2. Les récurrences variables

Elles correspondent à l'organigramme suivant :



SYNTAXE

```
DO WHILE (<condition d'exécution>)
   .
   .
    Séquence à répéter
   .
   .
ENDDO
```

Remarque sur <la condition d'exécution> :

Cette structure présente un risque qui est de voir la séquence se répéter indéfiniment si la condition ne prend jamais la valeur `.FALSE.` .

Ce cas de figure peut se présenter si la condition résulte d'un calcul, dans l'évolution n'est pas toujours convergente (Exemple recherche d'un zéro d'une fonction $f(x)$).

Dans ce cas pour éviter ce genre d'incident en rajoute à la condition d'arrêt normal une condition sur le nombre maximum de boucles à effectuer, pour être sûr que la boucle s'arrêtera obligatoirement.

Ainsi la condition sera de la forme :

<conditions> = <conditions d'exécution> **.AND.** <maximum de boucle non atteint>

à l'inverse de la boucle DO cette boucle peut n'être jamais exécutée si la condition d'exécution à la valeur **.FALSE.** dès la première itération.

Exemple

On veut calculer

$$e^x = \sum_{i=0}^{i=\infty} \frac{x_i}{i!}$$

en arrêtant la sommation quand le module du terme courant

$$|t_i| = \left|\frac{x_i}{i!}\right|$$

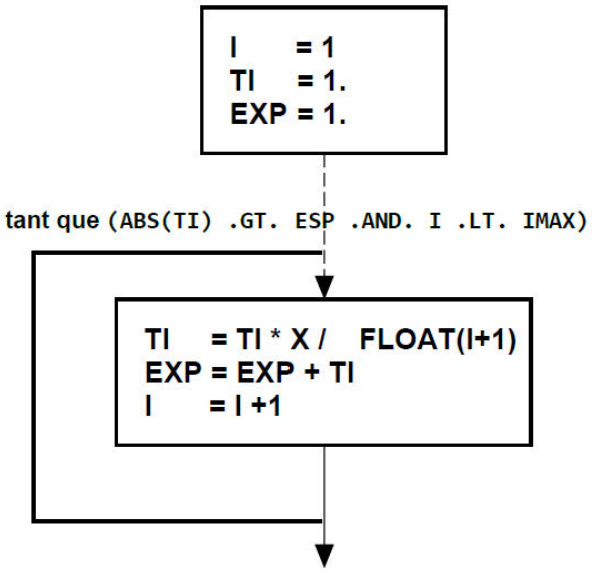
est inférieur à **esp** ou quand l'indice de sommation atteint **IMAX** pour éviter des calculs à l'infinie.

Pour le calcul pratique on améliore l'efficacité de l'algorithme en remarquant que :

$$t_i = t_i . \frac{x}{i+1}$$

Ce qui va limiter considérablement le nombre de calculs.

on suppose que les valeurs de **x** , **eps** et **IMAX** sont connus, (lus ou calculées auparavant).




```
      I    = 1
      TI   = 1.
      EXP  = 1.

C     Les initialisations sont faites
C     on commence les itérations.
C     noter que l'arrêt se fait quand
C     la condition prend la valeur  .FALSE.

      DO WHILE (ABS(TI) .GT. ESP .AND. I .LT. IMAX)
        TI = TI * X / FLOAT (I+1)
        EXP =EXP +TI
        I = I +1
      ENDDO
```

Chapitre V Instructions d'Entrée et de Sortie

V.1. Généralités

Le but d'une **E/S** est de faire communiquer le programme avec son environnement extérieur.

On dit qu'il y a entrée si le programme lit des données sur un périphérique externe (clavier, disques,…).

On dit qu'il y a sortie, si le programme écrit des données sur un périphérique externe (encrant, disques,…).

V.2. Entrée/Sortie

L' **E/S** consiste à indiquer quelles type de données on va traiter, comment on va les traiter et sur quelle fichier logique.

Il y a deux types d'instructions d'entrée et de sortie

- Les **E/S** en format libre.
- Les **E/S** contrôlées par un format (formaté).

Dans la réalisation d'une **E/S**, **4** instructions sont utilisées :

`PRINT` ou `WRITE` : impression (affichage) des informations (données) vers un périphérique ou un fichier.

READ : Lecture des données à partir d'un périphérique externe.

FORMAT : Format décrit de quelle façon doit se faire le codage et le décodage des informations.

V.2.1. Instruction PRINT

Instruction pour afficher des résultats sur l'écran (monitor).

Syntaxe

a) Avec format libre :

```
PRINT* , <listVar>
```

<list> : Suite de variables séparées par des virgules.

Exemple

```
PRINT*,' le résultat est : '
PRINT*, I , J , f
```

b) Avec format imposée :

```
PRINT <fmt> , <listVar>
```

fmt : Spécificateur de format.

Peut-être une ***étiquette*** de l'instruction FORMAT dans le même programme; ou une valide format.

<listVar> : Suite de variables séparées par des virgules.

Exemple

```
      PROGRAM CH4
C
      PRINT '(A6)' , ' ABED '
      PRINT 100    , ' BOUABDELLAH'
  100 FORMAT(A12)
C
      STOP
      END
```

V.2.2. Instruction READ

Instruction pour lire des données à partir d'un fichier logique spécifié.

Syntaxe

```
READ (<unit>, <fmt>) <ListVar>
```

<unit> : Variable entière indiquant un numéro de fichier logique de lecture. Si ***unit*** = *, la lecture se fait à partir du clavier, sinon elle se fait à partir d'un fichier de données.

<fmt> : Spécificateur de format. Peut-être une étiquette de l'instruction FORMAT dans le même programme; ou une valide **format** (spécificateurs).Ce format décrit de quelle façon doit se faire le codage des informations.

<ListVar> : Suite de variables séparées par des virgules.

a) *Avec format libre :*

Dans ce cas, le Spécificateur de format ***fmt*** = * .

Syntaxe

```
READ (<unit>,*)  <ListVar>
```

Exemple

```
REAL    x , y
INTEGER i , j
READ (*,*) x , y , i , j
```

b) *Avec format imposée :*

Dans ce cas, le Spécificateur de format ***fmt*** est défini par le programmeur.

Syntaxe

```
READ (<unit>, <fmt>) <listVar>
```

Exemple

```
      INTEGER i , j
      READ (*, 100) i , j
100   FORMAT (2I5)
```

V.2.3. Instruction WRITE

Instruction pour écrire des données sur un fichier logique spécifié.

Syntaxe

```
WRITE (<unit>, <fmt>) <listVar>
```

<unit> : Variable entière indiquant un numéro de fichier logique d'écriture.

<fmt> : Spécificateur de format. Peut-être une étiquette de l'instruction FORMAT dans le même programme; ou une valide **format**

(spécificateurs).Ce format décrit de quelle façon doit se faire l'écriture des informations.

<listVar> : Suite de variables séparées par des virgules.

a) *Avec format libre :*

```
WRITE (*,*) <listVar>
```

Exemple

```
WRITE (*,*) I , J , K
WRITE (*,*) ' Valeurs finales de I , J et K : ' , I , J , K
```

Dans cette exemple, ***unit*** = *, et ***fmt*** = *, l'écriture des variables i, jet k se fait sur l'écran en format libre.

a) *Avec format imposée :*

```
WRITE ( <unit> , <fmt> ) <listVar>
```

Exemple

```
PROGRAM CH41
M = 123456
WRITE (*, '(I8)') M
END
```

V.2.4. Cas des Variables Indicées

Très souvent on traite des variables indicés sous forme de vecteur ou de matrice. Pour les écrire de façon particulière, on utilise des boucles implicites de lecture ou d'écriture.

Exemple 1 : lecture/écriture en colonne du vecteur *V (k)*

```
READ (<unit> , <fmt>) (V (k), k=1, 3)
WRITE (<unit> , <fmt>) (V (k), k=1, 3)
```

Cette lecture/écriture est équivalente à :

```
Read (<unit> , <fmt>) V (1) , V (2), V (3)
WRITE (<unit> , <fmt>) V (1) , V (2), V (3)
```

Exemple 2 : Pour des matrices, on peut imbriquer les boucles pour une lecture/écriture ligne après ligne.

```
READ (<unit> , <fmt>)  (V(i, j), j=1,3), i=3)
WRITE (<unit> , <fmt>)  (V(i, j), j=1,3), i=3)
```

j : indice de colonnes

i : indice de lignes

V.3. Instruction FORMAT

L'instruction FORMAT est une instruction étiquetée non exécutable qui peut apparaître n'importe où dans le programme. Elle contient des spécificateurs d'édition qui définissent comment sont codées et décodées les valeurs des variables en sorties ou entrées avec les instructions PRINT , WRITE et READ . Elle précise la mise en forme des informations et leur type. L'instruction FORMAT, offre aussi la possibilité d'une mise en page agréable et lisible.

Syntaxe

```
<étiq> FORMAT (<spécificateur>)
```

<étiq> : permet d'associer ce FORMAT à un ordre d'Entrée/Sortie.

<spécificateur> : spécifie, selon le cas, de quelle façon la variable associée doit être lue ou écrite, et le cadrage, et la mise en page des informations.

Exemple

```
      OPEN (11, FILE = 'DATA.INPUT')
      OPEN (12, FILE = 'DATA.OUTPUT')

      PI = 3.14159

      READ  (11,*)   I
      READ  (11,102) I

      WRITE (*,100)  PI
      WRITE (12,100) PI

101   FORMAT (I3)
102   FORMAT (' LA VALEUR DE PI EST : ' , G13.6)

      STOP
      END
```

V.4. Spécificateurs

Dans la présentation qui suit, les lettres ***w***, ***m***, ***d*** et ***e*** sont des entiers non signés qui représentent :

w	la longueur totale de la zone en caractères (doit être supérieure à zéro), y compris le signe et le point décimal.
d	le nombre de chiffres à éditer à droite après le séparateur décimal (peut être zéro)

Tout spécificateur de données peut être précédé d'un nombre de répétitions ***n*** qui indique la fréquence de répétition du spécificateur.

n	facteur de répétition.

Dans les exemples suivants, les espaces sont désignés par ␣.

V.4.1. Spécificateurs d'édition de variables

V.4.1.1. Spécificateur I

Il s'applique aux variables entières qui ne peuvent pas contenir de points décimaux, de notation exponentielle ni d'autres signes de ponctuation (tels que des virgules). En plus, la longueur de la zone en caractères doit être suffisamment grande pour inclure un caractère du signe plus ou signe moins.

Syntaxe

[*n*] **I***w* [.*d*]

Ecriture ou Lecture de ***n*** entiers chacun sur ***w*** caractères. Chaque entier est justifié à droite dans sa zone en caractère avec les blancs non significatifs si nécessaire.

d : indique le nombre de chiffre a édité (en complète avec des Zéro si le nombre s'écrit avec en moins de ***d*** chiffres)

REMARQUE :

Si le nombre est supérieur aux nombres de positions indiquées par *w*, le nombre ne sera pas imprimé. Eventuellement, les positions seront remplacées par des astérisques ****.

Exemple

```
      PROGRAM CH5
      OPEN(1, FILE ='FORMAT.TXT')
      I =  -2019
      J =   2019
      WRITE(1,100) I , J
      WRITE(1,200) I , J
      WRITE(1,300) I , J
      WRITE(1,400) I , J
100    FORMAT(' ',I10,I10)
200    FORMAT(' ',2I10)
300    FORMAT(' ',2I10.8)
400    FORMAT(' ',I3, I17)

      CLOSE (1)
      STOP
      END
```

L'éditeur donnera :

```
␣␣␣␣␣-2019␣␣␣␣␣␣2019

␣␣␣␣␣-2019␣␣␣␣␣␣2019

␣␣␣␣␣-2019␣␣␣␣␣␣2019

␣0000-2019␣␣00002019

***␣␣␣␣␣␣␣␣␣␣␣␣␣2019
```

V.4.1.2. Spécificateur F

Il est utilisé pour la représentation des nombres de types réels en point fixe simple précision.

Syntaxe

```
[n]Fw.d
```

Règle générale :

w=d+p+2 (*p* : nombre de caractères de la partie entière)

Exemple

Soit le nombre -123.4567 à imprimer.

Nous obtiendrons:

F9.4	-123.4567
F11.4	␣␣-123.4567
F8.4	********
F13.6	␣␣-123.456700
F6.0	␣-123.

V.4.1.3. Spécificateur E

Il est utilisé pour la représentation des nombres de type réel en point flottant avec exposant. Cette notation en exponentielle est utile pour les très grands et les très petits nombres.

Syntaxe

```
[n]Ew.d
```

d : représente le nombre de chiffres de la partie fractionnaire avec ***d<w***.

La longueur de la chaine en caractère ***w*** doit accueillir :

- un signe plus ou moins à gauche au début de la chaine (optionnel),
- un chiffre (optionnel) devant le point décimal,
- le point décimal,

- le nombre de chiffres requis ***d*** après le point décimal,

- l'indicatif d'exposant E,

- le signe de l'exposant (plus ou moins) et

- le nombre de chiffres dans l'exposant ***e***.

En conséquence, ***w*** doit être un peu plus grand que d. En pratique, cela signifie que : ***w> d + 4 + 2***

Exemple

Soit à imprimer le nombre -123.4567.

E9.4	*********
E11.4	␣-.1235E+03
E8.4	********
E13.6	␣- .123457E+03
E6.0	0.E+03

V.4.1.4. Spécificateur D

Il est utilisé pour représenter des nombres en double précision. Notez que la notation D est identique à la notation E en entrée. En sortie, il produit un indicateur d'exposant D au lieu d'un indicateur E.

Syntaxe

```
[n]Dw.d
```

avec $w \geq d+6$

Exemple

Soit à imprimer le nombre -123.4567.

D9.4	*********
D11.4	␣-.1235D+03
D8.4	********
D13.6	␣-.123457D+03
D6.0	0.D+03

V.4.1.5. Spécificateur G

Le spécificateur G peut être utilisé si on ne connaisse pas à l'avance l'ordre de grandeurs de nos chiffres.

Si la valeur du chiffre a présenté est supérieure à 0.1 mais pas trop grande pour être présentée dans la zone en caractères, le nombre est alors écrit comme au format décimal (similaire au spécificateur ***F***) en utilisant la longueur de champ ***w*** et en générant ***d*** chiffres significatifs. Ce nombre est suivi de quatre blancs. Sinon, il se comporte exactement comme le spécificateur ***Ew.d***.

Syntaxe

```
[n]Gw.d
```

Exemple

```
      DOUBLE PRECISION PI

      PI =    3.14159
      R  = -100.6
      D  =  123.456789E-03
      H  =    2.99792458D+08

      WRITE(*,100)  PI , R , D , H
100   FORMAT(' ',4G10.3)
```

L'éditeur doit générer quatre nombres en virgule flottante au format à virgule fixe ou exponentielle. L'affichage est en fonction de l'amplitude du nombre. Le résultat est :

```
␣␣3.14␣␣␣␣␣␣-101.␣␣␣␣␣␣␣.123␣␣␣␣␣␣␣.300E+09
```

Pour homogénéiser les écritures de format, il est commandé d'utiliser les spécifications standards suivantes :

F10.0 (pour les entrées)

G13.6 (pour les sorties)

Notez qu'un **nombre complexe** nécessite deux spécificateurs de données, un pour la partie réelle et un pour la partie imaginaire. Ils peuvent être ***D, E, F, G*** ou n'importe quelle combinaison, à condition qu'il y en ait deux.

```
      COMPLEX T

      T = ( -15.8 , 30.16 )
      WRITE(*,100)  T
100   FORMAT(2F12.4)
```

Nous obtiendrons:

```
␣␣␣␣-15.8000␣␣␣␣␣30.1600
```

V.4.1.6. Spécificateur logique

Le champ de saisie doit contenir la lettre **T** ou la lettre F dans le champ en caractères *w* spécifiés. Les lettres T ou F peuvent être précédées d'un point et d'un nombre quelconque de blancs. Tout ce qui apparaît après les lettres T ou F est ignoré.

Le champ de sortie a une longueur w en caractères et est constitué de la lettre T ou de la lettre F précédée de (*w-1*) blancs.

```
      LOGICAL X,Y

      X = .FALSE.
      Y = .TRUE.
      WRTTF(*,100) X
      WRITE(*,200) Y
100   FORMAT('X =',L2)
200   FORMAT('Y =',L5)
```

A l'édition, nous aurons :

```
X =␣F
```

```
Y =␣ ␣ ␣ ␣T
```

V.4.2. Spécificateurs de mise en page et d'édition

V.4.2.1. Spécificateur A

Syntaxe

```
[n]Aw
```

avec :

n : répétiteur.

w : longueur de la zone en caractères.

Remarque : si on met la valeur de w la spécification s'adapte automatiquement à la longueur de la chaîne de caractères à éditer.

Exemple

```
      CHARACTER  x*4

      x='math'
      WRITE (*,10) x
      WRITE (*,11) x
      WRITE (*,12) x

10    FORMAT (a2,'!')
11    FORMAT (a5,'!')
12    FORMAT (a,'!')
```

Nous obtiendrons:

```
ma!

␣math!
```

```
mat

math!
```

V.4.2.2. Spécificateur X

Il permet d'insérer des blancs (ou d'espace).

Syntaxe

```
[n]X
```

n : facteur de répétition (entier positif ou nul).

Exemple

```
      CHARACTER x*4

      x='math'
      WRITE (*,10) x
      WRITE (*,11) x
      WRITE (*,12) x

10    FORMAT (2X,a,'!')
11    FORMAT (2X,a,2X'!')
12    FORMAT (2X,a,5X,'!')
```

Nous obtiendrons :

```
␣ ␣math!

␣ ␣math␣ ␣!

␣ ␣math␣ ␣ ␣ ␣ ␣!
```

V.4.2.4. Spécificateur / (slash)

Il permet le passage à l'enregistrement logique suivant.

On répète cette spécification tout en répétant les / .

// est équivalente à / , /

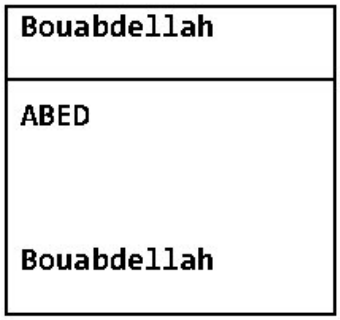
//// est équivalente à / , / , / , /

Exemple

```
      CHARACTER  x*4 , y*11

      x = 'ABED'
      y = 'Bouabdellah'
      WRITE (1,10) x , y
      WRITE (1,11) x , y
      WRITE (1,12) x , y
      WRITE (1,13) x , y
      WRITE (1,14) x , y

10    FORMAT (a)
11    FORMAT (2a)
12    FORMAT (a,1x,a)
13    FORMAT (a,/,a)
14    FORMAT (a,//,a)
```

L'éditeur donnera :

ABED **Bouabdellah**
ABEDBouabdellah
ABED Bouabdellah
ABED



V.4.2.3. Spécificateur chaine de caractère ' '

Édite la chaîne de caractère définie par la constante caractère ' '.

Exemple

```
      CHARACTER  x*4 , y*11

      x = 'ABED'
      y = 'Bouabdellah'
      WRITE (1,10) x
      WRITE (1,11) y

10    FORMAT ('Nom    : ' , a)
11    FORMAT ('Prénom : ' , a)
```

à l'édition, nous aurons :

```
Nom    : ABED

Prénom : Bouabdellah
```

Chapitre VI SOUS-PROGRAMMES

VI.1. Généralités

VI.1.1. But des sous-programmes

Il arrive très souvent qu'en cours d'élaboration d'un algorithme, on repère des séquences dont la fonction est identique et dont seules les données varient d'une utilisation à l'autre.

On peut dupliquer autant de fois que nécessaire les séquences en question et les adapter, mais il semble bien plus judicieux de créer une seule fois la séquence dans un module distinct dont on déclenchera l'exécution à chaque utilisation. *C'est le sous-programme (S/P).*

Il arrive aussi que les logiciels soient très importants et comportent un très grand nombre d'instructions, ce qui nuit considérablement à leur clarté.

Dans ce cas, on a recours aux **S/P**, non plus pour éviter les répétitions de séquences, mais pour fragmenter le programme en un ensemble de **S/P** plus courts et donc plus faciles à concevoir, lire et mettre au point.

Le **S/P** se présente donc comme un module distinct qui va, à partir de données, réaliser une séquence de traitement (inversion de matrice, intégration de f(x), ...) pour aboutir aux résultats.

FORTRAN 77 fait communiquer les **S/P** avec leurs environnements extérieurs au moyen d'une liste de paramètres qui leur sont associés.

Cette liste de paramètres constitue la "fenêtre" à travers laquelle le **S/P** reçoit ses données ou renvoie ses résultats. La description précise de cette liste est essentielle pour une bonne utilisation du **S/P**. Nous distinguerons les 4 classés de paramètres suivants :

1- **Paramètres d'entrée :** Ils contiennent les données du S/P et ne sont pas modifiés par celui-ci.

2- **Paramètres d'entrée-sortie :** Ils contiennent des données avant l'appel au **S/P** et des résultats après. Il faut faire attention à l'usage de ces paramètres car les données qu'ils contiennent sont détruites par le **S/P**.

3- **Paramètres de sortie :** Ils contiennent les résultats produits par le S/P.

4- **Paramètres de travail :** Ce sont des paramètres sans signification ni en entrée, ni en sortie, que le **S/P** utilise pour travailler.

Le but d'un **S/P** est donc, soit d'éviter des répétitions inutiles, soit de fragmenter les programmes pour les rendre 'mentalement accessibles' à celui qui les conçoit.

C'est un concept très puissant qui permet en une seule instruction de déclencher le déroulement des séquences prédéfinies et donc de créer un véritable 'macro-langage'.

VI.1.2. Différents types de Sous-Programmes en FORTRAN 77

FORTRAN 77 connaît deux types de **S/P** :

1 - **Les S/P internes :** Ils sont définis et utilisables uniquement dans un module (programme ou **S/P**).

2 - **Les S/P externes :** Ils sont définis, compilés et utilisables dans n'importe quel autre module (programme ou **S/P**). Ils peuvent éventuellement être rangés dans des bibliothèques de **S/P** (cas des fonctions standards : **SIN**, **COS**, …).

VI.2. Sous-Programmes internes

Ce sont des fonctions d'une ou plusieurs variables, dont la définition peut être donnée en une seule instruction **FORTRAN 77**, placée avant toute instruction exécutable.

Ils correspondent en fait à une affectation pour exprimer les formules dont l'emploi se répète.

Syntaxe :

FUNCTION_Name (<Liste de paramètres>) = <Expression>

FUNCTION_Name **:** chaîne de 6 caractères alphanumériques au plus, le 1er étant une lettre.

Liste de paramètre : *Var.1, Var.2, …*

Expression **:** Expression arithmétique de type entière, réelle, ou opération sur chaine de caractère,.... suivant le type de la fonction.

Le type de la fonction est défini, comme pour les variables soit par la convention implicite, soit par une déclaration.

Exemples :

```
C   fonction de type réel
      ER (A, B) = ABS ((A - B)/A)

C   fonction de type entier
      MD (I, J) = (I + J)*N**J

C   fonction de type réel
      DELTA (A, B, C) = B**2. - 4.*A*C

C   fonction de type complexe
      COMPLEX CER, A, B
      CER (A, B) = A*B + (0., 1.)

C   fonction de type caractère
      CHARACTER CAR*20, C*10
      CAR (C, I) = C(1 : I)//CHAR (I+1)
```

L'utilisation de ces fonctions se fait comme pour les fonctions standards en les nommant dans une expression de mode correspondant.

Exemple :

```
      X = - ALOG10(ER(X, Y)) + 7.
      INDICE = (MD(1, IB) + K)*(MD(I,j)
```

VI.3. Sous-Programmes externes

VI.3.1. Sous-Programmes fonctions

Ce sont des fonctions de **n** variables. La différence avec les **S/P** internes est qu'ils peuvent être utilisés par n'importe quel autre module et être définis par un nombre quelconque d'instructions.

Syntaxe

Un S/P fonction est la séquence comprise entre les mots clés **FUNCTION** et **END** .

[<Type>] FUNCTION *Name* **(** <*liste de paramètre*> **)**

[*Declaration statements*]

[*Executable statements*]

RETURN

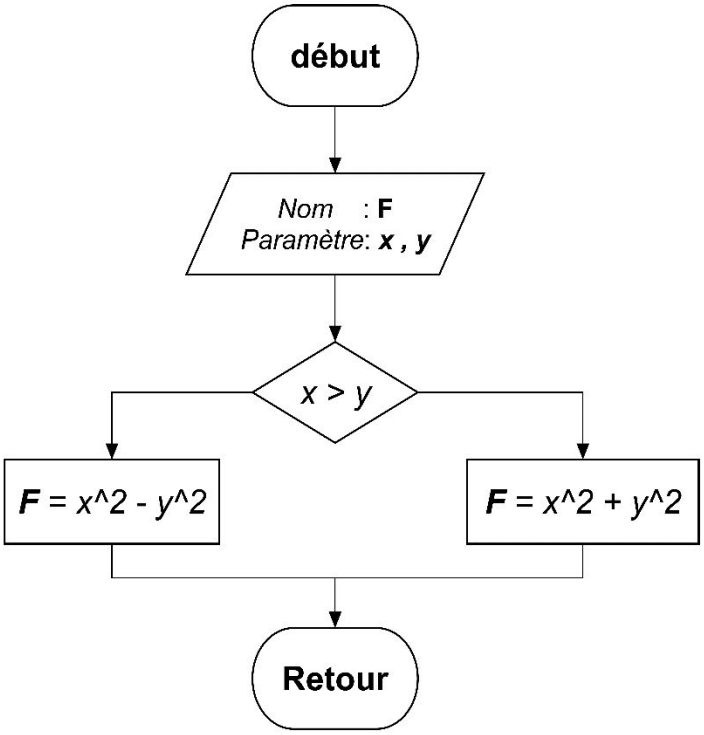
END

- La construction du < ***Name*** > et de la < ***liste de paramètre*** > obéit aux mêmes lois que pour les fonctions internes. Le mode de la fonction étant défini par la convention implicite ou une déclaration de mode.
- Le retour vers le module appelant se fait en utilisant l'instruction **RETURN**.

- Le retour de l'unique valeur de sortie se fait dans le nom de la fonction. Celui-ci doit donc être affecté au moins une fois dans le module (par exemple apparaître à gauche d'un signe égal).
- Si la fonction est de mode caractère, il faut préciser le nombre de caractères de cette fonction.

Exemple :

Soit l'organigramme suivant définissant une fonction ***f(x, y)***



On aura l'écriture **FORTRAN 77** suivante :

```
      FUNCTION F (X, Y)
C  Paramètres d'entrée :
C                       X : abscisse du point considéré (variable réelle)
C                       Y : ordonnée du point considéré (variable réelle)
C
C  Paramètre de sortie :
C                       F : valeur de la fonction
      IF ( X .GT. Y) THEN
        F = (X - Y)/SQRT(X**2 + Y**2)
       ELSE
        F = X*Y/ABS(X + Y)**1.5
      ENDIF
C
      RETURN
      END
```

L'utilisation de ce **S/P** se fait comme pour n'importe quelle autre fonction :

```
A = F(X1, X2)
X = F(X, Y)
U = F(X + 1., Y)"T + F(X, Y)^2
```

VI.3.2 Sous-Programmes généraux

Ce sont des applications de **n** variables d'entrée dans **p** variables de sortie. Comme les fonctions, ils peuvent être utilisés par n'importe quel autre module et être définis en un nombre quelconque d'instructions.

Un S/P général est la séquence comprise entre les mots clés **SUBROUTINE** , et **END** .

Syntaxe :

SUBROUTINE *Name* (<*liste de paramètre*>)

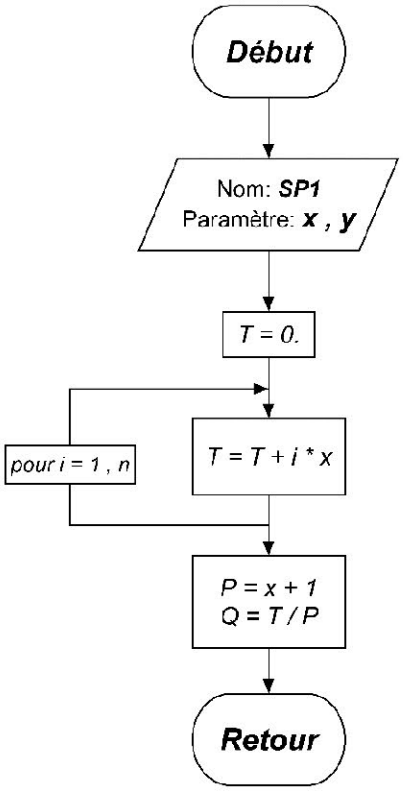
[*Declaration statements*]

[*Executable statements*]

```
      RETURN

      END
```

La construction du <nom symbolique> et de la <liste de paramètres> obéit aux mêmes lois que pour les fonctions.

Le retour vers le module appelant se fait en utilisant l'instruction **RETURN**.



On aura l'écriture **FORTRAN 77** suivante :

```
      SUBROUTINE SP1 (X, N, Q, P)
C       Ce S/P calcule P et Q  .....
C
C  Paramètres d'entrée :
C                       X : abscisse du point considéré (variable réelle)
C                       N : ordre du développement à calculer (variable entière)
C
C  Paramètre d'entrée-sortie :
C                       Q : en entrée : valeur de ......
C                           en sortie : résultat de ....
C
C  Paramètre de sortie :
C                       P : valeur de .....
      T = 0.0
      DO I = 1, N
        T = T + X * I
      END DO

      P = X + 1.
      Q = P / T

      RETURN
      END
```

L'utilisation de ce S/P se fait en utilisant le mot clé **CALL** :

```
CALL SP1 (X, NOMB, U, V)
CALL SP1 (A, N, P(1), P(2))
CALL SP1 (A + B, K, U, V)
```

Remarques :

Les paramètres d'entrée peuvent être des variables ou des expressions. Les paramètres de sortie ou d'entrée-sortie ne peuvent être que des variables.

Il doit y avoir une correspondance stricte entre la liste de paramètres formels apparaissant dans la définition du S/P et la liste de paramètres effectifs (ou actuels) apparaissant dans l'appel. Correspondance en nombre, en position dans la liste de paramètres et en mode.

Toutes les variables qui n'apparaissent pas dans la liste des paramètres sont dites locales (par exemple : I, T). Elles n'ont d'existence et ne sont définies que pendant l'appel du **S/P**. En particulier, elles ne conservent pas leur valeur d'un appel à l'autre.

VI.4. Transmission de variables indicées en paramètres

Très souvent, il est nécessaire devoir dans la liste de paramètres des variables indicées. Dans ce cas, il faut déclarer que ces variables sont indicées avec un " DIMENSION ...".

FORTRAN 77 offre deux possibilités pour cette déclaration :

- Le dimensionnement fixe
- Le dimensionnement variable

VI.4.1. Dimensionnement Fixe

Identique au dimensionnement d'un programme principal, les valeurs mini et maxi des indices sont des constantes. On a, par exemple, dans le cas d'un sous-programme calculant le maximum des modules des termes d'un vecteur :

```
      FUNCTION VNORM (VECT, N)
C
C  Paramètres d'entrée :
C                       VECT : vecteur dont on calcule la norme
C                       N    : ordre du vecteur à normer
C
C  Paramètre de sortie :
C                       VNORM : norme de VECT
      DIMENSION VECT (10)

      VNORM = ABS(VECT(1))

      DO I = 2, N
        VNORM = AMAX1( VNORM , ABS(VECT(I)) )
      END DO

      RETURN
      END
```

Cette écriture qui a le mérite d'être simple souffre cependant de nombreux inconvénients. On peut citer par ordre d'importance :

- Non généralité du sous-programme :

  Celui-ci est valable pour des vecteurs d'ordre 10 maximums.

- Réservation d'un vecteur supplémentaire de 10 éléments :

  En effet, le module qui appelle cette fonction a déjà réservé un vecteur de 10 éléments par sa déclaration "DIMENSION VECT(10)".

Pour ces deux raisons, on préfère utiliser le dimensionnement variable.

Cette écriture qui a le mérite d'être simple souffre cependant de nombreux inconvénients. On peut citer par ordre d'importance :

- Non généralité du sous-programme :

  Celui-ci est valable pour des vecteurs d'ordre 10 maximums.

- Réservation d'un vecteur supplémentaire de 10 éléments :

  En effet, le module qui appelle cette fonction a déjà réservé un vecteur de 10 éléments par sa déclaration "DIMENSION VECT(10)".

Pour ces deux raisons, on préfère utiliser le dimensionnement variable.

VI.4.2. Dimensionnement Variable

Il consiste à remplacer les constantes dans la déclaration par des variables. Ces variables doivent obligatoirement être présentes dans la liste de paramètres. La fonction écrite au paragraphe précédent sera donc transformée de la façon suivante :

```
      FUNCTION VNORM (VECT, N)
C
C  Paramètres d'entrée :
C                       VECT : vecteur dont on calcule la norme
C                       N    : ordre du vecteur à normer
C
C  Paramètre de sortie :
C                       VNORM : norme de VECT

      DIMENSION VECT (N)

      VNORM = ABS(VECT(1))

      DO I = 2, N
        VNORM = AMAX1( VNORM , ABS(VECT(I)) )
      END DO

      RETURN
      END
```

Cette nouvelle écriture présente les avantages suivants :

- Généralité du sous-programme écrit : Celui-ci est valable pour des vecteurs d'ordre ***N*** .

- Pas de réservation supplémentaire de vecteur

  Cette déclaration `DIMENSION VECT(N)` ne réserve en fait pas de vecteur. Elle sert simplement à autoriser l'emploi de VECT comme variable indicée. Sinon *VECT(I)* serait compris comme l'appel d'une fonction externe.

  Etant donné les avantages de cette écriture, on généralisera donc son emploi dans le traitement des variables indicées apparaissant dans la liste d'appel des sous-programmes.

VI.5. Transmission de paramètres en zone commune

FORTRAN 77 donne aussi la possibilité de transmettre des informations à un **S/P** sans les faire apparaître explicitement en paramètres.

Pour cela, il faut placer les variables correspondantes dans des zones appelées "zones communes".

Le placement de variables en zone commune se fait par une déclaration.

On peut mettre des variables de modes différents dans une zone commune, à l'exception des variables en mode caractère qui ne doivent pas être mélangées avec les autres.

Exemple :

```
CHARACTER*1 CAR , LIGNE*80 , BUFF*OO

COMMON /ZONE1/ A , B , 1
COMMON /ZONE2/ C , J , K
COMMON /CAR1/  CAR , LIGNE , BUFF
```

Les variables ***A, B, I*** sont placées dans la zone commune appelée ***ZONE1*** et les variables ***C, J, K*** dans la zone commune appelée ***ZONE2***.

Les variables ***CAR, LIGNE, BUFF***, toutes de mode caractère sont regroupées dans la zone commune appelée ***CAR1***.

La transmission des variables placées en zone commune dans un **S/P** se fait en mettant la déclaration correspondante dans celui-ci.

Exemple

```
      FONCTION VNORM (N)
C
C  Paramètres d'entrée :
C                       VECT   : matrice dont on calcule la norme
C                       N      : ordre de la matrice à normer
C
C  Paramètre de sortie :
C                       VNORM  : norme de VECT

      DIMENSION VECT (10)

      COMMON /ZONE1/ VECT

      ANORM = ABS (VECT(1))
      DO I = 2, N
        ANORM = AMAX1( ANORM , ABS(VECT(I)) )
      END DO

      RETURN
      END
```

L'emploi de cette fonction nécessite le placement de VECT dans une zone commune de nom ZONE **1**.

Exemple

```
      PROGRAM ESSAI
C
      DIMENSION VECT (10)
      COMMON /ZONE1/ VECT
C
      N = 7
      X = VNORM (N)
```

Cette possibilité a l'avantage de diminuer (voire supprimer) la liste de paramètres dans le cas ou celle-ci est très longue.

Elle a par contre de nombreux inconvénients :

- Perte de généralité du sous-programme :
  - Due au fait que l'on ne peut plus utiliser le dimensionnement variable.
  - Due au fait qu'on impose à l'utilisateur des zones communes figées.
  - Impossibilité d'utiliser le **S/P** avec des variables de noms différents.
- Perte de lisibilité :
  - Les informations modifiées n'apparaissent plus explicitement en paramètres, ce qui peut conduire à des difficultés de compréhension.

Pour ces raisons, la transmission de variables en zone commune est peu utilisée.

VI.6. Transmission d'un nom de sous-programme en paramètre

FORTRAN 77 donne la possibilité de transmettre le nom d'un sous-programme

comme paramètre d'un autre sous-programme.

Cette possibilité est très intéressante, car elle permet un paramétrage plus complet d'un sous-programme.

Par exemple, si l'on désire écrire un sous-programme qui calcule une racine d'une fonction ($f(x) = 0$), on pourra passer le nom de la fonction en paramètre et ainsi écrire un sous-programme général (valable pour toute fonction).

L'écriture d'un tel sous-programme se fait en suivant les règles habituelles, mais son utilisation nécessite une déclaration pour indiquer que le paramètre effectif donnant le nom de la fonction n'est pas le symbole d'une variable ordinaire, mais représente un nom de fonction (ou de sous-programme).

Cette déclaration se fait dans le module appelant le sous-programme.

Deux cas peuvent se produire, suivant que le symbole à déclarer :

- est celui d'un sous-programme ou d'une fonction externe : **EXTERNAL**

- est celui d'une fonction standard : **INTRINSIC**

Exemple

```
      SUBROUTINE ZERO (F, A, B, MAXIT, X, IER)
C Paramètres d'entrée :
C         F     : Nom de la fonction dont on cherche une racine
C         A     : Borne mini de l'intervalle contenant la racine
C         B     : Borne maxi de l'intervalle contenant la racine
C         MAXIT : Nombre maximum d'itérations autorisé
C
C Paramètres de sortie :
C         X       : Valeur de la racine trouvée
C         IER     : Indicateur de sortie :
C                     IER = 0 racine trouvée
C                     IER = 1 racine non trouvée (trop d'itérations)

      XA = A
      XB = B
       .
       .
      T = (F(A) - F(B))/2.
       .
       .
      RETURN
      END
```

A l'utilisation de ce sous-programme il faudra déclarer :

```
EXTERNAL INVOL, FONC

CALL  ZERO (INVOL , X1 , X2 , MAX , SOLU , IER)

CALL  ZERO (FONC , X , Y , MAX , SOLU , IER)
```

Ou bien, si on veut faire passer des noms de fonctions standards

```
INTRINSIC SIN, TAN

CALL  ZERO (SIN , -PI , PI , MAXIT , RACINE , 1ER)

CALL  ZERO (TAN , X , Y , MAXIT , RACINE , 1ER)
```

ANNEXES

Annexe A - Instructions

ASSIGN : Attribue la valeur d'une étiquette à un entier

BACKSPACE : Positionne le pointeur de fichier sur l'enregistrement précédent

BLOCK DATA : Identifie un sous-programme bloc de données dans lequel on peut initialiser variables et tableaux

BLOCK DATA : initialisation de variables placées en zone COMMON.

CALL : Appelle et éxécute un sous-programme

CHARACTER : Déclaration pour variables alphanumériques

CLOSE : Fermeture de fichier

COMMON : Variables globales, partage par plusieurs modules

COMPLEX : Déclaration pour variables complexes

CONTINUE : Instruction désuète, sans effet.

DATA : Initialisation de variables

DIMENSION : Déclaration pour tableaux

DO : Boucle Pour

DO WHILE : Boucle **Tant Que** (F90)

DOUBLE : Déclaration double précision

ELSE : Sinon de la structure SI... ALORS... SINON

END DO : Fin de boucle pour ou tant que (F90)

END : Fin de module (programme, sous-programme,..)

END IF : Fin construction alternée SI...

EXIT : Sortie prématurée d'une boucle DO

EXTERNAL ; Identifie un nom comme étant un sous-programme ou une fonction

FORMAT : Format de lecture ou écriture

FUNCTION : Sous-programme de type fonction

GOTO : Saut

IF : Structure alternative SI

IMPLICIT : Attribue un type implicite à certaines variables

INQUIRE : Examen propriétés de fichiers

INTEGER : Déclaration pour variables entières

INTEGER : Déclaration pour variables entières

INTRINSIC : Déclaration pour fonctions intrinsèques

LOGICAL : Déclaration pour variables logiques

LOGICAL : Déclaration pour variables logiques

OPEN : Ouverture de fichier

PARAMETER : Donne un nom à une constante

PAUSE : Arrêt temporaire du programme

PRINT : Sortie écran

PROGRAM : Début de programme

READ : Lecture

REAL : Déclaration de type réel

RETURN : Retour de sous-programme ou fonction

RETURN : Retour de sous-programme ou fonction

REWIND : Pointe sur le début du fichier

SAVE : Maintien des variables dans des sous-programmes

STOP : Arrêt programme

SUBROUTINE : Nom de Sous-Programme

WRITE : Ecriture

Annexe B - Fonctions intrinsèques

Fortran 77 contient un ensemble de procédures prédéfinies permettant de réaliser les calculs les plus usuels (qu'on appelle Fonctions intrinsèques).

FONCTION	DESCRIPTION	TYPE DES ARGUMENTS	TYPE DE LA VALEUR DE RETOUR
EXP(X)	exponentielle	réel réel double précision complexe	réel réel double précision complexe
LOG(X)	log. népérien	réel réel double précision complexe	réel réel double précision complexe
LOG10(X)	log. en base 10	réel réel double précision complexe	réel réel double précision complexe
SIN(X)	sinus	réel réel double précision complexe	réel réel double précision complexe
COS(X)	cosinus	réel réel double précision complexe	réel réel double précision complexe
TAN(X)	tangente	réel réel double précision complexe	réel réel double précision complexe
ASIN(X)	sinus inverse	réel réel double précision complexe	réel réel double précision complexe
ACOS(X)	cosinus inverse	réel réel double précision complexe	réel réel double précision complexe
ATAN(X)	tangente inverse	réel réel double précision complexe	réel réel double précision complexe

COSH(X)	cosinus hyperbolique	réel réel double précision complexe	réel réel double précision complexe
SINH(X)	sinus hyperbolique	réel réel double précision complexe	réel réel double précision complexe
TANH(X)	tangente hyperbolique	réel réel double précision complexe	réel réel double précision complexe
ABS(X)	valeur absolue pour x réel et module pour x complexe	réel réel double précision complexe	réel réel double précision réel double précision
MAX(X1,..., XN)	maximum	entier réel réel double précision	entier réel réel double précision
MIN(X1,..., XN)	minimum	entier réel réel double précision	entier réel réel double précision
INT(X)	troncature	réel réel double précision	entier
FLOAT(X) **OU** **REAL(X)**	conversion entier en réel simple précision	entier	réel
DBLE(X)	conversion entier en réel double précision	entier réel	réel double précision
NINT(X)	arrondi	Réel réel double précision	entier
MOD(Y, X)	reste de la division de y par x	entier, entier	entier
CONJG(Z)	conjugué	réel double précision	réel double précision
SIGN(X1, X2)	renvoie le signe de x1*x2	entier réel réel double précision	entier réel réel double précision
CHAR(I)	renvoie le caractère ayant pour code ASCII i	entier	caractère
ICHAR(C)	retourne le code ASCII du caractère c	caractère	entier
LEN(C)	renvoie la longueur de c	chaîne de caractères	entier

Références

Livres

- Dubois, P. (1984). *Initiation au Fortran par l'exemple*. Éditions Technip.
- Etter, D. M., Etter, D. M., & Etter, D. M. (1993). *Structured FORTRAN 77 for engineers and scientists*. Benjamin/Cummings.
- Fuller, W. R. (1977). *FORTRAN programming: a supplement for calculus courses*. 1977 by *Springer-Verlag, New York Inc*.
- Monro, D. M. (1982). *Fortran 77*. London: Edward Arnold.
- Kupferschmid, M. (2009). *Classical Fortran: programming for engineering and scientific applications*. CRC Press.
- Lignelet, P. (1985). *La pratique du Fortran 77* . Masson.

Documentation générale disponibles sur le Web

https://web.stanford.edu › class › tutorial_77

https://www.idris.fr › formations › fortran › fortran-77

https://en.wikibooks.org › wiki › Fortran_77_Tutorial

https://www.star.le.ac.uk › ~cgp › prof77

http://www.strath.ac.uk/CC/Courses/fortran.html

http://www.softndesign.org/manuels/fortran.html