

**Université Pierre et Marie Curie
Paris VI**

**Master de Sciences et Technologie :
– Mention Physique et Applications
– Mention Sciences de l'Univers,
Environnement, Écologie**

**Méthodes numériques et informatiques :
programmation**

**Introduction
au fortran
90/95/2003**

Avertissement

Ce document est diffusé dans une version en évolution, qui reste encore très inachevée, notamment concernant les apports du fortran 2003. Plusieurs chapitres ne sont que des ébauches, en particulier ceux sur les structures, les pointeurs et l'interopérabilité avec le C ; d'autres questions sont à peine abordées. L'objectif reste de compléter progressivement le document.

Dans ce contexte, je vous remercie par avance de me signaler les erreurs aussi bien que les formulations obscures. Vos commentaires et suggestions seront les bienvenus.

Jacques.Lefrere@upmc.fr

30 octobre 2012

Remerciements

Je tiens à remercier mes collègues enseignants de la Maîtrise de Physique et Applications de Paris VI, Marie-Claude et Jean-Claude JUSTICE, Michel KARATCHENZEFF, Maï PHAM, ainsi que mes collègues du M1 du Master de l'UPMC, Albert HERTZOG, Philippe SINDZINGRE et François RAVETTA, pour leurs encouragements et leurs conseils au cours de la rédaction de ce document.

Je remercie aussi Marie-Alice FOUJOLS de l'Institut Pierre Simon Laplace pour ses séminaires enthousiastes qui nous ont sensibilisés au fortran 90.

Merci aussi aux collègues du Service d'aéronomie pour les nombreuses discussions au sujet du fortran au sein du laboratoire, et en particulier à Françoise PINSARD, Francis DALAUDIER et Stéphane MARCHAND pour leurs patientes relectures critiques.

Je remercie aussi Frédéric BERNARDO sans lequel le chapitre sur l'inter-opérabilité avec le langage C n'aurait pas vu le jour.

Merci enfin à tous ceux qui ont contribué de près ou de loin à l'élaboration de ce document, et surtout aux étudiants de la Maîtrise de Physique et Applications de Paris VI, puis à ceux du M1 du Master (mentions P&A et SDUEE), car c'est avant tout pour eux mais aussi grâce à eux qu'il a été rédigé.

Avertissement

Ce document n'est pas un manuel de référence du fortran 90/95/2003 décrivant exhaustivement la norme du langage. Il s'agit d'un document largement inspiré des références citées dans la bibliographie, références auxquelles il est vivement conseillé de se reporter pour une étude plus approfondie ou plus systématique. Incomplet, notamment sur certains aspects les plus récents du fortran, il s'apparente plus à un guide de l'utilisateur du fortran pour le calcul scientifique. Il s'adresse essentiellement à des étudiants ayant déjà une expérience de la programmation dans un langage structuré (fortran 77 ou C) souhaitant s'initier aux nouvelles fonctionnalités du langage à partir du fortran 90, notamment :

- la notation tableaux, les opérateurs et fonctions manipulant globalement des tableaux multidimensionnels,
- l'allocation dynamique, les structures et les pointeurs,
- le contrôle des passages d'arguments entre procédures via les modules.

Notations

- La police « machine à écrire », à espacement fixe, est utilisée pour indiquer les éléments du code source du fortran.
- Les crochets [...] délimitent les éléments optionnels de la syntaxe ou les arguments optionnels des procédures ; ces symboles ne font pas partie de la syntaxe.
- Lors de la description de la syntaxe, les symboles < et > indiquent où l'utilisateur doit substituer les identificateurs, expressions, ou valeurs qu'il a choisis ; ces symboles ne font pas partie de la syntaxe.

Exemple

La syntaxe décrite comme suit :

```
[<nom> :] IF (<expression logique>) THEN
  <bloc d'instructions>
END IF [<nom>]
```

peut être utilisée comme dans l'exemple suivant, où l'identificateur `chiffre` est optionnel :

```
chiffre: IF (i < 10) THEN
  WRITE(*,*) 'i est inférieur à 10'
END IF chiffre
```

Présentation

Après un chapitre 1 introductif (p. 1) présentant l'évolution du fortran, les étapes de mise au point des programmes et les éléments du langage, le chapitre 2 (p. 8) décrit les types d'objets du langage et leurs attributs, avant d'aborder au chapitre 3 (p. 18) les opérateurs qui permettent de les manipuler.

Le chapitre 4 (p. 22) est entièrement consacré aux structures de contrôle du fortran : c'est un chapitre essentiel et facile qui doit être étudié dès la première lecture.

Le chapitre 5 entrées-sorties (p. 30) aborde les instructions de lecture et d'écriture et les formats de conversion des données : il est conseillé de commencer par les sections d'introduction et de terminologie, quitte à revenir ensuite sur la description détaillée des instructions et surtout des formats, pour laquelle des allers et retours avec les sections d'exemples ainsi qu'avec le chapitre 8 (p. 84) sur les chaînes de caractères sont nécessaires.

Le chapitre 6 intitulé procédures (p. 51) constitue le deuxième thème majeur du document : il présente les fonctions et les sous-programmes en partant de la notion de procédure interne attachée à un programme pour aboutir aux procédures de module réutilisables grâce à la compilation séparée. La dernière section (6.5) abordant les fonctionnalités avancées peut être passée en première lecture.

Le troisième thème majeur du fortran, les tableaux, fait l'objet du chapitre 7 (p. 70) : il constitue l'atout principal du fortran 90 pour le calcul scientifique avec l'introduction des notations matricielles et des fonctions intrinsèques manipulant les tableaux multidimensionnels. Ce chapitre essentiel peut être abordé en deux étapes : la première lecture doit inclure l'allocation dynamique ainsi que le lien entre procédures et tableaux, mais on peut garder pour une deuxième lecture les notions de sections non-régulières, les masques et la structure FORALL.

Le chapitre 8 (p. 84) présente la manipulation des chaînes de caractères et les fonctions associées : c'est un chapitre facile, même s'il n'est pas central pour les applications au calcul scientifique.

Les chapitres 9 sur les types dérivés (p. 92), 10 sur la généricité (p. 97) et 11 sur les pointeurs (p. 104) présentent des fonctionnalités orientées objet du fortran, et peuvent être réservés à une deuxième lecture. Quant au chapitre 12 (p. 115), qui présente les techniques d'appel de fonctions écrites en C depuis le fortran et réciproquement, il peut intéresser des non-spécialistes du fortran pour l'interfacer avec le langage C.

Enfin, les annexes ont plus vocation à servir de référence qu'à être lues de façon linéaire, même s'il est utile de les parcourir pour connaître leur existence. Cela vaut par exemple pour la liste des fonctions intrinsèques du fortran (A, p. 125) quitte à revenir au document ponctuellement pour vérifier une syntaxe d'appel. De la même façon, parcourir l'annexe C, p. 140 sur la norme IEEE de représentation des nombres réels permet de visualiser concrètement les approximations des calculs en machine. Par ailleurs, les utilisateurs du langage C pourront trouver dans l'annexe D, p. 145, un aide-mémoire présentant une comparaison approximative des syntaxes et fonctions des deux langages. Pour terminer, l'annexe E, p. 149 fournit de brèves indications sur l'utilisation de quelques compilateurs.

Indications de lecture

♠ 0.0.0 Sous-section facultative

Les sections (ou sous-sections) qui peuvent être omises en première lecture sont indiquées par le symbole ♠ placé en marge comme ci-dessus.

f95/2003 0.0.0 Sous-section nouveauté f95/f2003

Les sections (ou sous-sections) qui font intervenir des nouveautés des standards 95 ou 2003 du fortran sont indiquées par le symbole f95/2003 placé en marge comme ci-dessus.

Conseils pratiques

♡ ⇒ Les règles de « bon usage » du langage, qui, au delà de la norme, sont motivées par des objectifs de lisibilité, de portabilité ou de robustesse du code source, sont repérées par le symbole ♡ en marge du texte, comme pour ce paragraphe.

Table des matières

Avertissement	iii
Notations	iii
Présentation	iv
Indications de lecture	iv
Table des matières	v
1 Introduction	1
1.1 Historique du fortran	1
1.2 Les étapes de mise en œuvre d'un programme	2
1.2.1 Édition du fichier source	2
1.2.2 Compilation et lien	3
1.2.3 Exécution	4
1.2.4 Cas des fichiers sources multiples	4
1.3 Les éléments du langage	5
1.3.1 Les caractères du fortran	5
1.3.2 Les identificateurs des objets	5
1.4 Les formats du fortran	6
1.4.1 Format fixe	6
1.4.2 Format libre du fortran 90	7
2 Types, constantes et variables	8
2.1 Les types intrinsèques	8
2.2 La représentation des types numériques	8
2.2.1 Représentation des entiers	8
2.2.2 Représentation des réels en virgule flottante	10
2.2.3 Domaine et précision des types numériques	11
2.3 Déclaration	13
2.4 Les constantes	14
2.5 Initialisation	14
2.5.1 Les constantes nommées (ou symboliques),	15
2.5.2 Variantes des types prédéfinis	15
2.5.3 Exemples de constantes nommées spécifiant les variantes des types prédéfinis	16
2.5.4 Une méthode modulaire pour fixer la précision des réels	16
2.6 Fonctions de conversion de type	17
3 Opérateurs et expressions	18
3.1 Les opérateurs numériques	18
3.1.1 Règles de priorité entre les opérateurs numériques binaires	18
3.1.2 Imperfections numériques des opérations liées aux types	18
3.1.3 Conversions implicites	19
3.2 Les opérateurs de comparaison	20
3.3 Les opérateurs booléens	21
3.4 Opérateur de concaténation des chaînes de caractères	21

4	Structures de contrôle	22
4.1	Structures IF	22
4.1.1	L'instruction IF	22
4.1.2	La structure IF ... END IF	22
4.1.3	Utilisation du ELSE IF	23
4.2	Structure SELECT CASE	23
4.3	Structures de boucles	24
4.3.1	Boucles DO avec compteur	25
4.3.2	Boucles DO WHILE	25
4.3.3	Ruptures de séquence dans les boucles : EXIT et CYCLE	25
4.3.4	Boucles DO sans compteur	26
4.4	Autres instructions de contrôle	27
4.4.1	Instruction CONTINUE	27
4.4.2	Branchement par GO TO	27
4.4.3	Instruction STOP	27
4.4.4	Instruction RETURN	28
4.4.5	Remarques sur les instructions STOP et RETURN	28
4.4.6	Branchements dans les entrées-sorties	28
5	Entrées-sorties, fichiers	30
5.1	Introduction : entrées et sorties standard	30
5.1.1	Une instruction d'E/S = un enregistrement	30
5.1.2	Saisie de données au clavier	31
5.2	Généralités et terminologie	32
5.2.1	Fichiers externes et fichiers internes	32
5.2.2	Notion d'enregistrement	33
5.2.3	Fichiers formatés et fichiers non formatés	33
5.2.4	Méthodes d'accès aux données	34
5.2.5	Notion de liste d'entrée-sortie	35
5.3	Instructions d'entrées-sorties	36
5.3.1	Le module intrinsèque ISO_FORTRAN_ENV	36
5.3.2	OPEN	37
5.3.3	CLOSE	39
5.3.4	READ	39
5.3.5	WRITE	39
5.3.6	PRINT	40
5.3.7	INQUIRE	40
5.3.8	Instructions de positionnement dans les fichiers	41
5.3.9	Entrées-sorties sans avancement automatique	41
5.4	Descripteurs de format	42
5.4.1	Descripteurs actifs	42
5.4.2	Descripteurs de contrôle	43
5.4.3	Syntaxe des formats et règles d'exploration	44
5.4.4	Boucles et changements d'enregistrement	45
5.4.5	Format variable	45
5.5	Exemples d'entrées-sorties formatées	46
5.5.1	Descripteurs de données numériques en écriture	46
5.5.2	Descripteurs de contrôle en écriture	47
5.6	Exemples de fichiers en accès direct	48
5.6.1	Exemple de fichier formaté en accès direct	48
5.6.2	Exemple de fichier non-formaté en accès direct	49

6	Procédures	51
6.1	Introduction	51
6.1.1	Intérêt des procédures	51
6.1.2	Variables locales, automatiques et statiques	52
6.1.3	Arguments des procédures	52
6.1.4	L'attribut <code>INTENT</code> des arguments	52
6.2	Sous-programmes et fonctions	53
6.2.1	Sous-programmes	54
6.2.2	Fonctions	54
6.3	Procédures internes et procédures externes	55
6.3.1	Procédures internes : <code>CONTAINS</code>	55
6.3.2	Procédures externes	56
6.3.3	La notion d'interface	57
6.4	Les modules	58
6.4.1	Exemple de procédure de module	58
6.4.2	Partage de données via des modules	60
6.4.3	Éléments d'encapsulation	61
6.4.4	L'instruction <code>IMPORT</code> dans les interfaces	62
6.4.5	Modules intrinsèques	63
6.5	Fonctionnalités avancées	63
6.5.1	Arguments optionnels	63
6.5.2	Transmission d'arguments par mot-clef	64
6.5.3	Noms de procédures passés en argument	64
6.5.4	Procédures récursives	67
6.5.5	Procédures pures	69
6.5.6	Procédures élémentaires	69
7	Tableaux	70
7.1	Généralités	70
7.1.1	Terminologie des tableaux	70
7.1.2	Ordre des éléments dans un tableau multidimensionnel	71
7.1.3	Constructeurs de tableaux	72
7.2	Sections de tableaux	72
7.2.1	Sections régulières	72
7.2.2	Sections non-régulières	73
7.3	Opérations sur les tableaux	73
7.3.1	Extension des opérations élémentaires aux tableaux	73
7.3.2	Instruction <code>WHERE</code>	75
7.3.3	Affectation d'une section non-régulière	75
7.4	Fonctions intrinsèques particulières aux tableaux	75
7.4.1	Fonctions d'interrogation	75
7.4.2	Fonctions de réduction	76
7.4.3	Fonctions de transformation	77
7.4.4	Notion de masque	78
7.4.5	<code>ALL</code> , <code>ANY</code> , <code>COUNT</code>	78
7.4.6	Instruction et structure <code>FORALL</code>	79
7.5	Tableaux et allocation dynamique	79
7.5.1	Tableaux et procédures	79
7.5.2	Tableaux dynamiques	81
7.5.3	Allocation au vol de tableaux dynamiques par affectation	83

8	Chaînes de caractères	84
8.1	Le type chaîne de caractères	84
8.1.1	Les constantes chaînes de caractères	84
8.1.2	Les déclarations de chaînes de caractères	84
8.1.3	Les variantes du type chaînes de caractères	85
8.2	Expressions de type chaîne de caractères	85
8.2.1	Concaténation de chaînes	85
8.2.2	Sous-chaînes	85
8.2.3	Affectation	86
8.3	Les fonctions opérant sur les chaînes de caractères	86
8.3.1	Suppression des espaces terminaux avec TRIM	86
8.3.2	Justification à gauche avec ADJUSTL	86
8.3.3	Justification à droite avec ADJUSTR	87
8.3.4	Les fonctions LEN et LEN_TRIM	87
8.3.5	Recherche de sous-chaîne avec INDEX	87
8.3.6	Recherche des caractères d'un ensemble avec SCAN	87
8.3.7	Recherche des caractères hors d'un ensemble avec VERIFY	87
8.3.8	Duplication de chaînes avec REPEAT	88
8.3.9	Conversion de caractère en entier	88
8.3.10	Conversion d'entier en caractère	88
8.3.11	Comparaison de chaînes de caractères	88
8.3.12	Le caractère de fin de ligne NEW_LINE	88
8.4	Les entrées-sorties de chaînes de caractères	88
8.4.1	Les entrées-sorties de chaînes formatées	88
8.4.2	Les entrées de chaînes en format libre	89
8.4.3	Les fichiers internes : codage en chaîne de caractères et décodage	89
8.5	Les tableaux de chaînes de caractères	89
8.6	Chaînes et procédures	90
9	Types dérivés ou structures	92
9.1	Définition d'un type dérivé	92
9.1.1	L'attribut SEQUENCE	92
9.2	Référence et affectation des structures	93
9.2.1	Constructeur de type dérivé	93
9.2.2	Sélecteur de champ	93
9.3	Imbrication et extension de structures	94
9.3.1	Imbrication de structures	94
9.3.2	Extension d'un type dérivé	94
9.4	Structures et tableaux	94
9.5	Types dérivés à composantes allouables dynamiquement	95
9.6	Procédures et opérateurs agissant sur les structures	95
10	Généricité, surcharge d'opérateurs	97
10.1	Procédures génériques et spécifiques	97
10.2	L'interface-opérateur	98
10.2.1	Surcharge d'opérateurs	99
10.2.2	Création d'opérateurs	99
10.3	L'interface-affectation	100
11	Pointeurs	104
11.1	Pointeurs, cibles et association	104
11.1.1	Cible nommée	105
11.1.2	Cible dynamique anonyme	106
11.2	Pointeurs sur des tableaux	107
11.3	Tableaux de types dérivés contenant des pointeurs	109
11.4	Manipulation de listes chaînées	112

12 Inter-opérabilité avec le C	115
12.1 Inter-opérabilité des types intrinsèques	115
12.2 Inter-opérabilité des types dérivés	116
12.3 Inter-opérabilité avec les pointeurs du C	116
12.4 Inter-opérabilité des procédures	116
12.4.1 Le passage par copie de valeur (value)	117
12.4.2 Exemple : appel de fonctions C depuis le fortran	117
12.4.3 Exemple : appel de sous-programmes fortran depuis le C	118
12.5 Inter-opérabilité des tableaux	119
12.5.1 Exemple : fonctions C manipulant des tableaux définis en fortran	119
12.5.2 Exemple : fortran manipulant des tableaux dynamiques alloués en C	121
A Procédures intrinsèques	125
A.1 Fonctions numériques de base	125
A.2 Conversion et troncature	127
A.3 Générateur pseudo-aléatoire	127
A.4 Représentation des nombres	129
A.5 Fonctions opérant sur des tableaux	130
A.6 Manipulation de bits	131
A.7 Pointeurs	131
A.8 Accès à l'environnement système	131
A.9 Gestion du temps	132
A.10 Caractères et chaînes de caractères	134
B Ordre des instructions	139
C La norme IEEE 754	140
C.1 Introduction	140
C.1.1 Représentation des réels en virgules flottante	140
C.1.2 Arithmétique étendue	141
C.2 Les codes normaux	141
C.2.1 Les nombres normalisés	141
C.2.2 Les nombres dénormalisés	141
C.3 Les codes spéciaux	142
C.3.1 Arithmétique IEEE et options de compilation	142
C.4 Le codage IEEE des flottants sur 32 bits et 64 bits	143
C.4.1 Le codage IEEE des flottants sur 32 bits	143
C.4.2 Le codage IEEE des flottants sur 64 bits	144
D Correspondance entre les syntaxes du fortran et du C	145
D.1 Déclarations des types de base	145
D.2 Opérateurs algébriques	145
D.3 Opérateurs de comparaison	145
D.4 Opérateurs logiques	146
D.5 Opérations sur les bits	146
D.6 Structures de contrôle	146
D.7 Pointeurs	147
D.8 Fonctions mathématiques	147
D.9 Formats	148
D.10 Codage des valeurs numériques	148
D.10.1 Codage des entiers	148
D.10.2 Codage des flottants	148

E	Compilateurs et options de compilation	149
E.1	Compilateur <code>xlf</code> sous IBM-aix	149
E.1.1	Options du compilateur <code>xlf</code> conseillées	149
E.2	Compilateur <code>f95</code> ou <code>nagfor</code> de NAG	150
E.2.1	Options du compilateur <code>nagfor</code> conseillées	150
E.3	Compilateur <code>pgf95</code> de PORTLAND	151
E.3.1	Options du compilateur <code>pgf95</code> conseillées	151
E.4	Compilateur <code>ifort</code> d'INTEL	152
E.4.1	Options du compilateur <code>ifort</code> conseillées	152
E.5	Compilateur <code>g95</code>	152
E.5.1	Options du compilateur <code>g95</code> conseillées	153
E.6	Compilateur <code>gfortran</code>	154
E.6.1	Options du compilateur <code>gfortran</code> conseillées	154
E.7	Options contrôlant le nombre d'octets des types numériques	155
E.7.1	Options passant sur 64 bits les entiers par défaut	155
E.7.2	Options passant sur 64 bits les réels par défaut	155
	Bibliographie	156
	Index	161

Chapitre 1

Introduction

1.1 Historique du fortran

Le fortran (**FORMULA TRANSLATION**) est le premier langage informatique de haut-niveau. Né à la fin des années 1950 sous l'impulsion de John BACKUS¹, il a été standardisé en 1972 sous la forme du FORTRAN 66 et son efficacité dans le calcul scientifique en a fait le langage le plus utilisé dans les applications non commerciales. La mise à jour du standard à la fin des années 1970 a apporté d'énormes améliorations en particulier dans le traitement des chaînes de caractères avec le FORTRAN 77.

Mais c'est avec FORTRAN 90, dont la norme fut particulièrement longue à négocier, qu'est intervenue une véritable modernisation du langage fortran. Cette nouvelle version a permis un nettoyage des éléments les plus obsolètes du fortran (format fixe par exemple, lié à l'utilisation des cartes perforées). Elle a aussi introduit des fonctionnalités nouvelles parfois présentes dans des langages plus récents, parmi lesquelles nous ne soulignerons que les plus attendues dans notre domaine d'applications :

- langage de programmation structuré ;
- outils de manipulation des tableaux (multidimensionnels) puissants, concis et adaptés au calcul vectoriel et parallèle ;
- gestion dynamique, pointeurs ;
- création de types dérivés (structures), surcharge d'opérateurs, généricité, ...
- fiabilisation des passages d'arguments entre procédures.

Enfin, l'évolution du langage fortran a continué avec le FORTRAN 95, qui constitue une révision mineure, mais surtout le FORTRAN 2003, dont le standard a été publié en novembre 2004². Il apporte notamment :

- une interopérabilité normalisée avec le langage C ;
- de nouvelles possibilités concernant les tableaux dynamiques et les types dérivés ;
- des fonctionnalités de programmation orientée objet, notamment la notion de procédure attachée (BOUND) à un type ;
- une meilleure intégration dans le système d'exploitation.

Une partie de ces fonctionnalités étaient déjà intégrées dans certains compilateurs FORTRAN 95 sous forme d'extensions, mais la publication de la norme va accélérer leur intégration³ et leur portabilité.

Dans ce contexte, nous ne documenterons pas en général les aspects qualifiés d'obsolescents en FORTRAN 90/95, donc destinés à disparaître des prochaines versions.

La nouvelle norme FORTRAN 2008 constitue une évolution mineure par rapport au fortran 2003 avec notamment la notion de sous-module, des outils de programmation parallèle (en particulier les tableaux distribués, *coarrays*⁴), de nouvelles fonctions mathématiques intrinsèques... Le document

1. John BACKUS, mort en 2007, a reçu le prix TURING en 1977, notamment pour ses travaux sur Fortran.

2. Consulter <http://j3-fortran.org/doc/>.

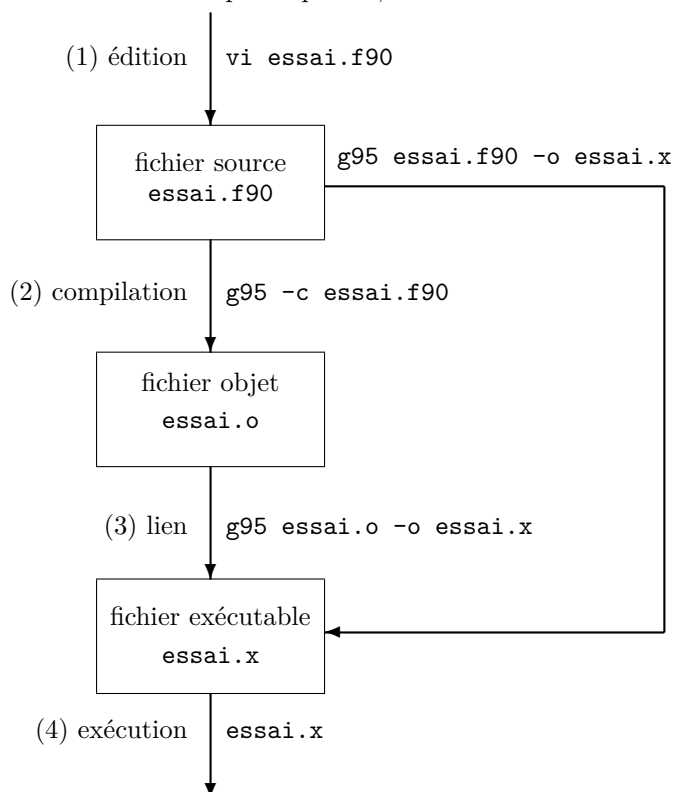
3. Voir la disponibilité sur les différents compilateurs : <http://fortranwiki.org/fortran/show/Fortran+2003+status>

4. Les *coarrays* sont déjà implémentés sous *g95*.

final du standard fortran 2008 (METCALF *et al.* (2011)) a été approuvé⁵ en septembre 2010.

1.2 Les étapes de mise en œuvre d'un programme

Outre l'analyse de l'algorithme, la mise en œuvre d'un programme en fortran comporte essentiellement quatre phases, schématisées ci-contre dans le cas d'un seul fichier source⁶ :



1. rédaction des unités de programme source à l'aide d'un éditeur de texte
2. compilation (proprement dite) des fichiers source pour produire des fichiers objets
3. lien entre les objets et les bibliothèques produisant un exécutable (grâce à l'éditeur de liens `ld`, lui-même appelé par le compilateur)
4. exécution

Par défaut, les phases de compilation (2) et de lien (3) sont initiées par un seul appel au compilateur et le fichier objet n'est pas conservé. Il est nécessaire de passer l'option `-c` au compilateur pour ne pas enchaîner automatiquement l'édition de lien.

Sans cette option, le compilateur détermine la nature du fichier (source ou objet) qui lui est fourni au vu du suffixe de son nom et lance la compilation des fichiers sources éventuels (suffixe `.f90`) puis l'édition de lien des objets (suffixe `.o`).

Les erreurs...

Si certaines erreurs peuvent être détectées lors de la compilation (essentiellement les erreurs de syntaxe), voire lors de l'édition de liens, d'autres peuvent ne se manifester que dans la phase d'exécution (run-time errors) : elles devront être corrigées en reprenant le processus à la phase (1), l'édition des programmes source.

♡ ⇒ N.-B. : il est vivement conseillé de corriger les erreurs diagnostiquées par le compilateur dans l'ordre d'apparition, car une première erreur⁷ peut en induire une quantité d'autres⁸ dans la suite du programme (par exemple quand elle porte sur une déclaration de variable).

1.2.1 Édition du fichier source

N'importe quel éditeur de texte peut permettre la saisie des programmes sources, qui sont des fichiers texte. Mais certaines fonctions de l'éditeur seront particulièrement utiles pour éditer un langage structuré comme le fortran. Par exemple, sous l'éditeur `vi`, la cohérence des parenthèses éventuellement emboîtées peut être vérifiée par la simple frappe de la touche `%` qui, si le curseur est placé sur un délimiteur ouvrant, le déplace sur le délimiteur fermant associé, et réciproquement.

5. Voir la dernière version en discussion : <ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf>

6. cf. 1.2.4, p. 4 et Fig. 6.1, p. 59 pour le cas de plusieurs fichiers sources.

7. Certains compilateurs possèdent une option (`-fone-error` pour `g95`, cf. E.5.1, `-fmax-errors=1` pour `gfortran` cf. E.6.1,) permettant d'arrêter la compilation dès la première erreur.

8. Sous UNIX, les messages d'erreur sont envoyés sur la sortie d'erreur standard. C'est donc celle-ci qu'il faut rediriger pour maîtriser le défilement des messages soit dans un fichier par `2> fichier_erreurs`, soit dans un tube après fusion avec la sortie standard par `2>&1 | more` par exemple.

Certains éditeurs de texte sont dits « sensibles au langage », dont la syntaxe est décrite par l'intermédiaire de fichiers spécifiques. Parmi ces éditeurs « intelligents », citons **emacs** et **xemacs**, **asedit(nedit)** et la version **vim**⁹ (**vi improved**) de l'éditeur **vi** disponible sous LINUX.

Ils sont alors capables de repérer les commentaires, les chaînes de caractères, de reconnaître les mots-clés du langage, ses structures de contrôle, ses fonctions intrinsèques, et d'autoriser leur saisie abrégée. La structure syntaxique du fichier source peut alors être mise en évidence par l'usage de couleurs spécifiques, ce qui constitue une aide précieuse pour contrôler la syntaxe dès la phase d'édition.

Les éditeurs **emacs** et **xemacs** possèdent un *mode* fortran 90¹⁰ activé automatiquement au vu de l'extension du fichier édité. Dans ce mode, il est possible de lancer une compilation depuis l'éditeur, d'indenter automatiquement les structures... La présentation syntaxique du fichier source est elle-même configurable : par exemple, le niveau de marquage syntaxique est ajustable sur 4 niveaux et l'affichage des mots-clés peut être basculé de minuscules en majuscules, ou en minuscules avec l'initiale en capitale. Mais cet usage avancé d'**emacs** nécessite une installation adaptée, pas forcément disponible sur tous les serveurs.

Un fichier source élémentaire ne comportant qu'un programme principal débute par l'instruction **PROGRAM**, suivie du nom du programme et se termine par **END PROGRAM**, suivi du même nom. Par exemple, le programme suivant nommé **bienvenue** permet d'afficher le message **Bonjour**.

```
PROGRAM bienvenue
  WRITE(*,*) "Bonjour"
END PROGRAM bienvenue
```

1.2.2 Compilation et lien

La syntaxe des commandes de compilation est spécifique de chaque compilateur (*cf.* Annexe E, p 149), mais, au moins sous unix, certaines options importantes respectent une syntaxe générale :

- l'option **-c** qui permet de créer un fichier objet sans lancer l'édition de liens.
- l'option **-o** suivie d'un nom de fichier permet de spécifier le nom du fichier produit par le compilateur¹¹, que ce soit un fichier objet (dans le cas où l'on utilise aussi l'option **-c**) ou un fichier exécutable si l'édition de lien est lancée.
- l'option **-l** suivi du nom (dépouillé de **lib** en tête et de **.a** en fin) d'une bibliothèque à laquelle on fait appel lors de l'édition de liens. Pour utiliser par exemple la bibliothèque **libnrecip.a**, on lancera¹² :

```
g95 <essai>.f90 -lnrecip
```

- les options de type **-O<n>**, où **<n>** est un entier, contrôlent l'optimisation du code. Plus **<n>** est élevé, plus le code peut s'exécuter rapidement, mais les optimisations fortes ne sont pas forcément sûres. Il est conseillé de commencer la mise au point d'un code avec l'option **-O0**.

Dans la mise au point des programmes, il est recommandé d'utiliser toute la puissance du compilateur pour détecter dès la compilation les erreurs ou les fragilités d'un code. En choisissant les options adéquates, on lui demandera la plus grande sévérité et le maximum de diagnostics. On exploitera en particulier les avertissements (**warnings**) qui bien souvent détectent des sources d'erreur potentielles qui ne se manifestent qu'au moment de l'édition de lien ou de l'exécution (les messages d'erreur produits sont alors plus difficiles à interpréter). ⇐ ♥

9. Pour **vim**, la syntaxe du fortran 90 est décrite dans le fichier `/usr/share/vim/syntax/fortran.vim`.

10. Le mode fortran de **emacs** et **xemacs** est décrit dans le fichier lisp `f90.el`, dont la version compilée est `f90.elc`.

11. En l'absence de l'option **-o**, le fichier objet est nommé en substituant le suffixe **.o** au suffixe du fichier source et le fichier exécutable est nommé **a.out**

12. On insérera si nécessaire, avant l'option **-l**, le chemin d'accès à la bibliothèque, **-L<rep>** si elle n'est pas placée dans un des répertoires scrutés par défaut lors de sa recherche.

1.2.3 Exécution

Le fichier exécutable¹³ s'appelle par défaut `a.out`. On peut aussi spécifier son nom lors de la compilation par l'option `-o <fichier_exécutable>`. Ce fichier exécutable se comporte comme une commande du système d'exploitation, lancée en tapant `a.out`¹⁴ par exemple. En particulier, sous UNIX, ses flux d'entrée et de sortie peuvent être redirigés vers des fichiers ou d'autres commandes. De même, comme pour les autres processus, l'interruption en cours d'exécution est possible en frappant `^C`.

Débogueur

Dans la phase de mise au point et de recherche d'erreurs, on peut être amené à exécuter le programme sous le contrôle d'un débogueur (*debugger*) pour analyser en détail son comportement, notamment le contenu des mémoires où sont stockées les variables et le cheminement dans les instructions. Pour cela, il est nécessaire de compiler le programme avec l'option `-g` pour produire un exécutable spécifique que pourra lancer le débogueur.

Par exemple, sur les systèmes où le compilateur `gcc` est installé, le débogueur `gdb` est disponible et peut analyser des codes compilés avec `g95`. Après avoir lancé le débogueur, on peut placer des points d'arrêt (*breakpoints*) dans le programme source avant de le lancer sous son contrôle. Il est alors possible d'examiner des variables et même de les modifier avant de relancer l'exécution pas à pas ou jusqu'au prochain point d'arrêt...

1.2.4 Cas des fichiers sources multiples

Dès qu'une application est un peu importante, elle fait appel à plusieurs procédures (sous-programmes ou fonctions) qu'il est souvent préférable de stocker dans des fichiers distincts, à condition qu'il s'agisse de procédures externes (*cf.* 6.3.2, p. 56). Dans ce cas, on peut procéder à la *compilation séparée* des sous-programmes et fonctions, produisant ainsi des fichiers objets (que l'on pourra éventuellement rassembler dans une bibliothèque personnelle).

```
g95 -c <subpr1>.f90 <subpr2>.f90
```

Puis on lance la compilation du programme principal et l'édition de lien en fournissant au compilateur le fichier source du programme principal et la liste des fichiers objets produits par la compilation précédente.

```
g95 <principal>.f90 <subpr1>.o <subpr2>.o
```

Mais on verra (*cf.* 6.4, p. 58) qu'il est préférable d'intégrer les procédures dans des *modules* : la compilation d'un module produit, en plus du fichier objet, un « fichier de module » d'extension `.mod` permettant de stocker des informations (notamment les interfaces des procédures compilées¹⁵) utiles pour compiler ultérieurement d'autres procédures qui leur font appel : grâce à l'instruction `USE` portant sur ce module, elles pourront acquérir la visibilité sur l'interface des procédures du module (*cf.* Figure 6.1, p. 59).

Enfin, dans le cas où on fait appel à une bibliothèque constituée de modules pré-compilés, il faut éventuellement indiquer au compilateur dans quel répertoire trouver les « fichiers de module » associés à la bibliothèque grâce à l'option `-I<mod_rep>`¹⁶.

13. Sous UNIX, l'attribut exécutable est automatiquement positionné par l'éditeur de lien lors de la création de ce fichier. L'appel à la commande `chmod` n'est donc pas nécessaire.

14. Cela suppose que le répertoire où se situe l'exécutable fasse partie des chemins contenus dans la variable d'environnement `PATH` sous UNIX. Sinon, il faut préciser le chemin d'accès, en lançant par exemple `./a.out` si l'exécutable est dans le répertoire courant.

15. Ces fichiers jouent un rôle comparable aux fichiers d'entête du langage C.

16. Comme pour les fichiers à inclure par le préprocesseur.

L'outil make

La maintenance (génération et mise à jour) d'applications s'appuyant sur plusieurs fichiers source et éventuellement des bibliothèques peut être automatisée par des outils comme l'utilitaire `make` sous UNIX.

La commande `make` permet d'automatiser la génération et la mise à jour de *cibles* (*target*), en général un fichier exécutable, qui dépendent d'autres fichiers, par exemple les fichiers sources, en mettant en œuvre certaines *règles* (*rules*) de construction, constituées de commandes unix.

Elle s'appuie sur un fichier `makefile` qui liste les cibles, décrit l'arbre des *dépendances* et les règles de production des cibles. Pour plus détails, on pourra consulter la documentation en ligne de `gnumake` : <http://www.gnu.org/software/make/manual/make.html>.

Le compilateur `g95` possède une option permettant d'aider à l'écriture des dépendances après une compilation effectuée à la main :

`g95 -M <fichier.f90>` affiche les dépendances du fichier `<fichier.o>`.

1.3 Les éléments du langage

1.3.1 Les caractères du fortran

Les caractères du fortran 90

L'ensemble des caractères du fortran 90 est constitué des caractères alphanumériques, de signes de ponctuation et de quelques symboles spéciaux. Il comprend :

- les minuscules (lettres de `a` à `z`),
- les majuscules (lettres de `A` à `Z`),
- les chiffres arabes de `0` à `9`,
- le caractère souligné : « `_` »,
- et les symboles suivants¹⁷, parmi lesquels `$` et `?` n'ont pas de signification particulière dans le langage :

	+	-	=	*	()
,	.	:	;	'	"	/
<	>	%	!	&	\$?

Les caractères du fortran 2003

⇐ f2003

La norme 2003 a complété le jeu de caractères du fortran par les caractères spéciaux suivants :

~	^	\	{	}	'	[]		#	@
---	---	---	---	---	---	---	---	--	---	---

Les crochets carrés `[` et `]` peuvent être utilisés comme délimiteurs dans les constructeurs de tableaux monodimensionnels (cf. 7.1.3, p. 72).

1.3.2 Les identificateurs des objets

Les objets (variables, procédures et unités de programme) sont identifiés à l'aide de mots commençant par une lettre et constitués en fortran 95 d'au plus 31 caractères alphanumériques¹⁸ (plus le souligné « `_` » qui permet un pseudo-découpage des noms en mots). Le nombre de caractères des identificateurs est porté à 63 en fortran 2003. Contrairement à d'autres langages (notamment le C et unix), le fortran ne distingue pas majuscules et minuscules en dehors des chaînes de caractères.

⇐ f2003

N.-B. : pour des raisons de lisibilité, il est évidemment déconseillé d'en profiter pour introduire des variantes dans l'orthographe des identificateurs, nommant successivement par exemple une même variable `TOTAL`, puis `total` quelques lignes plus loin. On préconise à l'inverse de réserver les

⇐ ♥

17. Le langage C comporte plus de caractères avec les délimiteurs (accolades `{}`, crochets `[]`), la contre-oblique `\`, les opérateurs `%`, `~`, `^` et `|`. Le caractère `#` est en fait interprété par le préprocesseur qui peut agir aussi bien sur un programme en C que sur un programme en fortran.

18. En fortran 66 standard, la norme ne différenciait que les six premiers caractères. Heureusement, la plupart des compilateurs sont plus tolérants ce qui permet de choisir des identificateurs plus évocateurs.

capitales pour les mots clefs du langage¹⁹ et d'écrire les identificateurs définis par l'utilisateur en minuscules, ou éventuellement avec les initiales en majuscules pour mettre en évidence les mots dans les noms des objets, comme par exemple dans `TotalPartiel`.

1.4 Les formats du fortran

Pour des raisons de compatibilité ascendante, les compilateurs fortran 90 acceptent deux formats pour les fichiers sources :

- *le format fixe* (fixed format) qui est celui du fortran 77 avec deux extensions ;
- *le format libre* (free format), qui permet une écriture plus souple du code source, et est fortement préconisé²⁰ pour les programmes nouveaux.

Ces deux formats sont incompatibles et le compilateur doit être informé (soit au vu de l'extension du fichier, soit grâce à une option, cf. annexe E) du format choisi dans les fichiers sources qui lui sont soumis. L'utilisation conjointe de sources aux deux formats est possible en compilant séparément les fichiers sources au format fixe, puis ceux au format libre, et en assemblant ensuite les objets.

1.4.1 Format fixe

- En format fixe, les instructions peuvent s'étendre de la colonne 7 jusqu'à la colonne 72.
- Au-delà de la colonne 72, les caractères ne sont pas pris en compte par un compilateur standard²¹ ;
- Une ligne de commentaire peut être introduite grâce au caractère `C` placé en première colonne.
- Une ligne ne peut contenir qu'une instruction, mais une instruction peut s'étendre sur plusieurs lignes en plaçant un caractère quelconque, mais différent de 0 et d'un blanc en colonne 6 des lignes de continuation. Toutefois, la coupure de ligne ne peut pas se faire à l'intérieur d'une chaîne de caractères délimitée par des `"..."` ou des `'...'`.
- En dehors d'une ligne de commentaires, les colonnes 2 à 5 sont destinées à des étiquettes numériques permettant de repérer certaines instructions (pour les formats, les branchements ou les boucles).

Les deux extensions suivantes au format du fortran 77 définissent le format fixe du fortran 90 :

- en plus de `C`, les caractères `c`, `*` et `!` placés en première colonne sont considérés comme introducteurs de commentaire.
- une ligne peut contenir plusieurs instructions, à condition qu'elles soient séparées par des `« ; »`.

```

C      exemple d'extrait de fichier source fortran au format fixe
C
C234567890123456789012345678901234567890123456789012345678901234567890
C      1          2          3          4          5          6          7          8
C                                     | | | | | | | |
C      SOMME = TERME_1 +
C      1          TERME_2 +
C      1          TERME_3
C      début de la boucle
C      DO 100 I = 1, N                                DEBUT
C          K = K + I
C      100 CONTINUE                                    FIN
C      fin de la boucle

```

19. Rappelons que certains éditeurs (Language Sensitive Editors), tels qu'`emacs` (cf. 1.2.1, p. 2), permettent de basculer globalement la présentation des mots-clefs du fortran entre minuscules, majuscules ou capitalisation des initiales.

20. Le format fixe est déjà obsolète en fortran 95.

21. Cependant, en utilisant certaines options des compilateurs (parfois activées par défaut), les colonnes de 73 à 80 (ou 132) peuvent être interprétées comme des instructions.

1.4.2 Format libre du fortran 90

- En format libre, les instructions peuvent s'étendre de la colonne 1 à la colonne 132.
- Le caractère « ! »²² permet d'introduire un *commentaire* n'importe où dans la ligne; ce commentaire se termine à la fin de la ligne²³.
- Une ligne peut contenir plusieurs instructions, séparées par un point-virgule « ; »²⁴.
Pour des raisons de lisibilité, il est cependant déconseillé de grouper plusieurs instructions sur une même ligne. ← ♥
- Terminer une ligne d'instruction par le *caractère de suite* & indique que l'instruction se continue sur la ligne suivante²⁵. Dans le cas où la coupure de ligne intervient au sein d'une chaîne de caractères, il faut insérer un & comme premier caractère non-blanc de la ligne de continuation pour indiquer où reprend la chaîne.
- Les caractères ! et & perdent leur interprétation spéciale lorsqu'ils sont situés à l'intérieur d'une chaîne de caractères (entre "... " ou entre '...'). Mais, dans le cas où la chaîne n'est pas fermée en fin de ligne, un & en *dernier caractère* non blanc est interprété comme indiquant que la fin de ligne ne termine pas l'instruction.

Noter que les espaces sont significatifs en format libre. En particulier, en l'absence de séparateur, un blanc est obligatoire pour délimiter deux identificateurs.

```
PROGRAM essai
!           exemple de source fortran 90 au format libre
IMPLICIT NONE
REAL :: somme, terme_1 = 1., terme_2 = 2., terme_3 = 3.
!
! une instruction longue scindée en plusieurs lignes
somme = terme_1 + &           ! description du premier terme
        terme_2 + &           ! description du deuxième terme
        terme_3               ! description du troisième terme
! cas où la coupure se fait au sein d'une chaîne de caractères
WRITE (*,*) ' chaîne de caractères comportant plusieurs lignes dans &
            &le programme source'
!           ce & est obligatoire pour marquer la continuation de la chaîne
END PROGRAM essai
```

L'ordre d'écriture suivant

```
WRITE (*, *) 'chaîne avec des & et des ! normaux et deux &&
            & spéciaux' ! suivie d'un commentaire
```

affichera sur une seule ligne :

```
chaîne avec des & et des ! normaux et deux & spéciaux
```

22. En fortran 90, c'est le seul introducteur de commentaire; en particulier, « C », « c » et « * » ne permettent plus d'introduire un commentaire dans le source.

23. Le caractère « ! », introducteur de commentaire n'est pas un délimiteur, contrairement à ce qui se passe en C avec le couple /* ... */. Il est à comparer avec le couple // introducteur de commentaires du C99 et du C++.

24. Mais, contrairement à ce qui se passe en C, une ligne d'instruction ne se termine pas par un « ; ».

25. Cette technique ne peut pas être employée pour des lignes de commentaire. À l'inverse, et bien que cette pratique soit déconseillée pour manque de lisibilité, on peut insérer une ou plusieurs lignes de commentaires (chacune introduite par un !) entre les différentes lignes (terminées, sauf la dernière, par un &) constituant une instruction.

Chapitre 2

Types, constantes et variables

2.1 Les types intrinsèques

Le langage fortran 90 possède les *types intrinsèques* ou prédéfinis suivants :

- CHARACTER : chaîne de caractères¹
- LOGICAL : booléen
- types numériques :
 - type numérique représenté exactement :
 - INTEGER : entier
 - types numériques représentés approximativement :
 - REAL : réel en virgule flottante
 - DOUBLE PRECISION : flottant en précision étendue (à éviter : lui préférer une variante de type déterminée de façon plus portable, grâce à l'attribut KIND, cf. 2.5.2, p. 15)
 - COMPLEX : nombre complexe flottant

2.2 La représentation des types numériques

Les entiers sont représentés en mémoire de façon exacte, donc sans perte de précision, mais leur domaine est limité par le nombre de bits choisi pour les représenter.

Au contraire, les réels ne peuvent pas, en général, être représentés exactement en mémoire : c'est pourquoi on évitera les tests d'égalité sur les réels ; c'est aussi ce qui explique l'impossibilité d'appliquer à des expressions réelles des tests de type énumération de valeurs².

Pour plus de détail sur la représentation des nombres et l'implémentation des opérations arithmétiques, on consultera le chapitre 9 « Arithmétique des ordinateurs » de [STALLINGS \(2003\)](#).

2.2.1 Représentation des entiers

Si la représentation la plus courante des entiers positifs est celle de la base 10 (qui utilise 10 symboles de 0 à 9), la plupart des ordinateurs utilisent une représentation en base 2 ne nécessitant que 2 symboles (0 et 1) ; en représentation binaire, l'équivalent du chiffre de la représentation décimale est le *bit*. La représentation en *octal* (base 8) utilise les 8 symboles de 0 à 7 alors que la représentation *hexadécimale*³ (base 16) utilise 16 symboles : 0,1, ..., 9, A, B, ...F.

Représentation des entiers positifs

La représentation des entiers positifs utilise des codes à poids dont les poids sont les puissances décroissantes de la base b .

$$n = \sum_{i=0}^{q-1} p_i b^i \quad \text{où } 0 \leq p_i < b \quad (2.1)$$

1. En fortran, un caractère est stocké dans une chaîne de type CHARACTER et de longueur 1, contrairement à ce qui se passe en langage C, où seul le type caractère est natif et les chaînes ne sont que des tableaux de caractères.

2. `select case` en fortran ou `switch ... case` en C.

3. La base 16 a été utilisée sur certains calculateurs IBM

Réciproquement, les q coefficients p_i peuvent être obtenus par des divisions entières successives par b : p_0 est le reste de la division de n par b , p_1 le reste dans la division par b^2 , ... Dans le cas où $b = 2$, chaque coefficient p_i peut être représenté par un bit. Par exemple, pour l'entier 13 :

$$13 = 1 \times 10^1 + 3 \times 10^0 \quad \text{noté } 13_{10} \text{ en base 10}$$

$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \quad \text{noté } 1101_2 \text{ en base 2}$$

Si on consacre q bits à la représentation d'un entier positif, on peut couvrir le domaine $[0, 2^q - 1]$.

Représentation des entiers négatifs

Les entiers négatifs sont représentés habituellement avec le symbole complémentaire « - » qu'il va falloir traduire par un bit complémentaire appelé *bit de signe* en codage binaire.

En binaire, un entier relatif est donc représenté sur $q + 1$ bits. On choisit de coder le signe sur le bit de plus fort poids avec 0 pour les entiers positifs et 1 pour les négatifs. Mais si on codait simplement la valeur absolue en binaire de n sur les q bits restants, une opération arithmétique élémentaire comme l'addition entre entiers de signes différents ne s'effectuerait pas comme celle de deux entiers de même signe.

On choisit au contraire de représenter l'entier n négatif par un bit de signe à 1 suivi des bits du complément à 2^{q+1} de $|n|$, c'est-à-dire ceux de l'entier positif $2^{q+1} - |n|$. On parle alors (abusivement) de complément à deux, obtenu en inversant tous les bits puis en ajoutant 1 avec perte de la retenue. L'arithmétique entière ainsi mise en œuvre fonctionne modulo 2^{q+1} . Le domaine couvert va donc de -2^q à $2^q - 1$ (voir par exemple 2.2.3, page 12). Par exemple, les entiers codés sur 8 bits, soit 1 octet ($q = 7$) couvrent le domaine $-2^7 = -128$ à $2^7 - 1 = +127$:

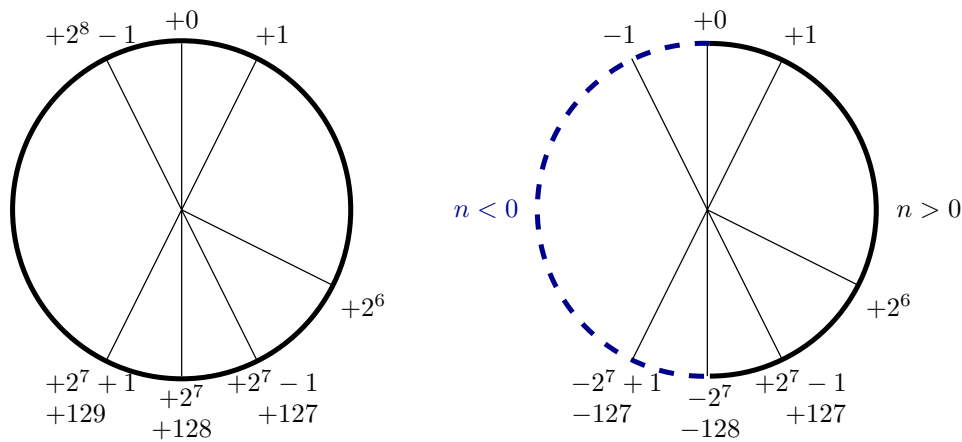


FIGURE 2.1 – Entiers positifs (à gauche) et entiers relatifs (à droite) sur 8 bits : pour passer des positifs aux relatifs, on soustrait 2^8 dans la partie gauche (pointillée) du cercle. La discontinuité initialement à gauche de 0 pour les positifs, se retrouve entre +127 et -128.

n	signe	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-128	1	0	0	0	0	0	0	0
-127	1	0	0	0	0	0	0	1
...
-1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0	1
...
+127	0	1	1	1	1	1	1	1

En utilisant le format binaire d'entrée-sortie B (cf. 5.4.1, p. 42) comme convertisseur binaire-décimal, on peut vérifier simplement la représentation des entiers.

<pre>PROGRAM entier_binaire IMPLICIT NONE INTEGER :: i0 ! entier par défaut (32 bits) ! entiers pour stocker au moins 100 => 8 bits INTEGER, PARAMETER :: k1=SELECTED_INT_KIND(2) INTEGER(kind=k1) :: i1 WRITE(*, *) "entiers sur",DIGITS(i0),"bits + signe" WRITE(*, *) "soit ",BIT_SIZE(i0), "bits avec signe" WRITE(*, "(i11)") HUGE(i0) WRITE(*, "(b32.32)") HUGE(i0) WRITE(*, "(i11)") -HUGE(i0)-1 WRITE(*, "(b32.32, /)") -HUGE(i0)-1 WRITE(*, *) "entiers sur",DIGITS(i1),"bits + signe" WRITE(*, *) "soit ",BIT_SIZE(i1), "bits avec signe" WRITE(*, "(i4, 1x, b8.8)") HUGE(i1), HUGE(i1) WRITE(*, "(i4, 1x, b8.8)") -HUGE(i1)-1_k1, & -HUGE(i1)-1_k1 END PROGRAM entier_binaire</pre>	<pre>entiers sur 31 bits + signe soit 32 bits avec signe 2147483647 011111111111111111111111111111111111 -2147483648 1000000000000000000000000000000000 entiers sur 7 bits + signe soit 8 bits avec signe 127 01111111 -128 10000000</pre>
---	---

Traditionnellement, comme en décimal, on range les bits par poids décroissant, le plus fort poids à gauche. Mais des variantes dans l'ordre de stockage⁴ des octets ou des groupes de deux octets existent : on distingue les conventions *little-endian* où les octets de poids fort sont stockés en fin et *big-endian* où les octets de poids fort sont stockés en tête.

2.2.2 Représentation des réels en virgule flottante

Si on codait les réels en virgule fixe⁵, la précision absolue de la représentation serait fixe ; mais la précision relative dépendrait de l'ordre de grandeur. Pour conserver une précision **relative** plus indépendante de l'ordre de grandeur, c'est-à-dire pour travailler avec un nombre fixe de chiffres significatifs, on préfère une représentation en *virgule flottante*⁶, avec une *mantisse*, représentant la partie fractionnaire et un exposant (cf. (2.2), p. 10). Par exemple en base 10, avec 4 chiffres après la virgule, on peut comparer les deux représentations approchées (par troncature et non par arrondi, pour simplifier) :

nombre exact	virgule fixe	virgule flottante
0.000123456789	.0001	0.1234×10^{-3}
0.00123456789	.0012	0.1234×10^{-2}
0.0123456789	.0123	0.1234×10^{-1}
0.123456789	.1234	0.1234×10^0
1.23456789	1.2345	0.1234×10^1
12.3456789	12.3456	0.1234×10^2
123.456789	123.4567	0.1234×10^3
1234.56789	1234.5678	0.1234×10^4

Représenter un réel (non nul) en virgule flottante, c'est en donner une approximation finie en le développant dans une base b sous la forme :

$$r = s b^e m = s b^e \sum_{i=1}^q p_i b^{-i} \quad (2.2)$$

4. On peut faire l'analogie avec les modes d'écriture de la date suivant les pays qui diffèrent notamment par l'ordre entre le jour, le mois et l'année.

5. On peut, pour simplifier, envisager le cas de la base décimale, et faire l'analogie avec le format **F** en fortran ou %f en C, avec un nombre fixe de chiffres après la virgule.

6. Pour poursuivre la comparaison, cela correspond à la notation exponentielle en format **E** en fortran ou %e en C

où

- $s = \pm 1$ est le signe ;
- e est un entier qualifié d'*exposant* ;
- la partie fractionnaire m est la *mantisse*.

Afin de choisir parmi les combinaisons de mantisse et d'exposant (par exemple $1,23410^{-1}$; $0,123410^0$ ou $0,0123410^1$ en décimal), on impose que la mantisse soit comprise entre $1/b$ inclus et 1 exclus soit $p_1 \neq 0$ (donc la deuxième combinaison en décimal).

Chaque variante du type réel (déterminée via le paramètre de codage KIND) est caractérisée par :

- un nombre q de symboles (chiffres décimaux en base 10 et bits en base 2) de la mantisse qui fixe la précision relative des réels ;
- un nombre de symboles pour l'exposant qui fixe le domaine de valeurs couvert.

Dans cette représentation, en décimal, les réels en virgule flottante sont répartis en progression arithmétique dans chaque *décade* avec un nombre fixe de valeurs par décade et un pas multiplié par 10 à chaque changement de décade. En binaire, c'est dans chaque *octave* (intervalle de type $[2^n, 2^{n+1}[$) qu'ils sont en progression arithmétique avec un nombre 2^{q-1} de valeurs par octave ; le pas double à chaque changement d'octave. (cf. Annexe C, p. 140 sur le codage IEEE).

2.2.3 Domaine et précision des types numériques

Les nombres étant stockés sur un nombre de bits fixe, ne sont représentés que dans un domaine fini. De plus, seul le type entier permet une représentation exacte des valeurs numériques, alors que les nombres réels ne sont en général représentés en virgule flottante qu'approximativement ⁷.

Fonctions d'interrogation sur le codage des types numériques

Plusieurs fonctions intrinsèques⁸ permettent de caractériser la représentation du type numérique de leur argument du point de vue :

- du nombre de **digits** utilisés pour le représenter : **DIGITS(x)** donne le nombre de symboles (c'est-à-dire de bits dans le cas usuel de la base 2) consacrés au stockage de la mantisse de **x** dans le type de **x**. Plus précisément :
 - si **x** est un entier, **DIGITS(x)** est le nombre q de bits sur lesquels l'entier est stocké hors bit de signe. Il détermine donc le domaine couvert par ce type. Dans le cas entier, **BIT_SIZE(x)** rend le nombre total de bits utilisés pour le stocker (bit de signe inclus).
 - si **x** est un réel en virgule flottante, **DIGITS(x)** est le nombre de bits (ici q)⁹ sur lesquels est stockée sa mantisse. Il détermine alors la précision relative de la représentation de **x**.
- du domaine couvert par un type donné :
 - **HUGE(x)**, la plus grande valeur (entière ou réelle) représentable dans le type de **x**.
 - **TINY(x)**, la plus petite valeur absolue flottante représentable¹⁰ dans le type de **x**.
 - **RANGE(x)** qui donne :
 - la puissance de 10 du plus grand entier défini dans le type de **x** s'il s'agit d'un entier : ainsi tout entier **x** tel que $|x| \leq 10^r$ où r est la valeur de **RANGE** est représentable dans ce type et **RANGE(x) = INT(LOG10(HUGE(x)))**
 - la plus petite des valeurs absolues des puissances de 10 du plus petit et du plus grand des réels dans le type de **x**, si **x** est un flottant ; ainsi tout réel **x** s'écrivant sous la forme $x = \pm 0.????? 10^k$ avec k entier et $|k| \leq r$ où r est la valeur de **RANGE**, est représentable dans ce type ; **RANGE(x) = INT(MIN(LOG10(HUGE(x)), -LOG10(TINY(x))))**
- de la précision de la représentation en virgule flottante :

7. Plus précisément, ne peuvent éventuellement être exactement représentés que les rationnels dont le dénominateur est une puissance de la base, donc en pratique du type $n/2^p$ avec des contraintes sur n et p . Ainsi les entiers de valeur absolue assez faible, et certains nombres rationnels de dénominateur 2^p sont représentés exactement en virgule flottante. En revanche la majorité des nombres décimaux, 0.1 par exemple, ne sont pas représentables exactement.

8. Seul le type et non la valeur de l'argument de ces fonctions d'interrogation a une importance.

9. Dans le cas d'une mantisse normalisée, son premier bit, toujours égal à 1, n'est pas stocké, ce qui permet « gagner » un bit, mais il est pris en compte dans la fonction **DIGITS**.

10. On se limite ici aux représentations normalisées, cf. annexe C, page 140, sachant qu'il est possible de représenter des valeurs absolues plus petites au prix d'une perte de précision.

- `PRECISION(x)` donne le nombre de chiffres décimaux *significatifs* d'un réel du type de `x`.
- `EPSILON(x)` donne la plus petite valeur réelle positive qui n'est pas négligeable devant 1. dans le type de `x`. C'est la plus petite valeur positive qui, ajoutée à 1., donne le flottant successeur de 1. (le plus proche de 1. par excès, dans la représentation de `x`).

Si un entier `i` est stocké sur `q+1` bits, `BIT_SIZE(i)` vaut `q+1`, `DIGITS(i)` vaut `q` et `HUGE(i)` vaut $2^q - 1$.

D'autres fonctions intrinsèques d'interrogation permettent de préciser la représentation des types numériques :

- `RADIX(r)` donne la base b (en général 2) employée dans la représentation (2.1) pour les entiers ou (2.2) pour les réels du type de `r`;
- `EXPONENT(r)` rend un entier donnant l'exposant e de `r` de la représentation (2.2);
- `FRACTION(r)` rend un réel donnant la mantisse m de `r` de la représentation (2.2);
- `NEAREST(r, s)` rend le réel le plus proche de `r` mais distinct de `r` dans la représentation (2.2) selon la direction fixée par le signe de `s` (successeur ou prédécesseur de `r`);
- `SPACING(r)` rend la plus petite distance entre `r` et son voisin le plus proche; (successeur ou prédécesseur sont en général à égale distance qui est le pas de la progression arithmétique des réels dans une octave, sauf si `r` est une puissance entière de 2).

Caractéristiques des types numériques prédéfinis

Les caractéristiques des *types numériques prédéfinis* dépendent à la fois du processeur et des compilateurs (souvent au travers d'options spécifiques). Le programme suivant affiche quelques caractéristiques des entiers et des réels par défaut.

```
PROGRAM precision_numerique
IMPLICIT NONE
INTEGER :: i ! type entier par défaut (en général 32 bits)
REAL    :: r ! type réel par défaut (en général 32 bits)
WRITE(*,*) 'entiers par défaut, kind = ', KIND(i)
WRITE(*,*) 'digits = ', DIGITS(i), ' huge = ', HUGE(i), ' range = ', RANGE(i)
WRITE(*,*) 'réels par défaut, kind = ', KIND(r)
WRITE(*,*) 'digits = ', DIGITS(r), ' précision = ', PRECISION(r), &
           ' epsilon = ', EPSILON(r), ' tiny = ', TINY(r), &
           ' huge = ', HUGE(r)
END PROGRAM precision_numerique
```

Types entiers par défaut Sur un processeur 32 bits, les entiers par défaut sont stockés sur 4 octets donc 31 bits pour la valeur absolue plus un bit de signe. La variante de type (`KIND`) des entiers représente généralement le nombre d'octets sur lesquels est codé cet entier¹¹.

Sur un processeur de type PC 64 bits, les entiers par défaut peuvent être stockés sur 8 octets, (même si ce n'est pas forcément la règle), donc 63 bits pour la valeur absolue plus un bit de signe. Des options de compilation permettent d'imposer les entiers sur 64 bits (*cf.* Annexe E.7.1 p. 155).

nombre d'octets	BIT_SIZE	DIGITS	HUGE	RANGE
4	32	31	$2147483647 = 2^{31} - 1$	9
8	64	63	$9223372036854775807 = 2^{63} - 1$	18

Le comportement en cas de dépassement de capacité entière dépend du compilateur et des options choisies. Avec les compilateurs `g95` et `gfortran`, l'option `-ftrapv` provoque un arrêt du programme en cas de dépassement de capacité lors d'un calcul en entier.

11. Avec le compilateur `nagfor` de NAG, par défaut, les variantes de type sont numérotées en séquence (option `-kind=sequential` du compilateur). Mais si on précise l'option de compilation `-kind=byte`, les valeurs de `KIND` représentent le nombre d'octets occupés par la variante de type (*cf.* Annexe E.2, p. 150).

Types réels par défaut Les réels sont, sur la plupart des machines, stockés sur 4 octets, dont 24 bits (dont 1 pour le signe) pour la mantisse et 8 bits pour l'exposant. Avec des options de compilation (cf. Annexe E.7.2 p. 155), on peut passer à 8 octets dont 53 bits (dont 1 pour le signe) pour la mantisse et 11 bits pour l'exposant (variante de réels aussi disponible sous le type DOUBLE PRECISION). On se reportera à l'annexe C p. 140 pour une description de la norme IEEE 754 qui définit une méthode de représentation portable des réels.

nb d'octets	DIGITS	PRECISION	EPSILON	TINY	HUGE
4	24	6	1.192093E-7	1.175494E-38	3.402823E+38
8	53	15	2.220446049250313E-016	2.225073858507201E-308	1.797693134862316E+308

2.3 Déclaration

Les objets doivent être déclarés en tout début¹² de programme, sous-programme, fonction ou module, précisément avant les instructions exécutables, selon la syntaxe générale suivante :

```
<type> [, <liste_d_attributs>::] <liste_d_objets>
```

Le séparateur « :: » n'est pas obligatoire si la déclaration ne comporte pas d'attributs ni d'initialisation, mais il est fortement conseillé. Pour chaque objet déclaré, on peut préciser des attributs qui seront décrits en détail plus loin : ⇐ ♥

- pour les variables :
 - PARAMETER constante nommée (cf. 2.5.1, p. 15)
 - DIMENSION profil d'un tableau (cf. 7.1.1, p. 70)
 - ALLOCATABLE objet alloué dynamiquement (cf. 7.5.2, p. 81)
 - SAVE objet statique (cf. 6.1.2, p. 52)
 - POINTER objet défini comme un pointeur (cf. 11.1, p. 104)
 - TARGET objet cible potentielle d'un pointeur (cf. 11.1, p. 104)
- pour les procédures :
 - EXTERNAL caractère externe d'une procédure (cf. 6.5.3, p. 64)
 - INTRINSIC caractère intrinsèque d'une procédure (cf. 6.5.3, p. 64)
- pour les arguments des procédures :
 - INTENT vocation (IN, OUT ou INOUT) d'un argument muet d'une procédure (cf. 6.1.4, p. 52)
 - OPTIONAL caractère optionnel d'un argument muet d'une procédure (cf. 6.5.1, p. 63)
 - VALUE pour le passage d'argument par copie de valeur (cf. 12.4.1, p. 117) ⇐ f2003
- pour les objets encapsulés dans des modules (cf. 6.4.3 p. 61) :
 - PUBLIC rend accessible un objet à l'extérieur du module courant : c'est l'attribut par défaut
 - PRIVATE limite la visibilité d'un objet au module courant
 - PROTECTED (fortran 2003 seulement) interdit la modification de l'objet en dehors du module sans restreindre sa visibilité (impose ainsi de passer par une procédure du module pour le modifier) ⇐ f2003

Par défaut, les variables numériques non déclarées suivent un *typage implicite* déterminé par leur initiale : les variables dont l'initiale est I, J, K, L, M ou N sont des entiers et les autres des flottants. L'ordre IMPLICIT permet de modifier éventuellement cette convention. La déclaration d'une variable impose explicitement son type et prévaut sur ces conventions.

Mais, il est très fortement conseillé de s'obliger à déclarer toutes les variables, sans mettre en œuvre le typage implicite¹³, grâce à l'ordre IMPLICIT NONE, placé avant toutes les déclarations. Cette contrainte permet en autres de confier au compilateur le repérage des erreurs typographiques dans les noms de variables : si on a déclaré la variable `somme` et que l'on utilise ensuite `some` sans déclaration, le compilateur va diagnostiquer une erreur. Elle oblige aussi à constituer en tête de programme une sorte de dictionnaire des variables qu'il est conseillé de commenter et qui facilite la lecture et la maintenance du code. ⇐ ♥

12. En fortran 2008, il est possible de déclarer tardivement des objets (comme en C99 ou C++) au début d'une structure de bloc (BLOCK ... END BLOCK). La portée et la durée de vie de ces objets sont alors limitées au bloc.

13. Ainsi, le comportement du fortran 90 se rapproche de celui du langage C, dans lequel toutes les variables doivent être déclarées.

```

IMPLICIT NONE
CHARACTER(LEN=5) :: nom_a_5_lettres ! chaîne de 5 caractères
LOGICAL          :: test, test1
REAL             :: a, longueur ! on ignore le typage implicite pour longueur
COMPLEX         :: nombre_complexe

```

On peut spécifier une variante de type en indiquant la valeur du paramètre `KIND` entre parenthèses, mais cette valeur doit être une constante :

```

<type>(KIND=<kind>) :: <liste_d_objets>

```

2.4 Les constantes

Les *constantes* sont des objets dont la valeur ne peut pas être modifiée lors de l'exécution du programme; elles ne peuvent constituer le membre de gauche d'une instruction d'affectation. On distingue les vraies constantes (*literal constants*) ou constantes non nommées d'une part et les constantes nommées d'autre part (*cf. 2.5.1, p. 15*), qui sont évaluées lors de la compilation. Il est fortement conseillé de nommer les constantes utilisées dans un programme, pour s'assurer que la même valeur est utilisée dans tout le programme et pour simplifier un éventuel changement de précision d'une constante flottante, voire la transformer en une variable lors d'une généralisation du programme.

Exemples de constantes vraies dans les types prédéfinis

```

-250          ! entier
17            ! entier
B'101'       ! entier en binaire : 5
O'17'        ! entier en octal : 15
Z'1F'        ! entier en hexadécimal : 31
3.14         ! réel en notation décimale
-.33         ! réel en notation décimale
2.1e3        ! réel en notation mantisse et exposant
-2e-3        ! réel en notation mantisse et exposant
2.1d3        ! double précision
(1.5, -2.)   ! complexe 1,5 - 2i
.true.       ! booléen: vrai
.false.      ! booléen: faux
'bonjour !'  ! chaîne de caractères délimitée par des apostrophes (')
"bonjour !"  ! chaîne de caractères délimitée par guillemets (")
'M1 "P & A"' ! chaîne de caractères (entre les ', seul ' est interprété)
"aujourd'hui" ! chaîne de caractères (l'apostrophe est autorisée entre les ")
'aujourd'hui' ! chaîne de caractères (l'apostrophe doublée permet
              ! d'insérer une seule apostrophe dans la chaîne)
''           ! chaîne de caractères vide

```

2.5 Initialisation

Il est possible d'initialiser des objets dès leur déclaration¹⁴, en leur affectant comme valeur des constantes vraies du même type¹⁵, par exemple

14. Une variable locale à une procédure devient statique si on l'initialise au moment de la déclaration, elle se comporte donc comme si elle possédait l'attribut `SAVE` (*cf. 6.1.2, p. 52*). Ce n'est pas le cas en C, où l'initialisation se fait à chaque appel de la fonction, sauf si on spécifie l'attribut `static` pour la variable.

15. Veiller à initialiser par exemple les variables de type `DOUBLE PRECISION` avec des constantes du même type, sous peine de perte de précision : dans l'exemple qui suit, les derniers chiffres significatifs de `r` n'auraient pas de sens en l'absence de `d0`

```

IMPLICIT NONE
CHARACTER(LEN=5) :: nom_a_5_l lettres = 'début'
LOGICAL          :: test = .TRUE. , test1 = .FALSE.
REAL             :: a = 0. , longueur = 1.e3
COMPLEX         :: nombre_complexe = (1.,-1.)
DOUBLE PRECISION :: r = 1.23456789012345d0

```

Mais des expressions simples, dites expressions d'initialisation peuvent aussi être utilisées pour initialiser les variables : elles doivent pouvoir être évaluées par le compilateur.

```

IMPLICIT NONE
REAL :: longueur = 10., largeur = 2.
REAL :: surface = longueur * largeur
REAL :: pi = 4.*ATAN(1.)
REAL :: enorme = HUGE(1.e0)

```

Sont notamment autorisés dans les expressions d'initialisation les fonctions intrinsèques élémentaires (cf. annexe A, p. 125), les fonctions intrinsèques de transformation (portant sur les tableaux), et les constructeurs de tableaux¹⁶ (cf. 7.1.3, p. 72) et de structures (cf. 9.2.1, p. 93). ⇐ f2003

2.5.1 Les constantes nommées (ou symboliques),

Pour déclarer une *constante nommée*, c'est à dire attribuer une valeur fixée à la compilation à un identificateur, il faut lui adjoindre l'attribut `PARAMETER`, et l'initialiser.

```

IMPLICIT NONE
INTEGER, PARAMETER :: n_points = 100
REAL, PARAMETER    :: pi = 3.1416

```

Il n'est alors plus possible de modifier la valeur de ces constantes nommées, par exemple en écrivant `pi=3.15`.

Les constantes entières nommées peuvent aussi être utilisées pour paramétrer les dimensions de certains tableaux (cf. 7.5.1, p. 79), qui seront ainsi fixées au moment de la compilation¹⁷.

2.5.2 Variantes des types prédéfinis

Les types prédéfinis comportent des *variantes (sous-types)* selon le nombre d'octets choisis pour stocker l'objet, variantes que l'on peut sélectionner à l'aide du paramètre `KIND`¹⁸. À chaque type est associé un sous-type par défaut, qui peut dépendre de la machine ; ainsi, les types numériques prédéfinis ne garantissent pas une précision indépendante de la machine.

Si l'on veut s'assurer de la portabilité numérique d'une déclaration, en termes de *domaine* ⇐ ♥ (*range*) couvert ou de précision, il faut faire appel à des fonctions intrinsèques afin de choisir les sous-types en fonction du domaine et de la précision.

16. En fortran 95, les fonctions intrinsèques n'étaient généralement pas autorisées dans les expressions d'initialisation, à quelques exceptions près, parmi lesquelles les fonctions de précision numérique et certaines fonctions liées au « profil » des tableaux, dont `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, `TRIM`, `LBOUND`, `UBOUND`, `SHAPE`, `SIZE`, `KIND`, `LEN`, `BIT_SIZE`, `HUGE`, `EPSILON`, `TINY`...

Ainsi, `REAL :: enorme = HUGE(1.e0)` était autorisé, mais `REAL :: pi = 4.*ATAN(1.)` n'était pas admis en fortran 95. Ces restrictions ont été levées avec le fortran 2003.

17. En C, déclarer une variable avec l'attribut `const` permet de diagnostiquer une éventuelle tentative de modification (erreur détectée par le compilateur `gcc` depuis la version 4.0), mais ne suffit pas pour en faire une constante nommée, susceptible par exemple de définir la dimension d'un tableau. Pour ce faire, il faut recourir à la directive `#define` du préprocesseur.

18. Le compilateur NAG (cf. Annexe E.2, p. 150) fournit un module `f90_kind.f90` qui définit un ensemble de variantes de type.

- pour les entiers, la fonction `SELECTED_INT_KIND(r)` où `r` est un entier positif, renvoie un entier donnant le numéro de la variante du type entier qui permet de couvrir le domaine $[-10^{+r}, 10^{+r}]$. Si aucune variante entière ne convient, cette fonction renvoie la valeur `-1` ;
- pour les réels, la fonction `SELECTED_REAL_KIND([p] [,r])` renvoie un entier donnant le numéro de la variante du type réel dont la précision est au moins `p` (au sens donné par la fonction `PRECISION`) ou une étendue couvrant le domaine fixé par `r` (au sens donné par la fonction `RANGE`, c'est à dire dont la valeur absolue reste dans l'intervalle $[10^{-r}, 10^{+r}]$). Si aucune variante flottante ne convient, cette fonction renvoie une valeur négative.

Enfin la notion de portabilité numérique reste limitée par la disponibilité des variantes dans un environnement (processeur et compilateur) donné de programmation. Un programme qui requiert une précision ou un domaine exigeants pourra, dans certains environnements, ne pas trouver les variantes nécessaires pour s'exécuter. Mais il existe des bibliothèques qui permettent des calculs dans des précisions non limitées par le compilateur, par exemple avec des types dérivés, notamment `FMLIB` en `fortran`.

2.5.3 Exemples de constantes nommées spécifiant les variantes des types prédéfinis

Il faut en premier lieu demander au compilateur le paramètre (`KIND`) de la variante de type voulue avec `SELECTED_INT_KIND`, `SELECTED_CHAR_KIND` ou `SELECTED_REAL_KIND` : le résultat est une constante utilisable pour les déclarations de variables qui suivent et aussi pour désigner la variante associée des constantes. Le caractère souligné « `_` » permet de préciser la variante du type choisie : la variante¹⁹ est indiquée **après** la valeur pour une constante numérique et **avant** pour une constante chaîne de caractères (cf. 8.1.3, p. 85).

```
! recherche des variantes
INTEGER, PARAMETER :: ki = SELECTED_INT_KIND(10)      ! ki=8 sous g95 32/64bits
! choix d'un type entier permettant de représenter 10**10
INTEGER, PARAMETER :: kr = SELECTED_REAL_KIND(9,100) ! kr=8 sous g95 32/64bits
! => type réel permettant de représenter 10**100 avec 9 chiffres significatifs
! déclaration des variables dans ces variantes
INTEGER(KIND = ki) :: i
REAL(KIND = kr)    :: r
WRITE(*,*) "ki = ", ki, " kr = ", kr ! dépendent du compilateur
i = 1000000000_ki      ! 1000000000_8 aurait été moins portable
r = 1.123456789e99_kr ! 1.123456789e99_8 aurait été moins portable
! r = 1.123456789e99 ! donnerait un dépassement de capacité
```

2.5.4 Une méthode modulaire pour fixer la précision des réels

Dans le cadre d'un programme sensible à la précision des réels, il est important de disposer d'un moyen rapide et portable de modifier le sous-type de tous les réels utilisés (constantes et variables), sans s'appuyer sur des options de compilation (cf. E.7.2). Il suffit pour cela de définir la variante de travail dans un module `my_precision` sous la forme d'une constante nommée (`wp`), puis de déclarer toutes les variables réelles dans ce sous-type. Noter qu'il faut aussi exprimer toutes les constantes réelles dans ce sous-type et spécifier le paramètre optionnel `KIND=wp` dans les fonctions où la généricité ne suffit pas à assurer un calcul dans le sous-type voulu. Sachant que les sous-types réels courants utilisent 32, 64 ou 80 bits, on pourra définir les paramètres de type associés (`sp`, `dp`, `edp`) dans un module préliminaire `real_precisions` et faire le choix (ligne 11) par renommage via `=>` dans le `USE` (cf. 6.4.3). C'est une méthode largement utilisée par les bibliothèques.

19. La variante elle-même peut être une constante vraie ou une constante nommée.

```

1  MODULE real_precisions
2      IMPLICIT NONE
3      ! les differentes variantes de reels disponibles (g95/gfortran)
4      INTEGER, PARAMETER :: sp=SELECTED_REAL_KIND(p=6)    ! simple precision 32 bits
5      INTEGER, PARAMETER :: dp=SELECTED_REAL_KIND(p=15)   ! double precision 64 bits
6      INTEGER, PARAMETER :: edp=SELECTED_REAL_KIND(p=18) ! extra double precision 80
7  END MODULE real_precisions
8
9  MODULE my_precision ! ne modifier que ce module pour choisir la precision wp
10     !use real_precisions, only : wp => sp ! par exemple simple precision
11     USE real_precisions, ONLY : wp => dp ! ou sinon double precision
12     !use real_precisions, only : wp => edp ! ou sinon extra double precision
13 END MODULE my_precision
14
15 PROGRAM test_precision
16 USE my_precision
17 IMPLICIT NONE
18 ! toujours utiliser le parametre (kind) de sous-type reel wp = working precision
19 ! dans les declarations, dans les constantes et comme argument optionnel
20 REAL(kind=wp) :: r1 = .1_wp ! constante en accord avec le type
21 REAL(kind=wp) :: r2 = .1    ! constante du type reel par default
22 REAL(kind=wp) :: r3, r4
23 r3 = REAL(HUGE(1)-100, kind=wp) ! conversion dans sous-type voulu
24 r4 = REAL(HUGE(1)-100)         ! sans preciser le sous-type => par default
25 WRITE(*,*) "digits=", DIGITS(r1), "epsilon=", EPSILON(r1)
26 WRITE(*,'(f23.21,a)') r1, " = 0.1 représenté à la précision de travail"
27 WRITE(*,'(f23.21,a)') r2, " = 0.1 représenté à la précision par défaut"
28 WRITE(*,'(i10.10, a)') HUGE(1)-100, " = huge(1) - 100 en entier"
29 WRITE(*,'(f12.1,a)') r3, " = huge(1) - 100 en réel à la précision de travail"
30 WRITE(*,'(f12.1,a)') r4, " = huge(1) - 100 en réel à la précision par défaut"
31 END PROGRAM test_precision

```

```

digits= 53 epsilon= 2.220446049250313E-16
0.100000000000000005551 = 0.1 représenté à la précision de travail
0.10000001490116119385 = 0.1 représenté à la précision par défaut
2147483547    = huge(1) - 100 en entier
2147483547.0 = huge(1) - 100 en réel à la précision de travail
2147483520.0 = huge(1) - 100 en réel à la précision par défaut

```

2.6 Fonctions de conversion de type

Fortran possède de nombreuses fonctions de conversion (explicite) entre les types numériques et leurs variantes, fonctions décrites dans l'annexe A.2, p. 127. En particulier, la conversion d'entier en flottant est assurée par `REAL` et celle de flottant en entier par exemple par `INT` qui tronque vers zéro, mais d'autres choix sont possibles : au plus proche avec `NINT`, par excès avec `CEILING`, par défaut avec `FLOOR`.

Ces fonctions de conversion numérique admettent un paramètre optionnel de `KIND` qui permet de préciser la variante de type du résultat.

Chapitre 3

Opérateurs et expressions

3.1 Les opérateurs numériques

Outre l'opérateur unaire - qui calcule l'opposé d'un nombre¹, le fortran dispose des quatre opérateurs binaires d'addition (+), de soustraction (-), de multiplication (*) et de division (/). Ces quatre opérateurs binaires sont prévus pour travailler avec des opérandes de même type (et même sous-type).

Mais le fortran² possède aussi un opérateur d'élevation à la puissance, noté ** et qui est défini différemment suivant la nature de l'exposant :

- si n est entier positif, $a**n$ est défini pour tout a réel ou entier par le produit de n facteurs $a \times a \times \dots \times a$ (l'exposant n'est pas converti en réel) ;
- si n est entier négatif, $a**n$ est défini comme $1/a**(-n)$;
- si x est réel, $a**x$ n'est défini que si a est strictement positif et vaut alors $\exp(x \ln(a))$

♥ ⇒ Ainsi, pour élever un nombre à une puissance entière positive, il faudra éviter d'utiliser un exposant réel, comme par exemple dans $a**4.$, qui forcerait l'utilisation de la deuxième méthode, moins précise numériquement et plus coûteuse.

3.1.1 Règles de priorité entre les opérateurs numériques binaires

Quand l'ordre d'évaluation n'est pas imposé par l'utilisation de parenthèses, les expressions numériques sont évaluées en respectant l'ordre de priorité suivant : (1) ** (2) * et / (3) + et -. À niveau de priorité égal et en l'absence de parenthèses, les évaluations s'effectuent en principe de gauche à droite, sauf pour l'opérateur **, pour lequel l'évaluation se fait de droite à gauche pour respecter l'interprétation classique de la notation mathématique $a^{b^c} = a^{(b^c)}$.

expression	évaluée comme	c'est-à-dire
$a + b ** c / d / e$	$a + (((b**c) / d) / e)$	$a + \frac{b^c/d}{e}$
$a ** b ** c * d + e$	$((a ** (b**c)) * d) + e$	$a^{b^c} d + e$

Mais un compilateur peut s'autoriser de modifier cet ordre, par exemple pour des raisons de rapidité de calcul. Ainsi $a/b/c$ est a priori évalué comme $(a/b)/c$, mais si le compilateur optimise la vitesse de calcul et considère que le processeur est plus rapide pour effectuer une multiplication qu'une division, il pourra demander d'évaluer dans l'ordre $a/(b*c)$.

3.1.2 Imperfections numériques des opérations liées aux types

♥ ⇒ Les opérations associatives algébriquement peuvent s'avérer non associatives sur ordinateur à cause :

- des erreurs d'arrondi sur les réels en précision limitée,

1. + peut aussi être utilisé comme opérateur unaire.
2. Contrairement au langage C, qui doit faire appel à la fonction pow.

– ou des dépassements de capacité sur les réels comme sur les entiers.

Il est donc parfois prudent d'imposer l'ordre d'évaluation par des parenthèses, voire de reformuler les expressions.

Exemple de dépassement de capacité

C'est le cas du calcul naïf de la constante de Stefan $\sigma = \frac{2\pi^5}{15} \frac{k^4}{c^2 h^3}$ avec des réels sur 32 bits. Les calculs de k^4 et de h^3 vont provoquer des dépassements de capacité³ par valeur inférieure (underflow) : numérateur et dénominateur étant alors considérés comme nuls, et leur rapport 0/0 est indéterminé, et σ est affiché sous la forme NaN soit « Not-A-Number ». La factorisation de $(k/h)^3$ dans l'expression de σ limite le domaine des résultats intermédiaires et permet d'éviter le dépassement.

Comment limiter les erreurs d'arrondi

Si par exemple a et b sont de signes opposés et de valeur absolue très proches et très supérieure à c , par prudence, on évitera d'écrire $a + b + c$ qui pourrait laisser le choix au compilateur. On écrira $(a+b) + c$ pour minimiser l'erreur d'arrondi, plus faible qu'avec $a + (b+c)$.

Le choix de l'ordre des opérations peut s'avérer crucial pour minimiser les erreurs d'arrondi, par exemple dans l'évaluation des sommes de séries. Dans ce contexte, on aura toujours intérêt à commencer la sommation en accumulant les termes les plus petits entre eux, c'est-à-dire en commençant par le terme d'indice le plus élevé.

De même, quand on doit parcourir un intervalle à pas constant avec une variable réelle x dans une boucle DO, le calcul par addition qui accumule les erreurs d'arrondi doit être évité au profit du calcul par multiplication. ⇐ ♡

<pre>x = xmin DO i=1, n x = x + dx END DO</pre>	⇒	<pre>DO i=1, n x = xmin + (i-1) * dx END DO</pre>
---	---	---

3.1.3 Conversions implicites

Lors de l'affectation d'une expression numérique d'un type donné à une variable d'un type numérique différent, il y a conversion implicite, avec éventuellement perte de précision (par exemple dans la cas d'une expression entière affectée à une variable réelle), voire dépassement de capacité (comme dans le cas d'une expression réelle affectée à un entier).

```
REAL          :: r = 3.14, s
INTEGER, PARAMETER :: ki=SELECTED_INT_KIND(14) ! variante pour stoker 10**14
INTEGER :: j
INTEGER(KIND=ki) :: i = 1000000000000_ki ! = 10**12
j = r          ! donnera j = 3, comme j = int(r)
s = i          ! donnera s très proche de 1000000000000 à environ 1,2 10**5 près
                ! mais stocké approximativement comme s = real(i)
```

Il en est de même lors de l'évaluation d'expressions dites mixtes impliquant des opérandes de types numériques différents : les opérandes sont alors promus au type le plus riche avant évaluation. La hiérarchie des types numériques comporte, par ordre de richesse croissante, les entiers, suivis des flottants et au plus haut niveau les complexes. Au sein d'un type donné, les variantes de type sont classées selon le nombre d'octets utilisés : par exemple le type DOUBLE PRECISION est plus riche que le type prédéfini REAL.

3. Noter cependant que l'unité de calcul flottant (Floating Processor Unit) effectue bien souvent les calculs dans des registres de 80 voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Le dépassement éventuel peut être mis en évidence en activant une option de compilation (`-ffloat-store` cf. E.5.1 pour `g95` et `gfortran`, `-float-store` cf. E.2.1 pour `nagfor`) interdisant l'usage des registres étendus.

```

COMPLEX      :: u, v
REAL         :: r, s
INTEGER      :: i, j
v = u + r    ! est calculé comme v = u + cplx(r)
v = u + i    ! est calculé comme v = u + cplx(i)
s = r + i    ! est calculé comme s = r + real(i)
!           combinaison des deux conversions implicites
j = u + r    ! est calculé comme j = int(u + cplx(r))

```

♡ ⇒ Pour une bonne lisibilité, on s'efforcera de rendre explicites de telles conversions, en utilisant les fonctions intrinsèques dites de conversion de type⁴ (cf. Annexe A.2, p. 127).

Cas de la division entière

Noter que l'opérateur / appliqué à des entiers permet de calculer le quotient au sens de la division euclidienne : par exemple avec des constantes, 5/3 donne 1, alors que 5./3. donne bien 1.666667. Ne pas croire qu'il suffit de stocker le résultat de la division de deux entiers dans une variable flottante pour imposer une division flottante : c'est seulement le résultat de la division euclidienne qui est ensuite converti en flottant.

```

INTEGER :: i, j
REAL    :: x, y
DOUBLE PRECISION :: t1, t2
i = 3
j = 2
x = i / j
y = REAL(i) / REAL(j)
t1 = 1./3.
t2 = 1./3.d0

```

Dans l'exemple ci-contre, x vaut 1. alors qu'avec la conversion forcée d'au moins un des opérandes en flottant, y vaut 1.5. De même, dans t1, la division est évaluée en simple précision et le résultat converti en double, alors que dans t2, elle est évaluée en double précision.

Noter enfin que l'élévation d'un entier à une puissance entière négative va donner lieu à une division entière : par exemple 10**(-3) sera interprété⁵ comme 1/10**3 ou encore 1/1000 qui est nul. Mais 10.**(-3) donnera bien .001.

3.2 Les opérateurs de comparaison

Fortran possède six opérateurs de comparaison dont le résultat est une variable booléenne. L'ancienne notation du fortran 77 est acceptée.

fortran	fortran77	signification	remarques à propos des notations
<	.LT.	inférieur à	Ne pas confondre avec l'opérateur d'affectation =
<=	.LE.	inférieur ou égal à	
==	.EQ.	égal à	
>=	.GE.	supérieur ou égal à	
>	.GT.	supérieur à	
/=	.NE.	différent de	
			Ne pas confondre avec != du langage C

```

REAL          :: a, b, r
INTEGER       :: i
IF( a > 0 )   b = LOG(a)      ! a > 0 est évalué à .true. si a est positif
IF( i /= 0 ) r = 1. / REAL(i)

```

4. Ces fonctions sont les équivalents de l'opérateur de conversion explicite du langage C ou cast, noté en préfixant l'expression par le type d'arrivée entre parenthèses.

5. En particulier, si la constante réelle $1.5e-3$ vaut $1,5 \times 10^{-3}$, l'expression constante réelle $1.5*10.**(-3)$ a la même valeur alors que l'expression $1.5*10**(-3)$ est nulle.

Étant donné que les nombres flottants ne sont représentés qu'approximativement en machine, les tests d'égalité entre flottants sont déconseillés; on préférera comparer les différences à un seuil, en utilisant la fonction intrinsèque `EPSILON` (cf. annexe A.4, p. 129).

On remplacera par exemple :

```
IF ( a == b ) THEN
```

par :

```
IF ( ABS(a-b) <= c *ABS(a) * EPSILON(a) ) THEN
```

où `c` est une constante supérieure à 1.

3.3 Les opérateurs booléens

Fortran dispose de quatre opérateurs logiques binaires permettant de combiner des expressions logiques entre elles, ainsi que de l'opérateur unaire de négation.

<code>.AND.</code>	ET
<code>.OR.</code>	OU (inclusif)
<code>.NOT.</code>	Négation unaire
<code>.EQV.</code>	Équivalence
<code>.NEQV.</code>	OU exclusif

```
REAL                :: r
LOGICAL             :: grand, petit, moyen
grand = r > 1000.
petit = r < .0001
moyen = .NOT. (grand .OR. petit)
```

L'évaluation d'une expression logique ne nécessite pas forcément celle de toutes ses sous-expressions : en particulier, si le premier opérande d'un `AND` est faux ou si le premier opérande d'un `OR` est vrai, le résultat ne dépend pas du second, mais la norme⁶ ne garantit pas qu'il ne soit pas évalué⁷. Afin d'écrire du code portable, on ne s'appuiera donc pas sur cette propriété pour conditionner un calcul (cf. par exemple 6.5.1, p. 63). ⇐ ♥

3.4 Opérateur de concaténation des chaînes de caractères

L'opérateur `//` permet de concaténer⁸ deux chaînes de caractères.

```
CHARACTER(len=5)    :: jour='mardi'
CHARACTER(len=4)    :: mois='juin'
CHARACTER(len=13)   :: date
date = jour//'-04-'//mois    ! contiendra 'mardi-04-juin'
```

6. Le compilateur `g95` fournit cependant une option `-fshort-circuit` qui assure que le second membre d'un `AND` ou d'un `OR` ne soit évalué que si nécessaire.

7. Le langage C, à l'inverse, assure une évaluation minimale des opérandes du `&&` et du `||`.

8. En langage C, la concaténation de chaînes de caractères est obtenue par simple juxtaposition, sans opérateur explicite.

Chapitre 4

Structures de contrôle

4.1 Structures IF

4.1.1 L'instruction IF

La forme la plus simple de l'exécution conditionnelle est réservée au cas d'une instruction unique.

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

Si l'expression logique est évaluée à `.TRUE.`, l'instruction est exécutée; sinon, le contrôle est passé à l'instruction suivante.

```
INTEGER :: i, j, k  
IF(j /= 0) k = i/j ! k n'est calculé que si j est différent de 0
```

4.1.2 La structure IF ... END IF

Pour conditionner l'exécution d'un bloc d'instructions à une condition, on emploie la structure :

```
[<nom> :] IF (<expression logique>) THEN  
  <bloc d'instructions> ! exécuté si la condition est vraie  
END IF [<nom>]
```

où `<nom>` est un identificateur optionnel permettant d'étiqueter le test.

```
INTEGER :: i  
...  
chiffre: IF (i < 10) THEN  
  WRITE(*,*) 'i est inférieur à 10'  
END IF chiffre
```

Cette structure peut comporter un bloc `ELSE` qui n'est exécuté que si la condition est évaluée à `.FALSE.`¹.

```
[<nom> :] IF (<expression logique>) THEN  
  <bloc d'instructions> ! exécuté si la condition est vraie  
ELSE [<nom>]
```

1. Si l'identificateur est spécifié en tête de la structure `IF`, il devient nécessaire à la fin du `END IF`, mais pas forcément après le `ELSE`.


```

    <bloc d'instructions> ! exécuté si la condition est fausse
  END IF [<nom>]

```

Si un branchement est possible de l'intérieur de la structure vers l'extérieur, aucun accès vers cette structure n'est autorisé sans passage par le test initial. Ces structures peuvent être imbriquées de façon complexe, bien que, parfois, le recours à un `SELECT CASE` s'avère plus efficace et plus lisible.

```

INTEGER :: i
...
chiffre: IF (i < 10) THEN
  WRITE(*,*) 'i est inférieur à 10'
ELSE chiffre
  WRITE(*,*) 'i est supérieur ou égal à 10'
END IF chiffre

```

4.1.3 Utilisation du ELSE IF

Dans le cas où plusieurs tests imbriqués sont effectués,

```

REAL :: a, b
IF ( a <= 0. ) THEN
  ...
ELSE
  IF ( b <= 0. ) THEN
    ...
  ELSE
    ...
  END IF
END IF

```

le test `ELSE IF` permet de « raccourcir » l'écriture. Noter qu'alors un seul `END IF` ferme la structure globale.

```

REAL :: a, b
IF ( a <= 0. ) THEN
  ...
ELSE IF ( b <= 0. ) THEN
  ...
ELSE
  ...
END IF

```

Sous cette dernière forme (non-imbriquée), il est possible d'ajouter autant de clauses `ELSE IF(. . .) THEN` que nécessaire et la dernière clause `ELSE` n'est pas obligatoire.

4.2 Structure SELECT CASE

Lorsque les choix possibles sont nombreux, et portent sur une même expression, la structure `SELECT CASE` est souvent plus efficace que son équivalent à l'aide de `IF` imbriqués. Elle permet, au vu de la valeur d'une expression, de choisir un traitement parmi un ensemble de cas qui doivent être énumérés : l'expression à tester doit faire l'objet d'un test d'égalité ; c'est pourquoi cette structure ne peut s'appliquer qu'à des types possédant une représentation exacte, à l'exclusion des réels.

```
[<nom> :] SELECT CASE (<expression>)
  CASE (<sélecteur_1>) [<nom>]
    <bloc d'instructions>
  CASE (<sélecteur_2>) [<nom>]
    <bloc d'instructions>
  ...
  CASE (<sélecteur_n>) [<nom>]
    <bloc d'instructions>
  CASE DEFAULT [<nom>]
    <bloc d'instructions>
END SELECT [<nom>]
```

où *<expression>* est une expression de type entier, booléen ou chaîne de caractères, et *<sélecteur_i>* une constante ou une liste de constantes séparées par des virgules, un élément de la liste pouvant être constitué d'un intervalle de valeurs spécifié selon la syntaxe suivante :

<borne_inférieure>:<borne_supérieure>,

l'une ou l'autre des bornes pouvant être omise², ce qui permet de représenter un nombre infini de valeurs. Le cas DEFAULT concerne les expressions qui n'ont coïncidé avec aucun des sélecteurs.

```
INTEGER :: i
tri: SELECT CASE (i)
  CASE(0) tri
    WRITE(*,*) ' i = 0'
  CASE(1,-1) tri
    WRITE(*,*) ' i = 1 ou i = -1'
  CASE(2:10) tri
    WRITE(*,*) ' 2 <= i <= 10'
  CASE(11:) tri
    WRITE(*,*) ' i >= 11'
  CASE DEFAULT tri
    WRITE(*,*) 'i < -1'
END SELECT tri
```

```
CHARACTER(len=3) :: rep
valid: SELECT CASE (rep)
  CASE('oui','OUI') valid
    WRITE(*,*) ' oui'
  CASE('non','NON') valid
    WRITE(*,*) ' non'
  CASE DEFAULT valid
    WRITE(*,*) 'réponse invalide'
END SELECT valid
```

4.3 Structures de boucles

Pour effectuer des itérations, le fortran dispose des boucles DO avec ou sans compteur, et de la boucle DO WHILE. Mais, lorsqu'il s'agit de conditionner le traitement des éléments d'un tableau par

2. Comme dans les shells de type `sh` d'UNIX, mais contrairement au comportement du `switch` en langage C et des shells de type `csh`, dès qu'un choix a été sélectionné (et donc un bloc d'instructions exécuté), le contrôle est passé à la fin de la structure ; cela équivaudrait en C à faire suivre chaque bloc par l'instruction `break` (ou `breaksw` en `csh`).

un test portant sur un tableau conformant (cf. 7.1.1), l'utilisation de l'instruction `WHERE`, ou de la structure `WHERE ... END WHERE` (cf. 7.3.2, p. 75) s'avère plus efficace.

4.3.1 Boucles DO avec compteur

```
[<nom> :] DO <entier>=<debut>,<fin> [,<pas>]
  <bloc d'instructions>
END DO [<nom>]
```

où `<debut>`, `<fin>` et `<pas>` sont des expressions *entières*³. La valeur du pas d'incréméntation du compteur est 1 par défaut, mais peut être négative. Pour certaines combinaisons des paramètres, il est possible que le bloc ne soit exécuté aucune fois. Il est interdit de modifier la valeur du compteur au sein de la boucle ; sa valeur à la fin de la boucle est la première valeur qui ne respecte pas la condition de fin de boucle, sauf en cas de sortie par `EXIT`.

```
INTEGER :: i, imax = 10, n
n = 0
impairs: DO i = 1, imax, 2      ! donc i = 1, 3, 5, 7, 9
  n = n + i
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

4.3.2 Boucles DO WHILE

Si le nombre de passages n'est pas connu avant l'entrée dans la boucle, elle doit être contrôlée non par un compteur mais par un test logique : c'est l'objet de la boucle `WHILE` qui intervient de façon classique dans les méthodes itératives. Par prudence, on comptera malgré tout le nombre de passages pour déclencher une sortie par `EXIT` (cf. 4.3.3) si le nombre de passages dépasse une limite raisonnable. ⇐ ♥

```
[<nom> :] DO WHILE (<expression_logique>)
  <bloc d'instructions>
END DO [<nom>]
```

Tant que l'expression logique est vraie, le bloc d'instruction est exécuté. Pour que la boucle ait un intérêt, il faut que le bloc d'instruction modifie la valeur de l'expression logique à un moment donné.

```
INTEGER :: i, imax = 10, n
n = 0
i = 1
impairs: DO WHILE ( i <= imax )
  n = n + i
  i = i + 2 ! modification de l'expression qui intervient dans le test
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

4.3.3 Ruptures de séquence dans les boucles : EXIT et CYCLE

Les ruptures de séquence dans les boucles sont possibles grâce aux branchements `EXIT` et `CYCLE`⁴.

3. La possibilité d'utiliser un indice de boucle réel a disparu de la norme lors du passage au fortran 90.

4. Ces instructions correspondent respectivement à `break` et `continue` en langage C.

Branchement à l'indice suivant (éventuel) : CYCLE

L'instruction `CYCLE` à l'intérieur d'une structure `DO ... END DO` permet de court-circuiter la fin du bloc et de passer le contrôle à l'indice de boucle suivant.

```

INTEGER :: i, imax = 10, n
n = 0
impairs : DO i = 1, imax                ! a priori 1,2,3,4, ... 10
  IF ( modulo(i,2) == 0 ) CYCLE ! ne pas considérer les entiers pairs
  n = n + i
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n

```

En cas de boucles imbriquées, l'instruction `CYCLE` concerne a priori la boucle *la plus interne* dans laquelle elle est située. Pour qu'elle puisse s'adresser à une boucle externe, il est nécessaire de nommer la boucle et d'utiliser son nom comme argument de `CYCLE`⁵.

```

INTEGER :: i, imax, j, jmax
externe: DO i = 1, imax
  ...
  interne: DO j = 1, jmax
    ...
    IF (<expr_log>) CYCLE externe ! fait passer au i suivant
    ...
  END DO interne
  ...
END DO externe

```

Sortie de boucle : EXIT

L'instruction `EXIT` à l'intérieur d'une structure `DO ... END DO` permet de sortir immédiatement de la boucle et de passer le contrôle à l'instruction qui suit le `END DO`.

```

INTEGER :: i, n
n = 0
DO i = 1, 1000, 2          ! a priori de 1 à 999
  IF ( i > 10 ) EXIT      ! limitation à 10 (un seul EXIT exécuté)
  n = n + i
END DO
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n

```

Dans le cas de boucles imbriquées, comme pour l'instruction `CYCLE`, l'instruction `EXIT` ne sort que de la boucle *la plus interne* dans laquelle elle est située. Pour pouvoir sortir directement d'une boucle externe, il faut nommer la-dite boucle et utiliser la syntaxe `EXIT nom`.

4.3.4 Boucles DO sans compteur

```

[<nom> :] DO
  <bloc d'instructions>
END DO [<nom>]

```

5. Ces possibilités de branchement s'adressant à une boucle externe nommée suivant la même syntaxe qu'en fortran existent aussi en java pour les branchements `break` et `continue` concernant des structures de contrôle itératives comme `while` ou `for`.

Cette structure de boucle, a priori infinie, n'a d'intérêt que si le bloc comporte une instruction EXIT permettant de sortir de la boucle au vu d'un test IF⁶.

```

INTEGER :: i
DO
  WRITE(*,*) 'entrer un entier positif, sinon 0 pour terminer'
  READ(*,*) i
  IF ( i == 0 ) EXIT
  ...                ! bloc d'instructions
END DO

```

4.4 Autres instructions de contrôle

4.4.1 Instruction CONTINUE

L'instruction *vide* CONTINUE, très utilisée pour délimiter les fins de boucles dans les anciennes versions du fortran, voit, avec l'apparition du END DO, son emploi réduit aux rares branchements spécifiés par des étiquettes numériques (*cf.* 4.4.2, p. 27 et 4.4.6, p. 28).

4.4.2 Branchement par GO TO

L'instruction GOTO <étiquette_numérique> permet le branchement vers l'instruction repérée par l'étiquette numérique. L'étiquette numérique est un nombre entier positif qui doit être placé en début de ligne (premier caractère non blanc) et séparée d'au moins un espace de l'instruction sur laquelle se fait le branchement. Inévitable dans les versions anciennes du fortran, l'usage du GOTO est fortement déconseillé pour la lisibilité des programmes, et doit être réservé au traitement des circonstances exceptionnelles, quand elles ne peuvent être gérées via CYCLE ou EXIT.

4.4.3 Instruction STOP

L'instruction STOP [<code d'arrêt>] impose l'arrêt immédiat du programme et l'affichage éventuel du code d'arrêt (une chaîne de caractères ou un entier d'au plus 5 chiffres) qui peut être lui-même repris par certains systèmes d'exploitation. Elle est utilisée pour interrompre l'exécution en fin de programme principal ou à un endroit quelconque depuis une procédure quelconque dans des circonstances exceptionnelles ne permettant plus de poursuivre le traitement.

```

PROGRAM trait
  ...
  CALL sub_prog
  ...
  IF ( ... ) STOP 10    ! interruption exceptionnelle
  ...
  STOP                 ! fin normale
END PROGRAM trait
SUBROUTINE sub_prog
  ...
  IF ( ... ) STOP 12   ! arret exceptionnel
  ...
  RETURN              ! retour normal au programme principal
END SUBROUTINE sub_prog

```

6. À condition de placer le test pour la sortie de la boucle en tête de bloc, cette structure permet de remplacer la structure DO WHILE ... END DO, déconseillée car mal adaptée au calcul parallèle.

4.4.4 Instruction RETURN

L'instruction RETURN transfère le contrôle de la procédure (sous-programme ou fonction) où elle est située à la procédure l'ayant appelée⁷. Elle n'est pas autorisée dans le programme principal. Elle permet de ménager plusieurs retours de contrôle vers le programme appelant, dans le cas de traitements conditionnels.

L'exemple élémentaire qui suit n'est cependant pas un modèle de programmation structurée et de lisibilité.

```
PROGRAM trait
  ...
  y1 = rac_cub(x1)
  y2 = rac_cub(x2)
  ...
END PROGRAM trait
! une fonction racine cubique étendue à tous les réels
FUNCTION rac_cub(x)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x
  REAL              :: rac_cub
  rac_cub = 0.      ! cas où x=0
  IF ( x == 0. ) RETURN ! retour immédiat au programme principal
  ! la suite n'est vue que pour x non nul
  rac_cub = ABS(x) ** (1./3.) ! calcul de la valeur absolue
  IF ( x > 0 ) RETURN ! retour immédiat si x>0
  rac_cub = - rac_cub ! cas x<0 : il faut encore changer le signe
  RETURN ! retour au programme principal
END FUNCTION rac_cub
```

4.4.5 Remarques sur les instructions STOP et RETURN

En fortran 77, l'instruction STOP dans le programme principal, de même que l'instruction RETURN dans les sous-programmes et fonctions, était obligatoire juste avant END, quand END n'était qu'un délimiteur. En fortran 90, END est aussi un ordre exécutable et STOP en fin de programme principal ainsi que RETURN en fin de sous-programme ou de fonction sont donc devenus facultatifs.

4.4.6 Branchements dans les entrées-sorties

Certains ordres d'entrée-sorties (*cf.* 5.3, p. 36), notamment OPEN, READ, WRITE⁸ possèdent des arguments optionnels (accessibles par mot-clef) de branchement à des instructions étiquetées permettant de traiter des circonstances particulières qu'il est toujours prudent de prévoir :

- en cas d'erreur : ERR=<étiquette_numérique> (OPEN, READ, WRITE);
- en cas de fin de fichier : END=<étiquette_numérique> (READ);
- en cas de fin d'enregistrement : EOR=<étiquette_numérique> (READ).

```
...
OPEN(...)
READ(..., ERR=900, ...)
... ! traitement normal
STOP ! arrêt normal
900 CONTINUE ! branchement en cas d'erreur de lecture
WRITE(*,*) 'erreur de lecture'
STOP 12 ! arrêt exceptionnel après erreur de lecture
END PROGRAM ...
```

7. Ne pas confondre RETURN, qui rend le contrôle à l'appelant, et STOP, qui interrompt l'exécution.

8. BACKSPACE, CLOSE, ENDFILE, INQUIRE et REWIND possèdent aussi un argument optionnel de mot-clef ERR.

Pour éviter les branchements sur des étiquettes numériques, on préférera tester si le code de retour `IOSTAT` des opérations d'entrées-sorties est non nul pour traiter ces cas particuliers à l'aide des structures de contrôle classiques. Avec fortran 2003, les constantes `IOSTAT_END` et `IOSTAT_EOR` du module `ISO_FORTRAN_ENV` permettent de distinguer dans ces cas les fins de fichier et d'enregistrement (cf. 5.3.1 p. 36). On peut aussi appeler une des fonctions booléennes `IS_IOSTAT_END` ou `IS_IOSTAT_EOR`. ⇐ ♡
⇐ f2003

Chapitre 5

Entrées–sorties, fichiers

Fortran dispose d'outils sophistiqués pour réaliser des transferts d'informations avec des fichiers, mais il n'est pas nécessaire de connaître toutes leurs possibilités pour mettre en œuvre des programmes élémentaires. En sortie comme en entrée, la mise en forme des données peut en effet être effectuée avec d'autres outils (filtres sous unix par exemple) plus adaptés à la manipulation des chaînes de caractères dans ces étapes de pré- ou post-traitement qui n'impliquent pas de calculs lourds. C'est pourquoi, nous aborderons d'abord (5.1) les entrées-sorties standard (interactions avec le terminal) avant de présenter les concepts généraux (5.2) et la syntaxe détaillée (5.3) des instructions d'entrées-sorties. L'étude des nombreux exemples présentés dans les dernières sections 5.5 et 5.6 du chapitre devrait éclairer les notions introduites dans les sections 5.2 et 5.3.

5.1 Introduction : entrées et sorties standard

Nous avons déjà utilisé, sans les présenter explicitement, les instructions d'affichage (PRINT) et de lecture (READ) au *format libre*. Ils respectent la syntaxe suivante :

```
PRINT *, <liste d'expressions>  
READ *, <liste de variables>
```

où * représente le format libre et où les éléments de la liste d'entrée-sortie sont séparés par des virgules.

```
INTEGER :: i, j  
REAL :: a  
PRINT *, 'donner deux entiers et un réel'      ! affichage au format libre  
READ *, i, j, a                               ! lecture au format libre  
PRINT *, 'vous avez choisi i = ', i, ' j = ', j, ' a = ', a
```

Dans l'exemple précédent, la liste qui suit PRINT comporte seulement des chaînes de caractères constantes et des variables. Mais elle peut aussi comporter des expressions :

```
PRINT *, 'a = ', a, ' a*a = ', a**2, ' sin(a) = ', sin(a)
```

5.1.1 Une instruction d'E/S = un enregistrement

La principale règle régissant la progression par défaut des lignes dans les entrées-sorties veut que :

```
Chaque instruction de lecture ou d'écriture provoque le passage à la ligne suivante.
```


En particulier, si la liste est vide, en écriture l'instruction `PRINT *` produit un saut de ligne, de même qu'en lecture l'instruction `READ *` saute une ligne d'entrées qui reste inexploitée.

Cette règle, essentielle pour comprendre en particulier écriture et lecture des tableaux implique que, si on utilise une boucle explicite pour afficher les éléments d'un tableau, on affiche un élément par ligne (affichage en colonne). Pour afficher le tableau en une seule ligne, il faut (cf. 5.4.4, p. 45) :

- soit, comme en fortran 77, utiliser une boucle implicite, qui produit « au vol » la liste des éléments pour l'instruction d'écriture,
- soit, de façon plus concise, utiliser la notation globale pour l'ensemble des éléments du tableau à afficher.

5.1.2 Saisie de données au clavier

Les données doivent être écrites au clavier sous une des formes admissibles pour les constantes (cf. 2.4, p. 14) du type de la variable à affecter. À la conversion implicite près des entiers en réels, une différence de type provoque une erreur d'exécution lors de la lecture.

Lors de la lecture d'un enregistrement, une combinaison quelconque des saisies suivantes au clavier :

- un ou plusieurs espaces ;
- une virgule¹ ;
- un ou plusieurs changements de ligne,

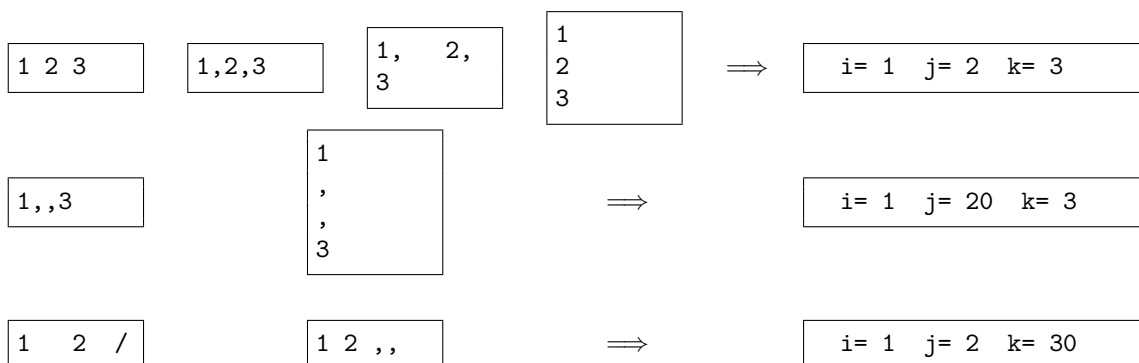
est considérée comme un séparateur de données, permettant de passer à la lecture de la variable suivante.

Au lieu de répéter $\langle n \rangle$ fois une valeur $\langle val \rangle$, on peut utiliser la notation $\langle n \rangle * \langle val \rangle$. Tant que la liste des variables² n'est pas épuisée, la lecture des données peut se poursuivre sur plusieurs lignes, donc la lecture d'un enregistrement peut correspondre à la saisie de plusieurs lignes.

Mais on peut arrêter la lecture de l'enregistrement avant d'avoir épuisé la liste par la saisie du caractère / de fin d'enregistrement. Il est aussi possible de « sauter » une variable au sein de la liste en saisissant deux virgules successivement. Dans ces deux cas, les variables non lues gardent leur valeur antérieure.

Par exemple, pour les jeux d'entrées indiqués à gauche de la flèche (\Rightarrow), le programme suivant affichera ce qui est indiqué à droite.

```
PROGRAM lecture
IMPLICIT NONE
INTEGER :: i=10, j=20, k=30      ! 3 entiers initialisés
PRINT *, 'donner 3 entiers'
READ *, i, j, k
PRINT *, ' i=', i, ' j=', j, ' k=', k
END PROGRAM lecture
```



1. Le séparateur de données virgule « , » est changé en point-virgule « ; » dans le cas où on a choisi la virgule comme séparateur décimal (cf. 5.4.2, p. 44).

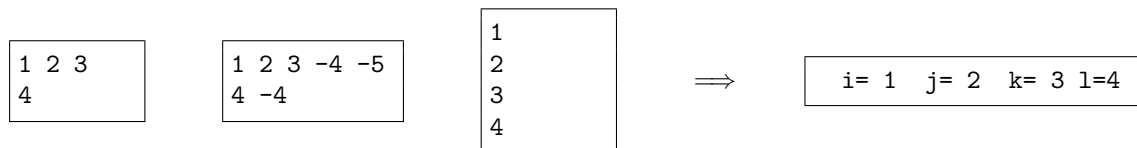
2. Au sens scalaire du terme, donc par exemple chacun des éléments d'un tableau de réels.



Si on fournit plus de données que nécessaire, les données superflues sont perdues : ne pas croire qu'elles sont disponibles pour l'instruction de lecture suivante.

Le programme suivant comporte deux instructions de lecture, et ne peut lire l'entier 1 qu'après un changement d'enregistrement.

```
PROGRAM lecture2
IMPLICIT NONE
INTEGER :: i=10, j=20, k=30, l=40      ! 4 entiers initialisés
PRINT *, 'donner 3 entiers puis 1 entier'
READ *, i, j, k
READ *, l
PRINT *, ' i=', i, ' j=', j, ' k=', k, ' l=', l
END PROGRAM lecture2
```



Dans les applications élémentaires et en phase de mise au point, on se contente souvent du format libre pour les entrées et les sorties avec le terminal. S'il est parfois nécessaire de contraindre le format en sortie pour améliorer la lisibilité, même à l'écran, à l'inverse, pour la lecture au clavier, il est conseillé d'utiliser le format libre, sauf à informer explicitement l'utilisateur des contraintes du format de saisie. En revanche, lorsque la quantité des données d'entrée devient importante, il est souvent plus efficace de les saisir dans un fichier avec toutes les possibilités d'un éditeur, puis de faire lire ce fichier par le programme.

5.2 Généralités et terminologie

Dans les échanges de données plus massifs qu'avec le terminal, ou avec des fichiers dont le format peut être imposé par d'autres applications, il est souvent nécessaire de préciser les règles de conversion en entrée ou en sortie grâce à une spécification de format.

```
INTEGER :: k
INTEGER, DIMENSION(100) :: ti ! tableau de 100 entiers
REAL, DIMENSION(100) :: tr ! tableau de 100 réels
OPEN(9, FILE='donnees.dat') ! ouverture du fichier => connecté à l'unité 9
DO k = 1, 100
  READ(9, '(i4, f9.5)') ti(k), tr(k) ! lecture d'un enregistrement
END DO
CLOSE(9) ! fermeture du fichier
```

Dans cet exemple, chaque instruction `READ` lit dans le fichier `donnees.dat` un enregistrement constitué d'un entier codé sur 4 chiffres, suivi d'un réel sur 9 caractères dont 5 après le point décimal. La chaîne de caractères `i4, f9.5` constitue le format de lecture. L'objectif de ce chapitre est de décrire les différentes instructions permettant les échanges de données et d'introduire la syntaxe des spécificateurs de format.

5.2.1 Fichiers externes et fichiers internes

Fichiers externes

Outre les entrées-sorties standard qui s'opèrent via le terminal, fortran peut contrôler plusieurs flux de données en lecture ou en écriture associés à des *fichiers externes* (external files), hébergés

par des périphériques (disquettes, disques, bandes magnétiques, CD, ...).

Ces fichiers externes doivent être connectés à des *unités logiques* (logical unit) à l'aide de l'instruction `OPEN`. L'instruction `CLOSE` permet de fermer le fichier et donc de libérer l'unité logique préalablement connectée. Une unité logique est désignée par un entier positif ou nul.

L'entier désignant l'unité logique peut être choisi entre 0 et une limite supérieure dépendant du processeur^{3, 4, 5}.

Unités logiques standard

Mais les unités logiques standard d'entrée et de sortie sont préconnectées par défaut, et peuvent être désignées par le symbole `*`.

Sous UNIX, l'entrée standard (clavier) et la sortie standard (écran) sont traditionnellement préconnectées respectivement aux unités logiques 5 et 6. La sortie d'erreur standard est préconnectée par défaut à l'unité logique 0; pour écrire sur la sortie d'erreur, il suffirait donc de préciser le numéro d'unité logique 0.

Mais depuis fortran 2003, le module intrinsèque `ISO_FORTRAN_ENV` (cf. 5.3.1, p. 36) définit plusieurs constantes nommées caractérisant l'environnement, dont les numéros des unités logiques standard : `input_unit`, `output_unit` et `error_unit`, qui permettent donc de désigner ces unités logiques de façon portable. ← f2003

Fichiers internes

À l'opposé des fichiers externes stockés sur un dispositif périphérique, on parle aussi d'écriture et de lecture sur des *fichiers internes* (internal files) cf. 8.4.3 p. 89 lorsque l'on code des données numériques sous forme de chaînes de caractères (écriture) ou inversement que l'on décode des chaînes de caractères représentant des données (lecture). Les fichiers internes ne sont que de simples variables de type chaîne de caractères, explicitement déclarées dans le programme et stockées en mémoire vive. Ils sont notamment utilisés pour générer des formats variables pour des opérations d'entrées-sorties (cf. 5.4.5, p. 45).

5.2.2 Notion d'enregistrement

Un *enregistrement* (record) est la généralisation à un fichier de la notion de ligne sur un terminal. Un fichier est une suite d'enregistrements.

Lors d'une opération d'entrée-sortie sur un fichier en accès séquentiel, chaque instruction de lecture ou d'écriture provoque par défaut⁶ le passage à l'enregistrement suivant.

5.2.3 Fichiers formatés et fichiers non formatés

Les données numériques peuvent être stockées dans des fichiers :

- soit sous la forme de chaînes de caractères (qui seront bien sûr eux-mêmes codés en binaire) : les fichiers sont dits alors *formatés* ou codés (*formatted*). Les entrées-sorties standard s'effectuent sous forme codée.
- soit directement sous la forme binaire utilisée pour le stockage dans la mémoire de la machine : les fichiers sont dits alors *non-formatés* (*unformatted*).

Ainsi, lors des entrées-sorties formatées de données numériques sur des fichiers externes, les instructions d'écriture ou de lecture déclenchent en réalité successivement deux opérations : une conversion⁷ selon les règles définies par un *format* et un transfert de données vers un périphérique. Plus précisément, suivant qu'il s'agisse d'entrée ou de sortie formatée, elle signifiera :

3. La limite supérieure est bien souvent celle des entiers par défaut : soit $2^{31} - 1$ sur processeur 32 bits et $2^{63} - 1$ sur processeur 64 bits avec le compilateur `g95`; mais elle reste de $2^{31} - 1$ avec le compilateur `XLF` d'IBM.

4. Certaines bibliothèques se réservent l'usage d'une plage *fixe* de numéros d'unités logiques, qui ne sont plus, en conséquence, disponibles pour les programmes de l'utilisateur de ses bibliothèques.

5. La lecture ou l'écriture sur une unité logique *n* non connectée force la connexion implicite à un fichier externe dont le nom par défaut est `fort.n` sous UNIX.

6. Il est possible, en fortran 90, de faire exception à cette règle grâce à l'option `ADVANCE=no`.

7. Dans le cas d'un fichier interne, c'est la seule opération déclenchée par l'instruction d'entrée-sortie.

- soit d’abord une lecture de chaînes de caractères, puis une conversion en représentation interne ;
- soit d’abord une conversion de la représentation interne en chaînes de caractères, puis une écriture de chaînes de caractères.

Si la conversion provoque une erreur à l’exécution, c’est l’ensemble de l’instruction qui échoue, ce qui s’avère particulièrement gênant par exemple pour les entrées interactives, en cas de faute de frappe. Pour fiabiliser les saisies au clavier, on est amené à exploiter les codes de retour de `READ`⁸. La lecture s’effectue alors dans une boucle qui ne s’arrête que lorsqu’il n’y a plus d’erreur.

```

1 PROGRAM robust_lect
2 IMPLICIT NONE
3 INTEGER :: i=-100, ok
4 WRITE(*,*) 'entrer un entier'
5 DO
6     READ(*, *, iostat=ok) i
7     IF(ok==0) EXIT
8     WRITE(*,*) 'erreur: recommencez'
9 END DO
10 WRITE(*,*) 'i=', i
11 END PROGRAM robust_lect

```

Choix entre fichier formaté ou non-formaté

Le choix entre les deux types de fichiers pour stocker des données numériques est dicté par les critères suivants :

non-formatés : pas de conversion, donc transfert plus rapide et sans perte de précision, fichiers plus compacts ; mais fichiers non portables⁹ et difficiles à lire par d’autres outils¹⁰ ; en particulier, dans les fichiers binaires en accès séquentiel, fortran écrit et s’attend à lire, en tête et en fin (c’est moins gênant) de chaque enregistrement, une balise d’enregistrement (record marker), sous forme d’un entier¹¹ indiquant le nombre d’octets de l’enregistrement : il faut en tenir compte si on accède au fichier au travers de programmes utilisant d’autres langages ;

formatés : lisibilité par d’autres outils (filtres ou simples éditeurs de texte), portabilité entre applications et entre machines ; mais fichiers plus volumineux et accès plus lent ; perte de précision possible.

5.2.4 Méthodes d’accès aux données

On distingue deux méthodes d’accès aux enregistrements stockés dans des fichiers :

8. Cela impose d’utiliser la deuxième syntaxe, plus complète, de `READ`, cf. 5.3.4, p. 39.

9. Les données numériques sur plusieurs octets peuvent être enregistrées en mémoire de deux façons, selon l’ordre choisi entre les octets qui représentent la valeur en binaire :

- dans l’ordre des poids décroissants (octets de poids fort au début) : on parle d’orientation **big-endian** ou gros-boutiste.
- dans l’ordre des poids croissants (octets de poids faible au début) : on parle d’orientation **little-endian** ou petit-boutiste.

Si la majorité les processeurs de PC, en particulier les x86 ont adopté l’orientation **litte-endian**, les processeurs de serveurs IBM , les POWERPC et les IA64 notamment sont d’orientation **big-endian**. On notera que ces différences peuvent concerner les entiers, mais aussi les flottants, car la norme IEEE 754 (cf. annexe C, page 140) ne spécifie pas l’orientation des octets, et les caractères multi-octets d’unicode (UTF-16 et UTF-32).

10. Sous UNIX, la commande `od` (octal dump) permet d’afficher sous diverses formes le contenu binaire d’un fichier. Elle permet entre autres d’afficher un aperçu du contenu de fichiers binaires grâce à une ou plusieurs options `-t` qui spécifient comment interpréter les octets (nombre d’octets à grouper et conversion suivant un type), par exemple :

```

od -t d4 pour des entiers en décimal sur 4 octets (32 bits) ;
od -t f4 ou od -t fF pour des réels sur 4 octets (32 bits) ;
od -t f8 ou od -t fD pour des réels en double précision sur 8 octets (64 bits).

```

11. Cet entier peut être codé sur 64 bits sur les processeurs 64 bits.

- l'accès *séquentiel* (sequential), où les enregistrements sont stockés les uns à la suite des autres¹² et où, pour accéder à un enregistrement particulier, il est nécessaire de *lire* tous les enregistrements précédents ; dans ce mode, il n'est pas possible de modifier un enregistrement sans compromettre l'intégrité de tous les suivants ;
- l'accès *direct* (direct), où les enregistrements sont repérés par un numéro qui permet (moyennant la connaissance de la taille, nécessairement fixe, des enregistrements) de calculer la position de l'enregistrement dans le fichier et d'accéder ainsi à un enregistrement particulier (y compris en écriture) sans lire explicitement ceux qui le précèdent, ni affecter les suivants (cf. 5.6, p. 48).

Le fortran 2003 a introduit une troisième méthode inspirée du langage C, l'accès *stream* ou *flot*, ⇐ f2003 combinant des avantages des deux précédents. Le positionnement dans le fichier se fait en spécifiant le numéro d'unités de stockage (en général des octets) en commençant à 1 en début de fichier. Les fichiers formatés en accès stream ont aussi une structure d'enregistrement, mais les fins d'enregistrement doivent être explicitement spécifiées en écrivant le caractère fourni par la fonction intrinsèque `NEW_LINE`¹³.

Choix entre accès direct et accès séquentiel

Le choix entre les méthodes d'accès aux données dépend de leur nature (qui peut déterminer la taille des enregistrements), de leur volume (stockable facilement ou non dans des tableaux en mémoire) et de l'usage qu'on leur prévoit. Ce choix est parfois imposé par les contraintes suivantes :

- les enregistrements d'un fichier à accès direct sont nécessairement de taille fixe ;
- seul l'accès direct permet d'écrire un enregistrement sans avoir à réécrire la totalité du fichier.

Ainsi, les fichiers de bases de données, dont les enregistrements doivent pouvoir être mis à jour de façon individuelle, et dont le volume ne permet pas un stockage facile en mémoire vive, sont stockés dans des fichiers à accès direct.

Parfois, alors que les deux méthodes sont envisageables, des considérations de performances (même en lecture seule) peuvent amener à préférer l'accès direct, si par exemple on doit très souvent accéder à des données non contiguës, de façon aléatoire. Supposons par exemple qu'une fonction soit échantillonnée dans le plan au sein d'un fichier selon une grille en coordonnées rectangulaires et que l'on souhaite la rééchantillonner selon des coordonnées polaires ; lorsque l'on calculera les valeurs de la fonction en suivant un cercle, on devra accéder à des données certainement non contiguës dans le fichier initial, qu'il sera donc préférable de lire en accès direct ; il sera donc préférable de créer le fichier initial en accès direct.

5.2.5 Notion de liste d'entrée-sortie

Dans une instruction d'entrée-sortie, les informations échangées sont spécifiées par une *liste d'entrée-sortie* (i/o-list), qui est une liste d'expressions pour une instruction d'écriture, mais doit être une liste de variables pour une instruction de lecture. Si un élément de la liste est un tableau, il est équivalent à la liste des éléments du tableau¹⁴ ; s'il s'agit d'une structure, elle doit être interprétée comme la liste des champs de la structure. Enfin, la liste d'entrée-sortie peut comporter des *boucles implicites* (aussi utilisées comme constructeurs de tableau, cf. 7.1.3, p. 72), éventuellement imbriquées.

Dans l'exemple suivant, chacune des trois instructions affiche le tableau `t` de la même façon sur une ligne.

```
INTEGER, DIMENSION(3) :: t = (/ 1, 2, 3 /)      ! tableau initialisé
INTEGER :: i
WRITE(*, '(3i4)') t                            ! affichage global
```

12. Comme sur une bande magnétique qui est un support physique à accès séquentiel.

13. La fonction intrinsèque `NEW_LINE` prend pour argument un caractère quelconque et rend le caractère de fin de ligne de même sous-type. C'est l'équivalent du `\n` du langage C.

14. Le tableau est transformé en liste en faisant varier d'abord le premier indice, puis le deuxième, ... (cf. 7.1.2, p. 71).

```
WRITE(*, '(3i4)') t(1), t(2), t(3)           ! liste exhaustive
WRITE(*, '(3i4)') (t(i), i=1, 3)           ! boucle implicite
```

5.3 Instructions d'entrées-sorties

Une opération de transfert de données requiert en général une instruction OPEN de connexion entre unité logique et fichier, des instructions d'échange READ ou WRITE, et enfin une instruction CLOSE de libération de l'unité logique. Les instructions d'entrées-sorties OPEN, READ, WRITE, PRINT et INQUIRE se partagent des arguments obligatoires ou optionnels accessibles par mot-clef dont on va décrire les plus usités.

f95/2003 5.3.1 Le module intrinsèque ISO_FORTRAN_ENV

Depuis fortran 2003, le module intrinsèque ISO_FORTRAN_ENV définit des constantes nommées qui permettent de désigner de façon portable des paramètres concernant les entrées sorties, notamment :

- Les numéros des unités logiques préconnectées aux flux standard :
 - INPUT_UNIT, qui désigne l'entrée standard (descripteur 0 sous unix permettant des redirections d'entrée via < fichier);
 - OUTPUT_UNIT, qui désigne la sortie standard (descripteur 1 sous unix permettant des redirections de sortie via > fichier);
 - ERROR_UNIT, qui désigne la sortie d'erreur standard (descripteur 2 sous unix permettant des redirections d'erreur standard via 2> fichier).

Si les unités d'entrée et de sortie standard peuvent être désignées par * de façon plus rapide que INPUT_UNIT et OUTPUT_UNIT, l'utilisation de ERROR_UNIT est nécessaire pour désigner la sortie d'erreur de façon indépendante du compilateur.

- Les entiers rendus par IOSTAT= dans les cas d'échec d'opérations d'entrées/sorties suivants :
 - IOSTAT_END qui désigne la fin de fichier ;
 - IOSTAT_EOR qui désigne la fin d'enregistrement (End Of Record) ;

La valeur du code de retour IOSTAT peut être comparée à ces paramètres pour effectuer des traitements spécifiques au lieu d'avoir recours aux arguments optionnels END= ou EOR= (cf. 4.4.6 p. 28), qui, eux, nécessitent le branchement sur une étiquette numérique.

- FILE_STORAGE_UNIT qui désigne l'unité de mesure en bits (en général 8) des tailles d'enregistrement (RECL) dans les opérations OPEN (cf. 5.3.2 p. 37) et INQUIRE (cf. 5.3.7 p. 40).

```
1 PROGRAM unites_standard
2 ! standard fortran 2003 : emploi du module ISO_FORTRAN_ENV
3 USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : error_unit, input_unit, output_unit
4 IMPLICIT NONE
5 INTEGER :: i
6 WRITE(*,*) "entrer un entier"
7 ! lecture sur l'entrée standard (descripteur 0 sous unix)
8 ! redirection d'entrée par < fichier
9 READ(input_unit, *)i
10 WRITE(*,*) "i vaut ", i
11 ! écriture sur la sortie standard (descripteur 1 sous unix)
12 ! redirection par 2> fichier
13 WRITE(output_unit,*) "message sur la sortie standard"
14 ! écriture sur la sortie standard d'erreur (descripteur 2 sous unix)
15 ! redirection par 2> fichier
16 WRITE(error_unit,*) "message d'erreur"
17 END PROGRAM unites_standard
```

Noter que fortran 2003 fournit aussi les deux fonctions IS_IOSTAT_END et IS_IOSTAT_EOR qui attendent pour argument la valeur du code de retour IOSTAT et rendent un booléen vrai en cas de fin

de fichier ou de fin d'enregistrement. Ces fonctions booléennes testent donc l'égalité à `IOSTAT_END` ou `IOSTAT_EOR`.

5.3.2 OPEN

L'instruction `OPEN` permet de connecter un fichier à une unité logique.

```
OPEN ([UNIT=]<entier>, FILE=<nom_fichier>, ERR=<étiquette>, &
      IOSTAT=<iostat>, ...)
```

Dans cette syntaxe, seul le premier argument (numéro d'unité logique) est positionnel et tous les paramètres sont accessibles par mot-clef. Seuls les paramètres désignés par les mots-clefs `UNIT` et `FILE` sont presque toujours obligatoires, la plupart des autres paramètres sont optionnels avec des valeurs par défaut indiquées ci-dessous.

- `[UNIT=]<entier>` désigne le numéro de l'unité logique ; cette unité logique ne doit pas être déjà connectée ¹⁵ ;
- `FORM=<fmt>` où `<fmt>` est une chaîne de caractères égale à `'FORMATTED'` (valeur par défaut) ou `'UNFORMATTED'` pour un fichier non formaté ;
- `FILE=<nom_fichier>` désigne le nom du fichier à connecter, sous forme d'une chaîne de caractères (constante ou variable) ;
- `ERR=<étiquette>` désigne une étiquette numérique permettant un branchement ¹⁶ en cas d'erreur ;
- `IOSTAT=<iostat>` est un paramètre de sortie qui rend une variable entière nulle si l'opération s'est déroulée sans erreur, ou sinon une valeur précisant le type d'erreur rencontrée ;
- `STATUS=<status>` désigne l'état du fichier à ouvrir et peut valoir :
 - `'OLD'` dans le cas d'un fichier préexistant
 - `'NEW'` dans le cas où le fichier n'existe pas
 - `'REPLACE'` qui permet le remplacement ou la création suivant que le fichier existe ou non
 - `'UNKNOWN'` dans le cas où on ne sait pas a priori si le fichier existe
 - `'SCRATCH'` pour un fichier temporaire qui sera détruit lors de l'instruction `CLOSE`
- `ACCESS=<méthode>` spécifie si le fichier est en accès séquentiel `'SEQUENTIAL'` (par défaut), en accès direct `'DIRECT'` ou (en fortran 2003 seulement) en accès stream `'STREAM'` ; ⇐ f2003
- `ACTION=<mode>` indique les opérations possibles sur le fichier (elles supposent que l'utilisateur possède des droits d'accès suffisants sur le fichier)
 - `'READ'` si ouvert en lecture seule
 - `'WRITE'` si ouvert en écriture seule
 - `'READWRITE'` (défaut) si ouvert en lecture et écriture
- `POSITION=<pos>` précise la position à l'ouverture d'un fichier en accès séquentiel :
 - `'APPEND'` positionne à la fin du dernier enregistrement avant la marque de fin de fichier et permet d'ajouter des informations à la fin d'un fichier existant
 - `'REWIND'` positionne en début de fichier
 - `'ASIS'` ouvre le fichier à la position courante s'il est déjà connecté, mais dépend du processeur sinon
- `RECL=<entier>` indique la longueur des enregistrements pour un fichier à accès direct (paramètre obligatoire dans ce cas) ou la longueur maximale des enregistrements pour un fichier séquentiel (`RECL` s'exprime en nombre de caractères pour les fichiers formatés) ;
- `BLANK` est une chaîne de caractères spécifiant l'interprétation des blancs dans les lectures de données numériques : si elle vaut `'null'` (par défaut), les blancs sont ignorés, si elle vaut `'zero'`, ils sont interprétés comme des zéros ;
- `DELIM` spécifie le délimiteur des chaînes de caractères en écriture, qui peut être `'none'` (défaut), `'quote'` ou `'apostrophe'`.
- `DECIMAL` spécifie le séparateur entre la partie entière et la partie fractionnaire des nombres flottants. Par défaut, c'est le point, mais la virgule peut lui être substituée par `DECIMAL='comma'`. ⇐ f2003

15. On peut vérifier qu'un numéro d'unité logique est disponible en utilisant l'instruction `INQUIRE` (cf. 5.3.7, p. 40).

16. On évitera ce branchement en testant si le code de retour `IOSTAT` est non nul.

Pour rétablir le point, il faut préciser `DECIMAL='point'`.

Noter que ces conventions peuvent aussi être modifiées dans les ordres de lecture (cf. 5.3.4, p. 39) et d'écriture (cf. 5.3.5, p. 39).

f2003 ⇒

- `ENCODING` permet de préciser le codage des chaînes de caractères dans le fichier formaté à ouvrir. Les fichiers texte au codage UTF-8 d'Unicode sont pris en charge par le standard 2003 grâce au paramètre optionnel `ENCODING='UTF-8'` de `OPEN`. Il est prudent de lire de tels fichiers dans des variables chaîne de caractères dont la variante de type, par exemple `'ISO_10646'` (cf. 8.1.3, p. 85), permet de représenter les caractères d'Unicode.

Exemple de lecture d'un fichier de 6 caractères codés en UTF-8 et d'écriture en code ISO-8859-1

```

1 PROGRAM io_utf8_latin1
2 IMPLICIT NONE
3 ! 9 oct 2012 fichier source en iso-latin1
4 ! avec NAG Fortran Compiler Release 5.3(854)
5 ! les kind des chaînes (4 en utf-32 et 1 en default)
6 INTEGER, PARAMETER :: utf32=SELECTED_CHAR_KIND('ISO_10646') ! UCS_4 = UTF-32
7 INTEGER, PARAMETER :: def_char=SELECTED_CHAR_KIND('DEFAULT') ! Default=latin1
8 CHARACTER(len=6, kind=def_char) :: latin
9 CHARACTER(len=6, kind=utf32) :: lu_utf32 ! chaîne UTF-32
10 ! lecture sur fichier externe codé UTF-8
11 OPEN(unit=10, file='fichier-utf8.txt', encoding='utf-8')
12 READ(10, *) lu_utf32(1:)
13 CLOSE(10)
14 latin = lu_utf32 ! transcodage UTF32-> default =latin1
15 WRITE(*,*) "latin1:", latin ! affichage iso-latin1
16 ! écriture sur fichier externe en latin1
17 OPEN(unit=10, file='fichier-latin1.txt', encoding='DEFAULT') ! latin1
18 WRITE(10,'(a)') latin
19 CLOSE(10)
20 END PROGRAM io_utf8_latin1

```

Le texte du fichier codé UTF-8 est lu dans des variables de type chaînes UTF-32, puis converti¹⁷ dans des variables de type ISO-8859-1 et enfin écrit dans un fichier au codage par défaut.

Exemples

Pour connecter le fichier formaté `lect.dat` à l'unité 10 en accès séquentiel, on peut se contenter de l'instruction :

```
OPEN(UNIT=10, FILE='lect.dat')
```

Il est aussi possible de préciser :

```
OPEN(UNIT=10, FILE='lect.dat', FORM='formatted', ACCESS='sequential')
```

Mais il est prudent de prévoir une gestion des erreurs avec par exemple :

```

INTEGER :: ok ! status de retour
OPEN(UNIT=10, FILE='lect.dat', IOSTAT=ok)
IF( ok /= 0 ) THEN
  PRINT *, 'erreur fichier lect.dat'
  STOP      ! arrêt à défaut de meilleure solution
END IF

```

¹⁷. Noter que dans la norme fortran 2003, seul le transcodage depuis et vers le code par défaut est requis : ici, le défaut étant ISO-8859-1, il permet de transcoder des chaînes avec des caractères non-ascii en UTF-32 vers du latin1.

5.3.3 CLOSE

L'instruction CLOSE permet de déconnecter une unité logique d'un fichier¹⁸. Elle libère l'unité logique pour une nouvelle connexion.

```
CLOSE ([UNIT=<entier>, ERR=<étiquette>, &
        IOSTAT=<iostat>, STATUS=<stat>)
```

Les arguments de CLOSE sont tous optionnels sauf l'unité logique et ont la même signification que dans l'instruction OPEN, sauf STATUS=<status> qui désigne ici le devenir du fichier après déconnexion et peut valoir :

- 'KEEP' (par défaut¹⁹) pour conserver un fichier préexistant ;
- 'DELETE' pour détruire le fichier à la déconnexion.

5.3.4 READ

L'instruction READ possède deux variantes :

- dans la première, l'unité logique n'est pas précisée et il s'agit de l'entrée standard ; elle est le symétrique en entrée de PRINT (cf. 5.1, p. 30) ;
- la seconde respecte la syntaxe suivante, où seul le premier argument (l'unité logique) est positionnel :

```
READ ([UNIT=<entier>, ERR=<étiquette>, END=<étiquette>, &
        IOSTAT=<iostat>, ...) liste d'entrées
```

Selon le type de fichier et d'accès, d'autres arguments deviennent obligatoires ou optionnels²⁰ :

- aucun autre argument pour les fichiers séquentiels non formatés ;
- l'argument FMT=<format> spécifiant le format est obligatoire et les arguments ADVANCE='no', SIZE=<taille> (qui donne le nombre de caractères effectivement lus par l'instruction READ), EOR=<étiquette> (branchement si on rencontre la fin de l'enregistrement en cas de lecture avec ADVANCE='no') sont optionnels pour les fichiers séquentiels formatés ;
- END=<étiquette> branchement si on rencontre la fin de fichier ;
- l'argument REC=<numéro_d_enregistrement> est obligatoire pour les fichiers en accès direct non formatés ; les arguments REC=<numéro_d_enregistrement> et FMT=<format> sont obligatoires pour les fichiers en accès direct formatés ;
- l'argument optionnel POS=<position> permet de choisir la position en unités de stockage dans les fichiers en accès stream.

Les branchements conditionnels EOR= et END= sont déconseillés en fortran 2003, où l'usage du `≡ f2003` module intrinsèque ISO_FORTRAN_ENV ou des fonctions booléennes IS_IOSTAT_END ou IS_IOSTAT_EOR (cf. 5.3.1, p. 36) permet de tester si le code de retour IOSTAT vaut IOSTAT_EOR ou IOSTAT_END.

5.3.5 WRITE

L'instruction WRITE suit la syntaxe générale

```
WRITE ([UNIT=<entier>, ERR=<étiquette>, &
        IOSTAT=<iostat>, ...) liste_de_sortie
```

où seul le premier paramètre est positionnel et à laquelle, il faut ajouter suivant le type de fichier et d'accès d'autres paramètres obligatoires ou optionnels (cf. 5.4.4, p. 45) :

- aucun autre argument pour les fichiers séquentiels non formatés ;

18. L'arrêt d'un programme provoque implicitement un CLOSE sur toutes les unités connectées par le programme

19. Sauf pour les fichiers ouverts avec STATUS='scratch', pour lesquels la valeur par défaut est DELETE.

20. avec un sens identique à celui éventuellement donné pour l'instruction OPEN

- l'argument `FMT=<format>` spécifiant le format est obligatoire et l'argument `ADVANCE='no'`, est optionnel pour les fichiers séquentiels formatés ;
- l'argument `REC=<numéro_d_enregistrement>` est obligatoire pour les fichiers en accès direct non formatés ;
- les arguments `REC=<numéro_d_enregistrement>` et `FMT=<format>` sont obligatoires pour les fichiers en accès direct formatés ;
- l'argument optionnel `POS=<position>` permet de choisir la position en unités de stockage dans les fichiers en accès stream.

5.3.6 PRINT

L'instruction `PRINT <fmt> [, <liste>]` permet d'écrire en mode formaté sur la sortie standard la liste optionnelle des variables ou constantes indiquées après la virgule ; elle est équivalente à `WRITE(*, <fmt>) [, <liste>]`.

5.3.7 INQUIRE

L'instruction `INQUIRE` permet de s'enquérir de certains paramètres caractérisant un flux d'entrée-sortie, défini :

- soit par une unité logique ;
- soit par un nom de fichier ;
- soit par une liste de sortie.

Par exemple, avant de connecter un fichier à une unité logique via l'instruction `OPEN`, on peut rechercher un numéro d'unité logique disponible et s'assurer que le fichier existe.

```

LOGICAL :: pris, present
INTEGER :: n = 1, nmax = 99
DO n = 1, nmax
  INQUIRE(UNIT=n, OPENED=pris)
  IF (.not. pris) EXIT           ! l'unité n est libre
END DO
INQUIRE(FILE='fichier', EXIST=present)
IF(present) THEN
  OPEN(UNIT=n, FILE='fichier', ...) ! le fichier existe
  ...
ELSE
  ...
END IF

```

f2003 ⇒ Le paramètre de sortie `ENCODING` du standard 2003 permet d'obtenir le codage des chaînes de caractères dans un fichier formaté connecté à une unité logique. On peut ainsi détecter le codage 'UTF-8' d'Unicode mais 'UNKNOWN' sera rendu si la détection n'est pas possible.

D'autre part, l'instruction `INQUIRE` permet d'évaluer la taille de l'enregistrement que créerait l'écriture non-formatée d'une liste²¹. Elle permet ainsi de connaître la longueur d'un enregistrement d'un fichier non-formaté en accès direct avant de le connecter à une unité logique (*cf.* 5.6.2, p. 49), de façon à rendre l'instruction d'ouverture du fichier indépendante de la machine.

```

INTEGER :: l, iunit
INQUIRE(iolength=l) <liste d'expressions>
...
OPEN(iunit, file=..., access='direct', form='unformatted', recl=l)
...

```

21. Cette taille dépend en général de la machine.

Enfin, pour les fichiers en accès stream, INQUIRE permet de s'enquérir de la position courante dans le fichier exprimée en unités de stockage qui sont souvent des octets. Dans le cas d'un fichier non formaté, les décalages peuvent être calculés avec INQUIRE(ilolength=).

```

INTEGER :: iunit, position
REAL :: x, y
OPEN(iunit, file=..., access='stream', form='unformatted')
...
INQUIRE(iunit, POS = position) ! POS = argument de sortie
READ(iunit) x ! lecture de x
...
WRITE(iunit, POS = position) y ! POS = argument d'entrée
! écriture de y à la place de x
...

```

5.3.8 Instructions de positionnement dans les fichiers

Il existe enfin des instructions permettant, sans effectuer de transfert de données, de modifier la position courante dans un fichier d'accès séquentiel :

- REWIND ([UNIT=]<entier> [, ERR=<étiquette>] [, IOSTAT=<iostat>])
positionne en début de fichier;
- BACKSPACE ([UNIT=]<entier> [, ERR=<étiquette>] [, IOSTAT=<iostat>])
positionne avant l'enregistrement courant si on est dans un enregistrement ou avant le précédent enregistrement si on est entre deux enregistrements (cette instruction permet la relecture d'un enregistrement);
- ENDFILE ([UNIT=]<entier> [, ERR=<étiquette>] [, IOSTAT=<iostat>])
écrit²² une marque de fin de fichier à la position courante et positionne juste après cette marque de fin de fichier.

5.3.9 Entrées-sorties sans avancement automatique

Le paramètre ADVANCE='no' permet de ne pas avancer automatiquement à l'enregistrement suivant à chaque instruction d'entrée-sortie sur un fichier en accès séquentiel et donc de transférer des parties d'enregistrement, c'est à dire de lignes pour les interactions avec le terminal²³.

Par exemple, pour maintenir le curseur juste après une invite à saisir une donnée, on écrira :

```

WRITE(*, '(a)', ADVANCE='no') 'entrer un entier : '
READ(*, *) i

```

De même, pour traiter une saisie incomplète au clavier, on pourra utiliser le branchement EOR et le paramètre de retour SIZE :

```

INTEGER :: nlus = 4
CHARACTER(len=4) :: chaine='++++'
WRITE(*, '(a)') 'entrer quatre caractères'
READ(*, '(a4)', advance='no', size=nlus, eor=999) chaine(1:4)
GO TO 1000 ! si lecture correcte
999 WRITE(*,*) 'saisie tronquée' ! si retour chariot avant les 4 caractères
1000 WRITE(*, *) nlus, ' caractères lus'

```

22. Ne pas croire que ENDFILE est une simple instruction de positionnement en fin de fichier. Ce positionnement est possible à l'ouverture du fichier via l'option POSITION='APPEND' de l'instruction OPEN (cf. 5.3.2).

23. C'est le comportement par défaut en langage C, où les changements de ligne doivent être explicitement spécifiés par des \n. On retrouve ce comportement par défaut avec l'accès STREAM du fortran 2003.

5.4 Descripteurs de format

Les règles de conversion entre représentation interne des données et chaîne de caractères des fichiers formatés sont déterminées par le format de lecture ou d'écriture. Le format peut être spécifié sous une des formes suivantes :

- le format libre, désigné par *;
- une chaîne de caractères constante ou variable, décrivant le format entre parenthèses ;
- une étiquette numérique (entier positif) faisant référence à une instruction dite de format.

Le format libre est utilisé pour les échanges avec le terminal, en phase de test ou pour accéder à des fichiers manipulés toujours sur la même machine, car il n'est pas portable. Mais, pour des échanges de données plus massifs avec des fichiers ou de données dont le format peut être imposé par des applications externes, il est souvent nécessaire de préciser les règles de conversion en entrée ou en sortie grâce à une spécification de format. La dernière forme, avec étiquette numérique d'instruction est préférée lorsqu'une même spécification de format est utilisée plusieurs fois dans une unité de programme ; il est alors d'usage de regrouper les instructions de format en fin de programme et de leur réserver une plage d'étiquettes numériques particulière.

```

CHARACTER(len=4) :: fmt1, fmt2
INTEGER :: j = 12
WRITE(*, *) 'bonjour'           ! format libre
WRITE(*, '("bonjour")')
WRITE(*, '(a7)') 'bonjour'     ! chaîne de caractères constante
fmt1='(a7)'
WRITE(*, fmt1) 'bonjour'       ! chaîne de caractères variable
WRITE(*, 1000) 'bonjour'      ! étiquette numérique
1000 FORMAT(a7)
!
WRITE(*, *) j                   ! format libre
WRITE(*, '(i)') j              ! chaîne de caractères constante
WRITE(*, '(i2)') j
fmt2='(i2)'
WRITE(*, fmt2) j               ! chaîne de caractères variable
fmt2='(i4)'
WRITE(*, fmt2) j               ! chaîne de caractères variable
WRITE(*, 1100) j              ! étiquette numérique
1100 FORMAT(i2)

```

Comme le format est une expression de type chaîne de caractères, il peut être défini par une variable évaluée lors de l'exécution : cette méthode sera présentée à la sous-section 5.4.5, p. 45.

On distingue les *descripteurs actifs*, qui spécifient le mode de conversion d'une donnée en chaîne de caractères ou réciproquement (a priori un descripteur actif par élément de la liste d'entrée-sortie), les chaînes de caractères et les *descripteurs de contrôle*, qui, entre autres, règlent le positionnement, gèrent les espaces et les changements d'enregistrement.

5.4.1 Descripteurs actifs

D'une façon générale, le premier paramètre (entier), n , détermine le nombre *total* de caractères occupés par la donnée codée, c'est à dire avant conversion en lecture ou après conversion en écriture. La signification de l'éventuel deuxième paramètre dépend du type de donnée à convertir.

Si le champ est plus large que nécessaire, la donnée codée est justifiée à droite ; en écriture, le champ est complété par des blancs à gauche. Si la largeur n du champ de sortie est insuffisante compte tenu de tous les caractères requis (signe, exposant éventuel avec son signe, ...), la conversion en sortie est impossible et provoque l'écriture de n signes *²⁴.

²⁴. Ce comportement est contraire à celui du langage C, qui étendrait alors le champ occupé pour convertir la donnée.

- entiers (en sortie, p précise le nombre minimal de caractères de chiffres ou de lettres (hors signe et blancs), quitte à compléter par des zéros à gauche) :
 - $I_n [.p]$ en base 10
- réels (p spécifie le nombre de chiffres après la virgule) :
 - $F_n.p$ en notation décimale
 - en virgule flottante :
 - mantisse (<1) plus exposant : $E_n.p$
 - mantisse (<1) plus exposant avec q chiffres : $En.p [eq]$
 - notation scientifique (mantisse entre 1 et 10) : $ES_n.p$
 - notation ingénieur (mantisse entre 1 et 1000, exposant multiple de 3) : $EN_n.p$
- booléens : L_n
- chaîne de caractères : A_n

Pour une analyse plus bas niveau, les descripteurs B, O et Z (voir les notations associées pour les constantes, 2.4, p. 14) sont utilisables pour les entiers et, en fortran 2008, les réels : ⇐ f2008

- $B_n [.p]$ en binaire
- $O_n [.p]$ en octal
- $Z_n [.p]$ en hexadécimal

Enfin, un descripteur qualifié de général permet la conversion de tous les types de données : $G_n.p$. Dans le cas des réels, il assure une conversion en virgule fixe (format $F_n.p$) si possible ou sinon en virgule flottante (format $E_n.p$).

Les nombres complexes sont traités comme un couple de nombres réels, entre parenthèses et séparés par une virgule, comme dans les constantes complexes ; ils nécessitent donc deux descripteurs de format réels.

Enfin, on peut demander que la largeur du champ s'ajuste à l'exécution de façon à n'utiliser que le minimum de caractères nécessaires en spécifiant une largeur nulle pour les spécificateurs I, B, O, Z et F seulement ²⁵. ⇐ f95/2003

5.4.2 Descripteurs de contrôle

- X provoque l'écriture d'un blanc en sortie et le saut d'un caractère en entrée
- / provoque le changement d'enregistrement (c'est à dire le changement de ligne)
- Tabulations
 - Tn positionne le « pointeur » sur le n^e caractère de l'enregistrement
 - TLn (tab left) déplace le « pointeur » de n caractères vers la gauche
 - TRn (tab right) déplace le « pointeur » de n caractères vers la droite ; TRn est équivalent à nX.
- Interprétation des blancs en lecture ²⁶
 - BN ignore les blancs (internes ou finaux) en lecture de champs numériques
 - BZ considère les blancs (internes ou finaux) comme des zéros en lecture de champs numériques
- Signe plus optionnel : les trois options qui suivent contrôlent l'écriture éventuelle d'un signe + devant les quantités positives ; si une option est spécifiée, elle reste en vigueur tant qu'une autre option ne vient pas la contredire, pour toutes les écritures qui suivent dans la même instruction.
 - SP (sign print) force l'écriture du signe + devant les quantités positives
 - SS (sign suppress) supprime l'écriture du signe + devant les quantités positives
 - S rétablit l'option par défaut dépendant du processeur ²⁷ pour l'écriture du signe + devant les quantités positives
- : interrompt la prise en compte des descripteurs passifs lorsque la liste est épuisée par les descripteurs actifs du format

²⁵. Ce format de largeur minimale spécifiée par $n=0$ est comparable au format par défaut du langage C, où l'on ne précise pas la largeur du champ.

²⁶. Bien noter que les formats BN et BZ ne contrôlent pas blancs et zéros en écriture, qui dépendent du compilateur. L'interprétation des blancs en lecture peut aussi être spécifiée de façon plus générale via le paramètre optionnel BLANK= de l'instruction OPEN.

²⁷. Pour écrire des fichiers portables, on ne laissera donc pas le choix au processeur, et on spécifiera systématiquement soit SP, soit SS.

- kP descripteur de changement d'échelle, applicable seulement aux réels et complexes, et qui *modifie les valeurs des réels lus ou écrits sans exposant*²⁸, mais simplement la forme de ceux écrits avec exposant. Il s'applique à toutes les conversions via les descripteurs qui le suivent dans la même instruction, et peut être modifié par un descripteur nP ou annulé par OP .
 - en entrée, kP divise par le facteur 10^k les valeurs lues *sans exposant*²⁹, permettant par exemple une saisie plus rapide d'une série de valeurs très faibles;
 - en sortie, c'est le format qui impose la présence ou l'absence d'un exposant (sauf pour le format G) et l'effet de kP appliqué à un réel dépend du format de conversion :
 - en format F , il multiplie par 10^k ;
 - en format E , la mantisse est multipliée par 10^k alors que l'exposant est diminué de k , la valeur restant inchangée;
 - en format G , l'effet dépend du choix entre E et F suivant la valeur;
 - en format EN et ES , le facteur d'échelle est sans effet.
- f2003 ⇒ - dc (**d**ecimal **c**omma) ou dp (**d**ecimal **p**oint) spécifient le séparateur décimal : par défaut c'est un point, mais ces spécificateurs permettent de le changer et la modification perdure jusqu'au prochain descripteur dp ou dc . Ce choix³⁰ peut aussi être fait au niveau des instructions $OPEN$ (cf. 5.3.2, p. 37), $READ$ (cf. 5.3.4, p. 39), et $WRITE$ (cf. 5.3.5, p. 39), selon la syntaxe $decimal='point'$ ou $decimal='comma'$. Dans le cas où le séparateur décimal est la virgule, en entrée, le séparateur entre valeurs passe de la virgule au point-virgule « ; » (cf. 5.1.2, p. 31).

Exemple : à la saisie `1,23;5,67e8`, le programme suivant

```
PROGRAM decimal_comma
IMPLICIT NONE
REAL :: r,t
WRITE(*,*) 'entrer 2 réels séparés par ; (notation décimale avec virgule)'
READ(*, *, decimal='comma') r, t
WRITE(*, '("avec virgule", dc, 2e12.4)') r, t
WRITE(*, '("avec point ", dp, 2e12.4)') r, t
END PROGRAM decimal_comma
```

affiche :

```
avec virgule  0,1230E+01  0,5670E+09
avec point   0.1230E+01  0.5670E+09
```

5.4.3 Syntaxe des formats et règles d'exploration

- Une liste de descripteurs, séparée par des virgules, peut être érigée en groupe de descripteurs, délimité par des parenthèses.
- Un facteur de répétition (entier positif préfixant le descripteur) peut être appliqué à un descripteur de format ou à un groupe de descripteurs.
- **Chaque instruction d'entrée-sortie provoque le passage à l'enregistrement suivant**, sauf si l'argument optionnel $ADVANCE='no'$ a été spécifié.

En principe la liste d'entrée-sortie comporte autant d'éléments qu'il y a de descripteurs actifs dans le format. Dans le cas contraire, on applique les principes suivants :

- **Si la liste épuise les descripteurs actifs du format, il y a passage à l'enregistrement suivant**³¹ et ré-exploration du format en partant de la parenthèse ouvrante correspondant à l'avant-dernière parenthèse fermante du format, et en tenant compte de son éventuel facteur de répétition.

28. On évitera l'emploi du descripteur kP à cause de cette modification de valeur peu explicite lors de la conversion.

29. En particulier, malgré un format de lecture E ou D , on peut lire un réel avec ou sans exposant et la conversion dépendra alors de la forme lors de la lecture, rendant le résultat imprévisible pour une saisie au clavier : cela constitue une raison supplémentaire pour éviter l'emploi du format P en lecture.

30. Avec le compilateur $g95$, il est possible de choisir la virgule comme séparateur décimal au moment de l'exécution, grâce à la variable d'environnement $G95_COMMA$ booléenne, par défaut à $FALSE$ (cf. E.5, p. 152).

31. Sauf pour les fichiers à accès $stream$ ou dans le cas où $ADVANCE=no$ a été préalablement spécifié.

- Si la liste contient moins d'éléments qu'il y a de descripteurs actifs dans le format, il est honoré jusqu'au premier descripteur actif redondant exclus, sauf si on rencontre le descripteur de contrôle :, qui interrompt l'exploration du format lorsque la liste est épuisée.

5.4.4 Boucles et changements d'enregistrement

Comme, en accès séquentiel, chaque instruction d'entrée-sortie fait par défaut progresser d'un enregistrement dans le fichier, une boucle explicite sur le nombre d'éléments d'un tableau avec une instruction `READ/WRITE` dans la boucle ne permet pas de lire ou écrire tous les éléments du tableau dans un enregistrement unique. On peut alors utiliser soit une instruction agissant sur le tableau global, soit une boucle implicite. Mais, pour éviter des changements d'enregistrements intempestifs, il faut aussi s'assurer (*cf.* 5.4.3, p. 44) que le format spécifié n'est pas épuisé par la liste, ce qui provoquerait une réexploration du format.

```

1 PROGRAM format_tab
2 IMPLICIT NONE
3 INTEGER, DIMENSION(3) :: t = (/ 1, 2, 3 /) ! tableau initialisé
4 INTEGER :: i
5 WRITE(*, '(3i4)') t(:) ! tableau global => affichage en ligne
6 DO i = 1, 3 ! boucle explicite
7   WRITE(*, '(i4)') -t(i) ! => un changement d'enregistrement par ordre
8 END DO ! => affichage en colonne
9 WRITE(*, '(3i4)') (2* t(i), i=1, 3)! boucle implicite => affichage en ligne
10 WRITE(*, '(i4)') (-2*t(i), i=1, 3) ! format réexploré => en colonne
11 DO i = 1, 3 ! boucle explicite
12   WRITE(*, '(i4)', advance='no') 3*t(i) ! mais option advance='no'
13 END DO ! => affichage en ligne
14 WRITE(*, *) !passage à l'enregistrement suivant à la fin
15 END PROGRAM format_tab

```

Le programme précédent affiche successivement :

- `t` en ligne avec le tableau global (ligne 5);
- `-t` en colonne avec la boucle explicite (lignes 6 à 8);
- `2*t` en ligne avec la boucle implicite (ligne 9);
- `-2*t` en colonne malgré la boucle implicite (ligne 10), car le format ne satisfait qu'un élément de la liste et est donc réexploré;
- `3*t` en ligne malgré la boucle explicite (lignes 11 à 13), grâce à `ADVANCE='no'`.

	1	2	3
– t en ligne avec le tableau global (ligne 5);	-1		
– -t en colonne avec la boucle explicite (lignes 6 à 8);	-2		
– 2*t en ligne avec la boucle implicite (ligne 9);	-3		
– -2*t en colonne malgré la boucle implicite (ligne 10), car le format ne satisfait qu'un élément de la liste et est donc réexploré;	2	4	6
– 3*t en ligne malgré la boucle explicite (lignes 11 à 13), grâce à ADVANCE='no'.	-2		
	-4		
	-6		
	3	6	9

5.4.5 Format variable

Il n'existe pas à proprement parler³² de format variable en fortran. Mais il est possible de déterminer le format à l'exécution car il peut être défini par une variable chaîne de caractères. L'opération s'effectue donc en deux étapes avec par exemple :

- une instruction d'écriture sur un fichier interne qui construit la variable de type chaîne contenant le format ;
- l'instruction d'entrée-sortie utilisant ce format.

À titre d'exemple, le programme suivant montre comment choisir le nombre `n` de chiffres (entre 1 et 9) pour afficher un entier `i`.

³². Certaines extensions propriétaires du fortran, notamment le compilateur d'Intel (*cf.* chap. E.4, p. 152) et celui de Portland, permettent le format variable : des expressions numériques encadrées par les délimiteurs `<` et `>` peuvent figurer dans la chaîne de caractères définissant le format ; elles sont interprétées à chaque opération d'entrée-sortie. Par exemple, `WRITE(*, "(I<j>.<j>)"` j permet d'écrire l'entier j avec exactement j chiffres. Le format variable est aussi disponible dans le langage C avec le spécificateur de format `*` : `printf("%*d\n", n, i)`; imprime l'entier i avec n chiffres.

```

1 PROGRAM var_fmt
2 ! format variable à l'exécution
3 IMPLICIT NONE
4 INTEGER :: i = 123 ! le nombre (au maximum 9 chiffres) à afficher
5 INTEGER :: n      ! le nombre de chiffres (<=9) pour l'afficher"
6 CHARACTER(len=1) :: chiffres ! n codé sur un caractère
7 CHARACTER(len=6) :: fmt="(i?.?)" ! la chaîne de format à modifier
8 n = 4      ! premier cas
9 ! écriture sur fichier interne (en format fixe !)
10 WRITE(fmt(3:3), '(i1)') n
11 WRITE(fmt(5:5), '(i1)') n
12 WRITE(*, fmt) i
13 n = 7      ! deuxième cas
14 ! autre méthode : écriture sur une sous-chaîne
15 WRITE(chiffres, '(i1)') n
16 ! concaténation pour construire le format
17 fmt = "(i" // chiffres // "." // chiffres // ")"
18 WRITE(*, fmt) i
19 END PROGRAM var_fmt

```

Avec n=4 puis n=7, il produit les affichages successifs :

```

0123
0000123

```

5.5 Exemples d'entrées-sorties formatées

5.5.1 Descripteurs de données numériques en écriture

Écriture de données entières

```

1  écriture des entiers 1, 20, 300, -1, -20, -300 et 0
2  écriture de ces entiers en format libre
3  1 20 300 -1 -20 -300 0
4  écriture de ces entiers en format In[.p] (décimal)
5 |i1|.....|1|*|*|*|*|0|
6 |i1.0|...|1|*|*|*|*|_|
7 |i2|.....|_|1|20|**|-1|**|**|_|0|
8 |i2.2|...|01|20|**|**|**|**|00|
9 |i3|.....|_|_|1|_|20|300|_|-1|-20|***|_|_|0|
10 |i4|.....|_|_|_|1|_|_|20|_|300|_|_|-1|_|-20|-300|_|_|0|
11 |i4.2|...|_|_|01|_|_|20|_|300|_|-01|_|-20|-300|_|_|00|
12 |i5|.....|_|_|_|_|1|_|_|_|20|_|_|300|_|_|-1|_|_|-20|_|-300|_|_|_|0|
13 |sp,i5|..|_|_|_|+1|_|_|+20|_|+300|_|_|-1|_|_|-20|_|-300|_|_|+0|
14 |i5.0|...|_|_|_|_|1|_|_|_|20|_|_|300|_|_|-1|_|_|-20|_|-300|_|_|_|_|
15 |i5.4|...|_|0001|_|0020|_|0300|-0001|-0020|-0300|_|0000|
16 |i5.5|...|00001|00020|00300|*****|*****|*****|00000|
17   format I0 largeur ajustable (f95/2003 seulement )
18 |i0|.....|1|20|300|-1|-20|-300|0|
19 |i0.2|...|01|20|300|-01|-20|-300|00|
20  écriture de ces entiers en format Zn[.p] (hexadécimal)
21 |z8|.....|_|_|_|_|_|_|1|_|_|_|_|14|_|_|_|_|12C|FFFFFFFF|FFFFFFFEC|FFFFFFED4|_|_|_|_|_|_|0|
22 |z8.8|...|00000001|00000014|0000012C|FFFFFFFF|FFFFFFFEC|FFFFFFED4|00000000|
23   format Z0 largeur ajustable (f95/2003 seulement)
24 |z0|.....|1|14|12C|FFFFFFFF|FFFFFFFEC|FFFFFFED4|0|

```



```

4  .Répétition: (6i4)
5  UUU1UU20U300UU-1U-20-300
6  .Groupement avant répétition : (6(x,i4))
7  UUUU1UUU20UU300UUU-1UU-20U-300
8  .Changement d'enregistrement forcé par "/" : (3(x,i4),/,3(x,i4))
9  UUUU1UUU20UU300
10 UUU-1UU-20U-300
11
12 .Liste plus longue => réexploration avec changement d'enregistrement
13 .Réexploration du format (x, i4, x, i4)
14 UUUU1UUU20
15 UU300UUU-1
16 UU-20U-300
17 .Réexploration du format (3(x, i4))
18 UUUU1UUU20UU300
19 UUU-1UU-20U-300
20 .Réexploration à partir de 3(x, i4) : (9x, 3(x, i4))
21 UUUUUUUUUUUUUUUUU1UUU20UU300
22 UUU-1UU-20U-300
23
24 .Liste plus courte => arrêt avant le 1er descripteur actif inutilisé
25 .arrêt après "<" dans le format (8("<","<","i4,>",">"))
26 [<UUU1>][<UU20>][<U300>][<UU-1>][<U-20>][<-300>][<
27 .Liste plus courte avec ":" => arrêt juste après le ":"
28 ..suivant le dernier descripteur actif utile
29 .arrêt via ":" (8("<","<","i4,>",":",">"))
30 [<UUU1>][<UU20>][<U300>][<UU-1>][<U-20>][<-300>
31
32 .Usage des tabulations
33 .Référence (6(6x, i4))
34 UUUUUUUUUUUUUUUUUUU1UUUUUUUUUUUUUUUUUUUU300UUUUUUUUUUUUUUUUUUUU-1UUUUUUUUUUUUUUUUUUUU-20UUUUUUUUUUUUUUUUUUUU-300
35 1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
36 .Tabulation à droite format (6(tr6, i4))
37 UUUUUUUUUUUUUUUUUUU1UUUUUUUUUUUUUUUUUUUU300UUUUUUUUUUUUUUUUUUUU-1UUUUUUUUUUUUUUUUUUUU-20UUUUUUUUUUUUUUUUUUUU-300
38 .Tabulations absolues et à gauche => changement d'ordre possible!)
39 .format (t7,i4, t17,i4, t27,i4, t57,i4, t114,i4, t114,i4)
40 UUUUUUUUUUUUUUUUUUU1UUUUUUUUUUUUUUUUUUUU300UUUUUUUUUUUUUUUUUUUU-300UUUUUUUUUUUUUUUUUUUU-20UUUUUUUUUUUUUUUUUUUU-1

```

5.6 Exemples de fichiers en accès direct

Les deux exemples élémentaires suivants présentent l'écriture d'un fichier à accès direct, sa lecture avec affichage des données et la mise à jour d'un des enregistrements.

5.6.1 Exemple de fichier formaté en accès direct

```

1 PROGRAM direct
2 ! fichier formaté en accès direct
3 IMPLICIT NONE
4 INTEGER :: ok=0, i, ir, nr
5 REAL :: a
6 INTEGER, PARAMETER :: iunit=11, n=10, recl=12
7 ! création du fichier en accès direct
8 OPEN(file='base.dat', unit=iunit, access='direct', recl=recl, &
9 form='formatted', status='replace')
10 ecriture: DO i=1, n

```

```

11 WRITE(unit=iunit, rec=i, fmt='(i4, 2x, f6.2)') 1000+i, 10*i + float(i)/10.
12 END DO ecriture
13 ! lecture du fichier en accès direct et affichage
14 ir=1
15 lecture: DO ! on ne sait pas a priori où s'arrêter
16 READ(unit=iunit, rec=ir, fmt='(i4, 2x, f6.2)', iostat=ok) i, a
17 IF (ok /= 0) EXIT ! fin de fichier
18 WRITE(*, '(i4, 2x, f6.2)') i, a
19 ir = ir + 1
20 END DO lecture
21 WRITE(*,*) ir-1, ' enregistrements lus'
22 ok=0
23 ! modification d'un enregistrement du fichier en accès direct
24 WRITE(*,*) 'choisir un numéro d'enregistrement à modifier'
25 READ *, nr
26 READ(unit=iunit, rec=nr, fmt='(i4, 2x, f6.2)') i, a
27 WRITE(*,*) 'ancienne valeur ', a, ' entrer la nouvelle valeur réelle'
28 READ *, a
29 WRITE(unit=iunit, rec=nr, fmt='(i4, 2x, f6.2)') i, a
30 ir=1
31 relecture: DO ! on ne sait pas a priori où s'arrêter
32 READ(unit=iunit, rec=ir, fmt='(i4, 2x, f6.2)', iostat=ok) i, a
33 IF (ok /= 0) EXIT ! fin de fichier
34 WRITE(*, '(i4, 2x, f6.2)') i, a
35 ir = ir + 1
36 END DO relecture
37 WRITE(*,*) ir-1, ' enregistrements lus'
38 ok=0
39 CLOSE(iunit)
40 END PROGRAM direct

```

5.6.2 Exemple de fichier non-formaté en accès direct

Dans le cas non-formaté, c'est l'instruction `INQUIRE` (ligne 8) qui permet de déterminer la taille de l'enregistrement, qui dépend du processeur.

```

1 PROGRAM direct
2 ! fichier non formaté en accès direct
3 IMPLICIT NONE
4 INTEGER :: ok=0, i, ir, nr, recl
5 REAL :: a
6 INTEGER, PARAMETER :: iunit=11, n=10
7 ! calcul de la longueur d'un enregistrement via INQUIRE
8 INQUIRE(iolength=recl) i, a !
9 WRITE(*,*) 'longueur d'un enregistrement ', recl
10 ! création du fichier en accès direct
11 OPEN(file='base.dat', unit=iunit, access='direct', recl=recl, &
12 form='unformatted', status='replace')
13 ecriture: DO i=1, n
14 WRITE(unit=iunit, rec=i) 1000+i, float(10*i) + float(i)/10.
15 END DO ecriture
16 ! lecture du fichier en accès direct et affichage
17 ir=1
18 lecture: DO ! on ne sait pas a priori où s'arrêter
19 READ(unit=iunit, rec=ir, iostat=ok) i, a
20 IF (ok /= 0) EXIT ! fin de fichier

```

```
21 WRITE(*, '(i4, 2x, f6.2)') i, a
22   ir = ir + 1
23 END DO lecture
24 WRITE(*,*) ir-1, ' enregistrements lus'
25 ok=0
26 ! modification d'un enregistrement du fichier en accès direct
27 WRITE(*,*) 'choisir un numéro d'enregistrement à modifier'
28 READ *, nr
29 READ(unit=iunit, rec=nr) i, a
30 WRITE(*,*) 'ancienne valeur ', a, ' entrer la nouvelle valeur réelle'
31 READ *, a
32 WRITE(unit=iunit, rec=nr) i, a
33 ! re-lecture du fichier en accès direct et affichage
34 ir=1
35 relecture: DO ! on ne sait pas a priori où s'arrêter
36   READ(unit=iunit, rec=ir, iostat=ok) i, a
37   IF (ok /= 0) EXIT ! fin de fichier
38   WRITE(*, '(i4, 2x, f6.2)') i, a
39   ir = ir + 1
40 END DO relecture
41 WRITE(*,*) ir-1, ' enregistrements lus'
42 ok=0
43 CLOSE(iunit)
44 END PROGRAM direct
```

Chapitre 6

Procédures

6.1 Introduction

6.1.1 Intérêt des procédures

Dès qu'un programme commence à dépasser une page ou comporte des duplications d'instructions, il s'avère avantageux de le scinder en sous-ensembles plus élémentaires, nommés procédures, qui seront appelées par le programme principal. En itérant ce processus, on structure le code source en une hiérarchie de procédures plus concises et modulaires, qui permet :

- d'améliorer la lisibilité et de faciliter la vérification et la maintenance ;
- de réutiliser des outils déjà mis au point par d'autres applications, au prix d'une généralisation et d'un paramétrage des procédures.

Par exemple, au lieu de dupliquer à chaque usage toutes les instructions de calcul de la somme des n premiers entiers (le seul paramétrage de cet exemple élémentaire), il est préférable de les écrire une fois pour toutes dans un sous-programme noté `cumul` qui sera appelé chaque fois que nécessaire.

```
SUBROUTINE cumul(n, s)
  IMPLICIT NONE
  INTEGER, INTENT(in)  :: n           ! argument d'entrée
  INTEGER, INTENT(out) :: s           ! argument de sortie
  INTEGER              :: i           ! variable locale
  s = 0
  DO i = 1, n
    s = s + i
  END DO
END SUBROUTINE cumul
```

Les appels successifs de ce sous-programme se feront alors de la manière suivante :

```
...
CALL cumul(10, s)      ! appel de cumul pour sommer les 10 premiers entiers
PRINT *, ' somme des 10 premiers entiers = ', s
...
p = 5
CALL cumul(p, s)      ! appel de cumul pour sommer les 5 premiers entiers
PRINT *, ' somme des ', p, ' premiers entiers = ', s
...
```

6.1.2 Variables locales, automatiques et statiques

Les variables déclarées au sein d'une procédure sont, a priori, locales. On dit que leur *portée* (scope) est limitée à l'unité de programme. Par défaut, les variables locales d'une procédure n'ont d'existence garantie que pendant son exécution : elles sont qualifiées d'*automatiques* ; elles ne sont pas nécessairement mémorisées¹ entre deux appels. C'est le cas de la variable `i` dans la procédure `cumul` (cf. 6.1.1, p. 51).

On peut cependant forcer cette mémorisation, pour en faire des variables *statiques*, grâce à l'attribut `SAVE`. Ne pas croire qu'alors la portée de la variable soit étendue : elle reste locale mais est alors permanente. De plus, si une variable locale est initialisée lors de sa déclaration, elle devient immédiatement une variable statique². Ne pas oublier que l'initialisation est effectuée par le compilateur, une fois pour toute et diffère donc d'une instruction exécutable d'affectation effectuée à chaque appel³.

Par exemple, le sous-programme `compte` suivant affiche le nombre de fois qu'il a été appelé. Si la variable locale n'était pas *statique*, il faudrait passer `n` en argument de `compte` pour le mémoriser.

```

SUBROUTINE compte
  IMPLICIT NONE
  INTEGER, SAVE :: n = 0 ! n est locale, mais statique
  n = n + 1
  print *, n
END SUBROUTINE compte

```

Si l'on souhaite rendre statiques *toutes* les variables locales d'une procédure, ce qui est fortement déconseillé⁴, il suffit d'insérer l'ordre `SAVE`⁵ avant toutes les déclarations. On peut aussi passer une option au compilateur (cf. annexe E) : `-fstatic` pour le compilateur `g95`, `-qsave` pour `xlf` d'IBM, `-save` pour le compilateur `NAG`, `-save` pour le compilateur `ifort` d'INTEL, `-static` pour celui `f90` de DEC.

6.1.3 Arguments des procédures

La communication entre une procédure et le programme appelant peut se faire par le passage de paramètres appelés arguments de la procédure. Ces arguments sont :

- déclarés comme *arguments muets* (dummy arguments) formels ou symboliques lors de la définition de la procédure ;
- spécifiés comme *arguments effectifs* (actual arguments) lors des appels de la procédure, pouvant prendre des valeurs différentes à chaque appel.

Lors de l'appel de la procédure, la liste des arguments effectifs doit respecter :

- le nombre (hormis dans le cas d'arguments optionnels, cf. 6.5.1, p. 63),
- l'ordre (hormis en cas de passage par mot-clef, cf. 6.5.2, p. 64),
- et le type⁶ et la variante de type éventuelle

des arguments muets déclarés dans la procédure.

6.1.4 L'attribut `INTENT` des arguments

Le fortran 90 a introduit, dans un but de fiabilisation des communications entre procédures, un attribut facultatif de vocation, `INTENT`, permettant de spécifier si un argument est :

1. Cela dépend des options de compilation qui peuvent forcer ou non un stockage statique ;
 2. Par souci de lisibilité, on précisera explicitement l'attribut `SAVE` dans la déclaration de toute variable locale initialisée, au lieu de s'appuyer sur le caractère statique implicite.
 3. Au contraire, en C, une variable locale initialisée est initialisée à l'exécution, à chaque appel à la fonction hôte ; pour la rendre permanente (statique), il faut lui adjoindre l'attribut `static`.
 4. Sauf lors de certaines opérations de débogage.
 5. L'instruction `SAVE` doit alors être seule sur la ligne. Elle peut aussi être utilisée sous la forme `SAVE :: liste_de_variables` pour déclarer statiques une liste de variables.
 6. Dans le cas d'une procédure générique (cf. 10.1, p. 97), c'est le type des arguments effectifs passés qui permettra d'aiguiller vers la procédure spécifique adéquate.

- `INTENT(in)` : un argument d'entrée, qui ne doit pas être modifié par la procédure (c'est le seul cas où l'argument effectif peut être une expression, en particulier une constante);
- `INTENT(out)` : un argument de sortie, qui doit affecté au sein de la procédure (il est indéterminé au début de la procédure);
- `INTENT(inout)` : un argument d'entrée-sortie, fourni à la procédure, mais éventuellement modifié par cette procédure.

En précisant ces attributs, on permet au compilateur un meilleur contrôle, lui donnant les moyens de détecter par exemple une modification involontaire d'un argument d'entrée dans la procédure, ou l'absence d'affectation d'un argument de sortie au sein de la procédure. ← ♡

Par exemple, le calcul de l'aire d'un triangle effectué maladroitement comme suit, sans préciser la vocation des arguments, produit un effet de bord involontaire qui est de diviser `base` par deux.

```

SUBROUTINE aire(base, hauteur, surface) ! version sans vocation des arguments
  IMPLICIT NONE
  REAL :: base, hauteur, surface      ! déclaration des arguments
  base = base / 2.                    ! modification de base autorisée
  surface = base * hauteur             ! forme maladroite du calcul de surface
END SUBROUTINE aire

```

Si on avait précisé que l'argument `base` est un argument d'entrée, le compilateur n'aurait pas accepté la modification de la variable `base` dans le sous-programme `aire`.

```

SUBROUTINE aire(base, hauteur, surface) ! version avec vocation des arguments
  IMPLICIT NONE
  REAL, INTENT(IN)  :: base, hauteur ! déclaration des arguments d'entrée
  REAL, INTENT(OUT) :: surface      ! déclaration des arguments de sortie
  base = base / 2.              ! affectation de base interdite ici
  surface = base * hauteur           ! forme maladroite du calcul de surface
END SUBROUTINE aire

```

La vocation `INTENT(INOUT)` pour un argument modifiable, n'impose pas de contrainte dans la procédure, mais permet au compilateur d'imposer l'emploi d'une variable comme argument effectif alors qu'une expression est acceptée dans le cas d'un argument à vocation `INTENT(IN)` non modifiable.

6.2 Sous-programmes et fonctions

On distingue deux sortes de procédures :

- Les *sous-programmes* (*subroutine*) sont constitués d'une suite d'instructions qui effectuent une tâche déterminée quand on les appelle depuis une autre unité de programme via l'instruction `CALL` et rendent le contrôle à l'unité appelante après exécution. Des paramètres ou arguments peuvent être échangés entre un sous-programme et l'unité appelante.
- Les *fonctions* (*function*) sont des procédures qui renvoient un résultat sous leur nom, résultat qui est ensuite utilisé dans une expression au sein de l'unité de programme appelante : la simple mention de la fonction (avec ses arguments) dans l'expression provoque l'exécution des instructions que comprend la procédure. Cette notion s'apparente ainsi à celle de fonction au sens mathématique du terme.

Remarques :

- Une fonction pourrait être considérée comme un cas particulier d'un sous-programme, qui renverrait un résultat accessible sous le nom de la fonction. On peut donc en général transformer une fonction en sous-programme, quitte à utiliser un argument supplémentaire pour le résultat et une ou éventuellement plusieurs variables supplémentaires dans l'appelant, si plusieurs appels à la fonction coexistent dans une même expression.
- À l'inverse, s'il s'agit d'effectuer une action qui n'est pas un simple calcul et par exemple modifier les arguments d'appel, on préfère le sous-programme. Parallèlement, il est d'usage de ← ♡

ne pas modifier les paramètres d'appel d'une fonction, considérés comme arguments d'entrée (cf. 6.1.4, p. 52).

6.2.1 Sous-programmes

Un sous-programme est introduit par l'instruction `SUBROUTINE`, suivie du nom du sous-programme et de la liste éventuelle de ses arguments muets entre parenthèses et séparés par des virgules, et délimité par l'instruction `END`⁷, éventuellement suivie de `SUBROUTINE`, éventuellement suivi du nom du sous-programme⁸.

```
SUBROUTINE <nom_sous_programme> [( <arg_1>, <arg_2>, ... )]
...
[RETURN]
END [SUBROUTINE [ <nom_sous_programme> ] ]
```

Par exemple, le sous-programme `aire`,

```
SUBROUTINE aire(base, hauteur, surface) ! arguments muets
  IMPLICIT NONE
  REAL, INTENT(IN)  :: base, hauteur ! déclaration des arguments d'entrée
  REAL, INTENT(OUT) :: surface      ! déclaration des arguments de sortie
  surface = base * hauteur / 2.      ! calcul de surface
END SUBROUTINE aire
```

peut être appelé par exemple via

```
...
REAL :: b1, h1, s1, b2, h2, s2
...
b1 = 30.
h1 = 2.
CALL aire(b1, h1, s1)           ! appel avec arguments effectifs de même type
PRINT *, ' Surface = ', s1     ! s1 = 30.
b2 = 2.
h2 = 5.
CALL aire(b2, h2, s2)           ! appel avec arguments effectifs de même type
PRINT *, ' Surface = ', s2     ! s2 = 5.
...
```

6.2.2 Fonctions

Une fonction est encadrée par les déclarations `FUNCTION`, suivie du nom de la fonction et de la liste éventuelle de ses arguments muets et l'instruction `END`⁹, éventuellement suivie de `FUNCTION`, éventuellement suivie du nom de la fonction¹⁰. Le type du résultat peut être soit déclaré dans l'entête, comme préfixe de `FUNCTION`, soit déclaré en même temps que les arguments, mais il ne faut pas lui attribuer une vocation `INTENT(OUT)`. La fonction doit comporter une ou plusieurs instructions qui affectent une valeur au résultat.

7. On peut insérer `RETURN` avant `END`, cf. 4.4.4, p. 28.

8. Pour améliorer la lisibilité des sources, on utilisera systématiquement cette possibilité de délimiter clairement les procédures au lieu de se contenter de les terminer toutes par `END`.

9. On peut insérer `RETURN` avant `END`, cf. 4.4.4, p. 28.

10. Pour améliorer la lisibilité des sources, on utilisera systématiquement cette possibilité de délimiter clairement les procédures au lieu de se contenter de les terminer toutes par `END`.


```
[<type>] FUNCTION <nom_de_la_fonction> [(<arg_1>, <arg_2>, ...)]
...
[RETURN]
END [FUNCTION [<nom_de_la_fonction>] ]
```

Par exemple, le sous-programme `aire`, n'ayant qu'un argument de retour, peut être traduit en fonction ¹¹ :

```
FUNCTION surf(base, hauteur)          ! arguments muets
  IMPLICIT NONE
  REAL          :: surf                ! déclaration du résultat
  REAL, INTENT(IN) :: base, hauteur    ! déclaration des arguments muets
  surf = base * hauteur / 2.          ! affectation du résultat
END FUNCTION surf
```

qui peut être utilisée de la manière suivante :

```
...
REAL :: b1, h1, b2, h2, somme
...
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1) ! appel avec arguments effectifs
b2 = 2.
h2 = 5.
somme = surf(b1, h1) + surf(b2, h2) ! deux appels dans la même expression
...                               ! somme = 30. + 5. = 35.
```

6.3 Procédures internes et procédures externes

Les procédures appelées (sous-programmes et fonctions) peuvent être :

- soit incluses dans le programme appelant, appelé aussi *hôte* (host) de la procédure et qualifiées de *procédures internes* (internal procedure); elles sont alors introduites par la déclaration `CONTAINS`;
- soit extérieures au programme appelant (comme en fortran 77) et qualifiées alors de *procédures externes* (external procedure).

6.3.1 Procédures internes : CONTAINS

Les procédures internes sont compilées en même temps que la procédure qui les appelle, ce qui permet un contrôle de la cohérence des arguments. Mais elles se prêtent peu à une réutilisation : on les réservera à des procédures très dépendantes de leur procédure hôte. Un seul niveau d'imbrication est possible : une seule instruction `CONTAINS` est possible dans une procédure ¹².

```
PROGRAM principal
...
CALL sub1(...)          ! appel du sous-programme interne sub1
...
x = fonc1(...)         ! appel de la fonction interne fonc1
```

¹¹. On aurait aussi pu écrire `REAL FUNCTION surf(base, hauteur)` et éviter la déclaration explicite du type du résultat.

¹². Sauf en ce qui concerne les procédures de module, qui peuvent inclure une procédure interne.

```

...
CONTAINS                                ! introduit les procédures internes
  FUNCTION fonc1(...)
  ...
  END FUNCTION fonc1
  SUBROUTINE sub1(...)
  ...
  END SUBROUTINE sub1
END PROGRAM principal

```

Dans une procédure interne, toutes les entités déclarées au sein de la procédure hôte sont a priori accessibles, sauf si on les redéclare localement¹³, donc les communications entre l'hôte et la procédure interne ne nécessitent pas forcément l'emploi d'arguments.

Voici un exemple mettant à profit le caractère global des variables, au détriment de la lisibilité :

```

1 PROGRAM principal                        ! version sans passage d'arguments
2 IMPLICIT NONE
3 REAL :: base, aire, surface             ! variables globales
4 base = 30.
5 hauteur = 2.
6 CALL aire
7 PRINT *, ' Surface = ', surface         ! surface = 30.
8 base = 2.
9 hauteur = 5.
10 CALL aire
11 PRINT *, ' Surface = ', surface        ! surface = 5.
12 CONTAINS
13   SUBROUTINE aire                       ! version déconseillée
14     ! ne pas déclarer base, hauteur et surface (variables globales)
15     ! sous peine de créer des variables locales distinctes !!!
16     surface = base * hauteur / 2.      ! calcul de surface
17   END SUBROUTINE aire
18 END PROGRAM principal

```

Le simple ajout d'une déclaration du type `REAL :: base` dans le sous-programme `aire` créerait une variable locale `base`, empêchant toute visibilité de la variable `base` du programme principal et rendant le sous-programme inutilisable : on dit que la variable locale *masque* la variable globale.

♥ ⇒ Dans un souci de clarté et de portabilité, on évitera de tirer parti de la visibilité des variables globales¹⁴ ; au contraire, on transmettra explicitement en arguments les variables partagées et on donnera aux variables locales à la procédure interne des noms non utilisés par la procédure hôte.

6.3.2 Procédures externes

Seules les procédures externes se prêtent à la réutilisation du code, mais nous verrons que le contexte le plus fiable de leur utilisation est celui des modules (*cf.* 6.4, p. 58). Qu'elles soient placées dans des fichiers séparés, ou dans le même fichier que l'appelant, leur compilation, hors contexte des modules, se déroule de façon indépendante. Les communications entre procédure externe et les procédures appelantes se font au travers des arguments transmis.

Les procédures écrites dans d'autres langages sont évidemment des procédures externes.

13. Si on déclare une variable *locale* dans une procédure interne avec le même nom qu'une variable de portée globale, la déclaration locale masque la variable globale.

14. En particulier, si une procédure interne a été écrite en s'appuyant sur cette lisibilité, elle ne sera pas aisément transformable en procédure externe par la suite.

6.3.3 La notion d'interface

Dans le cas d'une procédure externe, la compilation séparée de la procédure appelée et de la procédure appellante ne permet pas au compilateur de connaître, sans information complémentaire, le type des arguments échangés et de vérifier la cohérence entre les arguments effectifs lors de l'appel et les arguments muets déclarés. Il est cependant possible d'assurer une visibilité de ces déclarations d'arguments à l'aide d'une *interface explicite* (explicit interface) insérée dans la ou les procédures appellantes¹⁵. Les déclarations des arguments sont ainsi communiquées aux appelants à l'intérieur d'un bloc d'interface¹⁶, délimité par les instructions `INTERFACE` et `END INTERFACE`¹⁷ et inséré après les déclarations de variables de l'appelant.

```

INTERFACE                                ! début de bloc d'interface
  SUBROUTINE sub1(...)
    ... déclaration des arguments de sub1
  END SUBROUTINE sub1
  SUBROUTINE sub2(...)
    ... déclaration des arguments de sub2
  END SUBROUTINE sub2
END INTERFACE                            ! fin de bloc d'interface

```

Par exemple, avec la fonction externe `surf`,

```

FUNCTION surf(base, hauteur)
  IMPLICIT NONE
  REAL          :: surf          ! déclaration du résultat
  REAL, INTENT(IN) :: base, hauteur ! déclaration des arguments muets
  surf = base * hauteur / 2.     ! calcul de surface
END FUNCTION surf

```

on doit préciser dans la procédure appellante :

```

...
REAL :: b1, h1
...
INTERFACE
  FUNCTION surf(base, hauteur)
    IMPLICIT NONE
    REAL          :: surf          ! déclaration du résultat
    REAL, INTENT(IN) :: base, hauteur ! déclaration des arguments muets
  END FUNCTION surf
END INTERFACE
...
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1) ! arguments effectifs de même type
...

```

L'inconvénient immédiat de l'interface explicite est que la déclaration des arguments doit être fidèlement dupliquée entre la procédure appelée et la ou les interfaces dans la ou les procédures appellantes. Lors des mises à jour du code, il est possible de créer des incohérences entre ces déclarations et cette technique est donc déconseillée. Seule l'insertion des procédures externes dans des modules (cf. 6.4, p. 58) fournit une solution satisfaisante à cette question.

15. Ces déclarations d'interface du fortran 90 sont à rapprocher des prototypes des fonctions en langage C.

16. Un seul bloc d'interface est possible dans une procédure, sauf si on déclare les interfaces via des modules (cf. 6.4, p. 58).

17. S'il est possible de nommer un bloc d'interface, son nom ne peut pas être répété sur l'instruction `END INTERFACE`.

6.4 Les modules

Les *modules* (modules) sont des entités compilables séparément (mais non exécutables) permettant d'héberger des éléments de code ou de données qui ont vocation à être utilisés par plusieurs procédures sans nécessité de dupliquer l'information. Un module est délimité par les instructions :

MODULE *<nom_de_module>* et END MODULE *<nom_de_module>*. Il peut comporter :

- des déclarations de variables¹⁸,
- un bloc d'interface
- et des procédures dites de module (*module procedure*) introduites par l'instruction CONTAINS.

L'instruction USE *<nom_de_module>*, placée *avant* toutes les déclarations, assure à l'unité de programme où elle est placée la visibilité de toutes les entités *publiques*¹⁹ (variables, types, procédures, blocs d'interface) du module²⁰.

La compilation du module produit un fichier de module²¹, dont le nom est celui du module et le suffixe généralement *.mod*²². Ce fichier sera exploité par l'instruction USE lors de la compilation des modules appelants afin d'effectuer notamment les contrôles inter-procéduraux de respect d'interface (type, vocation des arguments, ...) : il est donc nécessaire de compiler le ou les modules *avant* les procédures qui les utilisent.

6.4.1 Exemple de procédure de module

Par exemple, le module `surface` contient la procédure interne de module `surf`.

```

MODULE m_mod
CONTAINS
  FUNCTION surf(base, hauteur)
    IMPLICIT NONE
    REAL                :: surf           ! déclaration du résultat
    REAL, INTENT(IN)    :: base, hauteur  ! déclaration des arguments muets
    surf = base * hauteur / 2.           ! calcul de surface
  END FUNCTION surf
END MODULE m_mod

```

La visibilité du module `m_mod` et donc de sa procédure interne `surf`, est assurée dans une procédure appelante par l'instruction USE `m_mod` :

```

...
USE m_mod ! assure la visibilité du module sans duplication de code
IMPLICIT NONE
REAL :: b1, h1
...
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1) ! arguments effectifs de même type
...

```

18. Un module peut aussi héberger des déclarations de types dérivés, cf. 9.6, page 95.

19. cf. 6.4.3 p. 61

20. L'instruction USE peut elle-même être utilisée dans un module, permettant ainsi l'imbrication des modules.

21. La structure de ces fichiers dépend du compilateur et, pour g95 par exemple, plusieurs versions de format de ces fichiers incompatibles entre elles ont existé.

22. On peut comparer ce fichier *.mod* aux fichiers d'entête (header) des fonctions en langage C et l'instruction USE à la directive `#include`. Mais c'est le compilateur qui produit automatiquement le fichier *.mod* en fortran alors qu'en C, le fichier d'entête doit être écrit par l'utilisateur qui doit aussi le maintenir en conformité avec le code de la procédure à chaque mise à jour de la liste des paramètres. De plus, c'est le compilateur qui interprète l'instruction USE alors qu'une directive `#include` est traitée par le préprocesseur. D'ailleurs, il est possible d'utiliser le préprocesseur de la même façon en fortran qu'en C pour inclure des parties de code commun à plusieurs unités de programme.

Noter que cette inclusion de code pourrait aussi se faire via une directive de compilation `INCLUDE`, qui fait partie de la norme du fortran 90, et que certains compilateurs ont intégrée comme instruction fortran (hors norme cependant), mais cette méthode est actuellement déconseillée au profit de l'emploi de l'instruction USE.

La compilation du module `m_mod` crée un fichier de module `m_mod.mod` qui sera consulté par le compilateur lorsqu'il rencontrera l'instruction `USE m_mod` et qui assurera aux unités de programmes qui utilisent le module la visibilité de ce qu'il définit (déclarations, interfaces, ...).

Si on place le module `m_mod` dans un fichier source séparé `f_modul.f90`, il faut le compiler (avant les unités de programme qui lui font appel), via la commande `g95 -c f_modul.f90`, qui créera deux fichiers (cf. Figure 6.1, p. 59) :

- un fichier de module de suffixe `.mod` : `m_mod.mod`²³ destiné au compilateur ;
- un fichier objet de suffixe `.o` : `f_modul.o` qui sera utilisé par l'éditeur de liens pour construire l'exécutable à partir des objets (via la commande `g95 f_ppal.o f_modul.o -o f_ppal.x`).

Une autre méthode consisterait à lancer une seule commande

```
g95 f_modul.f90 f_ppal.f90 -o f_ppal.x
```

(en compilant le module avant l'appelant) qui déclencherait la compilation, puis l'édition de liens.

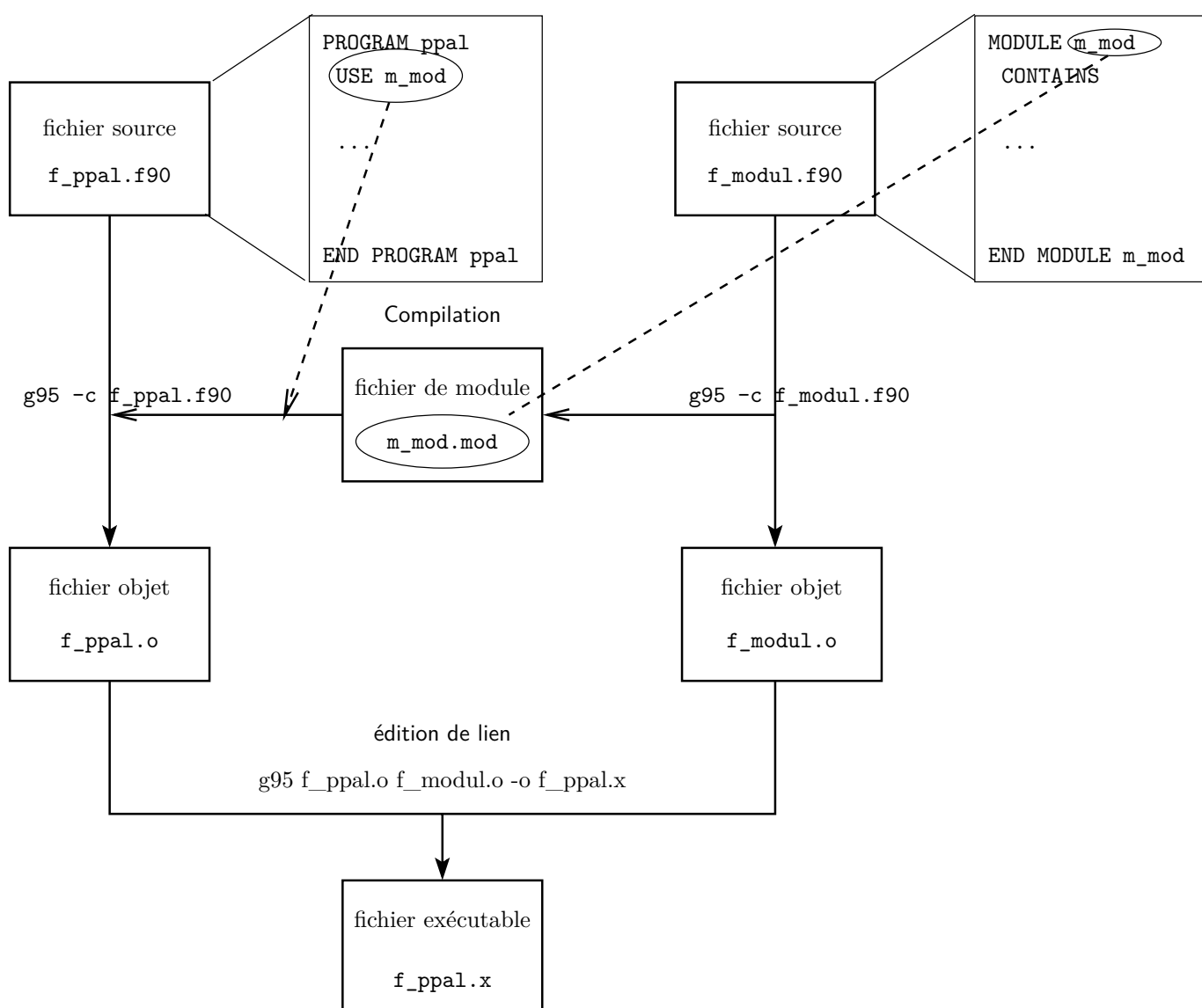


FIGURE 6.1 – Modules et fichiers impliqués en compilation séparée

23. Son nom ne dépend pas du nom du fichier source.

6.4.2 Partage de données via des modules

La déclaration de variables partageables par plusieurs unités de programme, équivalent du `COMMON` du fortran 77, est possible à l'intérieur d'un module. Chaque unité de programme qui attachera ce module par l'instruction `USE` partagera alors ces variables sauf si leur visibilité a été restreinte par l'attribut `PRIVATE`²⁴.

Les données d'un module ne sont pas a priori permanentes sauf si :

- leur portée s'étend au programme principal via un `USE`;
- leur portée a été étendue à plus d'une unité de compilation via au moins deux `USE`;
- l'attribut `SAVE` a été précisé explicitement ou impliqué par une initialisation (*cf.* 2.5, p. 14).

```

MODULE constantes
  IMPLICIT NONE
  REAL, SAVE :: pi          ! variable statique
END MODULE constantes
MODULE initialisation
  USE constantes            ! permet la visibilité de la variable pi
  CONTAINS
    SUBROUTINE init         ! sans argument
      pi = 4. * atan(1.)    ! mais pi est déclaré dans le module constantes
    END SUBROUTINE init
END MODULE initialisation
PROGRAM principal
  ! USE constantes ! inutile car appelé par le module initialisation
  USE initialisation       ! permet l'appel à la procédure de module init
  IMPLICIT NONE
  REAL :: y, x
  CALL init
  ...
  y = sin ( pi * x )
  ...
  STOP
END PROGRAM principal

```

Cette technique de partage des informations entre procédures est puissante, mais elle n'est pas aussi explicite que le passage par argument qui reste la méthode à privilégier. Elle reste cependant incontournable dans le cas d'appels de procédures imbriqués où la procédure intermédiaire ne peut pas voir les données à transmettre²⁵ : un exemple classique est l'application d'une procédure fonctionnelle (intégration, dérivation, ...) à une fonction d'une variable, qui possède des paramètres que la procédure fonctionnelle ignore alors qu'elle effectue des appels à la fonction. Le seul moyen de passer ces paramètres est de les partager via un module.

Dans l'exemple suivant, introduit à propos du passage d'argument de type procédure (*cf.* 6.5.3, p. 64), le paramètre `puissance` de la fonction effective `fonct` à intégrer ne peut pas être passé à la méthode d'intégration `integr`, mais doit être communiqué de l'appelant (ici le programme principal) à la fonction `fonct` à évaluer.

1	<code>MODULE mod_fonct</code>	<code>! module définissant la fct effective</code>
2	<code>IMPLICIT NONE</code>	
3	<code>INTEGER :: puissance</code>	<code>! masqué à la procédure d'intégration</code>
4	<code>CONTAINS</code>	
5	<code>FUNCTION fonct(x)</code>	<code>! définition de la fonction effective</code>
6	<code>REAL, INTENT(IN)</code>	<code>:: x</code>
7	<code>REAL</code>	<code>:: fonct</code>
8	<code>fonct = x**puissance</code>	<code>! paramétré par puissance par exemple</code>

24. *cf.* 6.4.3 p. 61

25. La transmission des données s'effectue à l'insu de la procédure intermédiaire : celle qui l'appelle et celle qu'elle appelle se partagent les données grâce à un `USE` commun inconnue de la procédure intermédiaire!

```

 9  END FUNCTION fonct
10  END MODULE mod_fonct
11  !
12  MODULE mod_integr                ! module définissant le sous-programme
13  IMPLICIT NONE
14  CONTAINS
15  SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! f = argument muet
16  REAL, INTENT(IN)                :: xmin, xmax
17  REAL, INTENT(OUT)               :: somme
18  INTEGER, INTENT(IN)             :: n_points
19  INTERFACE                       ! interface de la fonction muette
20  FUNCTION f(x)
21  REAL, INTENT(IN)                :: x
22  REAL                            :: f
23  END FUNCTION f
24  END INTERFACE
25  REAL                            :: x, pas
26  INTEGER                          :: i
27  pas = (xmax - xmin) / n_points
28  somme = pas * (f(xmin) + f(xmax)) / 2.
29  DO i = 2, n_points
30  x = xmin + (i - 1) * pas
31  somme = somme + pas * f(x)
32  END DO
33  END SUBROUTINE integr
34  END MODULE mod_integr
35  PROGRAM principal
36  USE mod_fonct                    ! accès au module de la fonction effective
37  USE mod_integr                   ! accès au module du sous-programme
38  IMPLICIT NONE
39  REAL    :: x0 = 0., x1 = 1., s
40  INTEGER :: n = 100
41  puissance = 1                    ! ne pas redéclarer: visibilité assurée par use
42  CALL integr(fonct, x0, x1, s, n)
43  WRITE(*,*) 'somme de f(x) = x : ', s
44  puissance = 2                    ! partagé avec fonct, mais inconnu de integr
45  CALL integr(fonct, x0, x1, s, n)
46  WRITE(*,*) 'somme de f(x) = x*x : ', s
47  END PROGRAM principal

```

6.4.3 Éléments d'encapsulation

Les modules permettent de faciliter la réutilisation des codes et leur maintenance. Au sein d'une procédure que l'on souhaite partager, on peut distinguer :

- l'interface (qui doit rester parfaitement stable), qui doit être visible de ses procédures appelantes et doit donc être publique ;
- les détails de l'implémentation (qui peuvent évoluer) qu'il est souvent prudent de masquer aux utilisateurs, en les rendant privés.

Par défaut, les entités (types dérivés, variables, procédures) définies dans un module `mon_module` sont publiques et donc rendues accessibles (et, pour les variables, modifiables) par l'instruction `USE mon_module`. À l'intérieur d'un module, on peut utiliser la déclaration `PRIVATE` pour masquer toutes les entités qui suivent cette déclaration en dehors du module où elles sont déclarées. On dispose aussi des attributs `PUBLIC` et `PRIVATE` pour spécifier plus finement la visibilité des objets lors de leur déclaration.

Par ailleurs, on peut aussi, dans la procédure appelante, restreindre la visibilité des objets publics d'un module en spécifiant la liste exhaustive des objets auxquels on souhaite accéder :
`USE nom_de_module, ONLY: liste_d'objets`

De plus, lorsque l'on réutilise des modules rédigés dans un autre contexte, on peut être confronté à des conflits de nommage des identificateurs. L'instruction `USE` permet le renommage des entités publiques du module qu'elle appelle, selon la syntaxe :

```
USE nom_de_module, nom_local => nom_dans_le_module
```

Ce renommage permet aussi une sorte de paramétrage au niveau de la compilation (*cf.* l'exemple 2.5.4 p. 16).

f2003 ⇒ Enfin, sans restreindre la visibilité d'une variable, on souhaite parfois réserver aux procédures du module où elles sont déclarées le droit de modifier leur valeur. La norme fortran 2003 a introduit l'attribut `PROTECTED` dans cet objectif. Le programme suivant :

```

1  MODULE m_protege
2  IMPLICIT NONE
3  INTEGER, PRIVATE   :: m ! non visible en dehors du module
4  INTEGER, PUBLIC   :: n ! visible et modifiable en dehors du module
5  INTEGER, PROTECTED :: p ! visible mais non modifiable en dehors du module
6  ! => oblige à passer par une des procédures du module pour le modifier
7  CONTAINS
8  SUBROUTINE sub1 ! pas de paramètres : m, n et p ont une portée suffisante
9      m = 1
10     n = 2
11     p = 3
12     WRITE(*,*) "m, n, p dans sub1 ", m, n, p
13  END SUBROUTINE sub1
14  SUBROUTINE sub2 ! pas de paramètres : m, n et p ont une portée suffisante
15     m = 10
16     n = 20
17     p = 30
18     WRITE(*,*) "m, n, p dans sub2 ", m, n, p
19  END SUBROUTINE sub2
20  END MODULE m_protege
21
22  PROGRAM t_protege
23  USE m_protege
24  CALL sub1
25  WRITE(*,*) "n, p dans l'appelant après appel de sub1 ", n, p
26  CALL sub2
27  WRITE(*,*) "n, p dans l'appelant après appel de sub2 ", n, p
28  ! m = -1 ! serait incorrect car m n'est pas visible ici (privé)
29  n = -2 ! autorisé car n est public dans le module
30  ! p = -3 ! serait incorrect car p est protégé => non modifiable hors du module
31  ! à comparer avec l'attribut intent(in) mais ici dans l'appelant
32  WRITE(*,*) "n, p dans l'appelant après affectation de n ", n, p
33  END PROGRAM t_protege

```

produit l'affichage :

```

m, n, p dans sub1  1 2 3
n, p dans l'appelant après appel de sub1  2 3
m, n, p dans sub2  10 20 30
n, p dans l'appelant après appel de sub2  20 30
n, p dans l'appelant après affectation de n  -2 30

```

6.4.4 L'instruction `IMPORT` dans les interfaces

f2003 ⇒

Contrairement aux procédures, en fortran 95, les interfaces encapsulées dans un module n'ont pas de visibilité sur les entités déclarées dans le module qui les héberge. Le fortran 2003 a introduit l'instruction `IMPORT` qui permet dans une interface, d'accéder aux entités du module soit globalement si on ne spécifie pas de liste, soit seulement à celles spécifiées par la liste.

6.4.5 Modules intrinsèques

← f2003

La norme 2003 du fortran a introduit plusieurs *modules intrinsèques* dont IEEE_ARITHMETIC, IEEE_EXCEPTIONS, et IEEE_FEATURES pour les questions de calcul numérique en flottant, ainsi que ISO_FORTRAN_ENV, pour l'interaction avec l'environnement système en particulier les entrées-sorties (cf. 5.3.1, p. 36) et ISO_C_BINDING, pour l'inter-opérabilité avec le langage C (cf. chap. 12). On peut accéder à ces modules avec l'instruction USE, INTRINSIC :: en priorité sur un module utilisateur homonyme.

6.5 Fonctionnalités avancées

6.5.1 Arguments optionnels

Lors de la déclaration d'une procédure, il est possible de préciser que certains arguments pourront être omis lors de certains appels, en leur spécifiant l'attribut OPTIONAL. Sauf à utiliser un passage d'arguments par mot-clef (cf. 6.5.2, p. 64), les arguments optionnels doivent être placés en fin de liste²⁶. Lors de l'écriture de la procédure, il faut prévoir un traitement conditionnel suivant que l'argument a été ou non fourni lors de l'appel; la fonction intrinsèque d'interrogation PRESENT(<arg>), de résultat booléen, permet de tester si l'argument a été effectivement passé.

Par exemple, le sous-programme `integr` de calcul d'intégrale d'une fonction (pour le moment fixée et non paramétrée) sur un intervalle [`xmin`, `xmax`] passé en argument, comporte deux paramètres optionnels : le nombre de points d'évaluation (`n_points`) et la précision relative (`precision`) demandée pour le calcul.

```
PROGRAM principal
...
CALL integr(xmin1, xmax1, somme1, .1, 100) ! tous les arguments sont passés
CALL integr(xmin2, xmax2, somme2, .1)      ! n_points est omis
...
CONTAINS
  SUBROUTINE integr(xmin, xmax, somme, precision, n_points)
    REAL, INTENT(IN)           :: xmin, xmax ! arguments d'entrée requis
    REAL, INTENT(OUT)          :: somme      ! argument de sortie requis
    REAL, INTENT(IN), OPTIONAL :: precision ! argument d'entrée optionnel
    INTEGER, INTENT(IN), OPTIONAL :: n_points ! argument d'entrée optionnel
    INTEGER                    :: points    ! variable locale indispensable
    ...
    IF( PRESENT(n_points) ) THEN
      points = n_points          ! utiliser la valeur transmise
    ELSE
      points = 200              ! valeur par défaut
    END IF
    ...
    somme = ...
  END SUBROUTINE integr
END PROGRAM principal
```

Remarques : dans cet exemple, la variable locale `points` est nécessaire, car `n_points`, déclaré comme argument d'entrée (INTENT(IN)), ne peut pas être modifié dans la procédure. Noter par ailleurs, qu'il faut conditionner toute référence au paramètre optionnel à sa fourniture effective et que la formulation suivante

```
IF( PRESENT(n_points) .AND. n_points > 1) points = n_points
```

²⁶. Si tous les arguments sont omis, l'appel se fera sous la forme `CALL sub()` pour un sous-programme et `fct()` pour une fonction.

est incorrecte car elle suppose que le second opérande du `.AND.` n'est pas évalué (cf. 3.3, p. 21) si le paramètre optionnel n'a pas été spécifié.

6.5.2 Transmission d'arguments par mot-clef

Dès le fortran 77, certaines procédures intrinsèques comme l'instruction `OPEN` par exemple autorisaient le passage d'argument par mot-clef, ce qui permettait de s'affranchir de l'ordre des arguments. Cependant le passage d'argument aux procédures définies par l'utilisateur restait nécessairement positionnel.

En fortran 90, on peut passer certains arguments par un mot-clef, en utilisant la syntaxe `<argument_muet> = <argument_effectif>`. Cette possibilité prend tout son sens quand la procédure possède des arguments optionnels. En effet, dans ce cas, le passage par mot-clef est le seul moyen de fournir un argument qui suit un argument optionnel non spécifié lors de l'appel²⁷. Plus précisément, dès qu'un argument optionnel a été omis, les arguments qui le suivent ne peuvent plus être passés de façon positionnelle.

En reprenant l'exemple précédant (cf. 6.5.2, p. 64), on peut mixer les modes de passage d'arguments :

```

...
CALL integr(0., 1., precision=.1, n_points=100, somme=resultat)
!
! plus de passage positionnel à partir du troisième
! tous les arguments passés, mais en ordre modifié avec les mots-clefs
CALL integr(xmin2, xmax2, somme2, n_points=500)
! precision est omis, mais on veut passer n_points, donc par mot-clef car après
...

```

6.5.3 Noms de procédures passés en argument

La paramétrisation des sous-programmes conduit parfois à transmettre le nom d'une procédure en argument d'une autre procédure. Par exemple, un sous-programme d'intégration numérique `integr` doit accepter en argument le nom de la fonction à intégrer. Dans le sous-programme d'intégration, l'identificateur (muet) `f` de la fonction est nécessairement vu comme une fonction dès que le sous-programme invoque cette fonction. Mais, vu du programme appelant `principal`, il est nécessaire d'indiquer explicitement que l'argument *effectif* fournissant le nom de la fonction à intégrer n'est pas une simple variable.

- En fortran 77, on informe le compilateur du caractère particulier de cet argument, grâce à l'instruction `EXTERNAL <fonc>`²⁸, où `<fonc>` est la fonction à intégrer, insérée dans la procédure appelante²⁹.
- En fortran 90, il vaut mieux assurer la visibilité de l'interface de la fonction effectivement intégrée, à l'appelant, ici le programme `principal`. Cela est possible soit en déclarant dans l'appelant l'interface explicite de la fonction `fonct`, soit en créant un module `mod_fonct` avec cette fonction `fonct` en procédure interne de module et en déclarant `USE mod_fonct` dans le programme `principal`.

Le premier exemple présente une version minimale du programme `principal` avec des procédures `integr` et `fonct` externes. Seule l'interface explicite de la fonction effective `fonct` (lignes 7 à 11) est nécessaire dans le programme `principal`.

```

1  ! version avec procédures externes et interface minimale
2  PROGRAM principal
3    IMPLICIT NONE
4    REAL :: x0 = 0., x1 = 1., s

```

27. Un appel du type `f(x1, , x3)` n'est pas autorisé en fortran.

28. Sauf dans le cas très rare où la fonction utilisée est une fonction intrinsèque du fortran ; alors il faut déclarer `INTRINSIC <fonc>`

29. Si `IMPLICIT NONE` a été déclaré dans l'appelant, il faut aussi déclarer le type de la procédure s'il s'agit d'une fonction.

```

5  INTEGER :: n = 100
6  INTERFACE
7      FUNCTION fonct(x)                ! interface obligatoire de la fonction effective
8          IMPLICIT NONE
9          REAL, INTENT(IN)             :: x
10         REAL                         :: fonct
11     END FUNCTION fonct
12 END INTERFACE
13 CALL integr(fonct, x0, x1, s, n)
14 WRITE(*,*) 'somme = ', s
15 END PROGRAM principal
16 ! sous-programme externe
17 SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! f = argument muet
18     IMPLICIT NONE
19     REAL, INTENT(IN)                :: xmin, xmax
20     REAL, INTENT(OUT)               :: somme
21     INTEGER, INTENT(IN)              :: n_points
22     ! déclaration de f nécessaire, sauf si l'interface de f est connue
23     REAL                             :: f
24     REAL                             :: x, pas
25     INTEGER                          :: i
26     pas = (xmax - xmin) / n_points
27     somme = pas * (f(xmin) + f(xmax)) / 2.
28     DO i = 2, n_points
29         x = xmin + (i - 1) * pas
30         somme = somme + pas * f(x)
31     END DO
32 END SUBROUTINE integr
33 ! fonction externe
34 FUNCTION fonct(x)                   ! définition de la fonction effective
35     IMPLICIT NONE
36     REAL, INTENT(IN)                :: x
37     REAL                             :: fonct
38     fonct = x                        ! par exemple
39 END FUNCTION fonct

```

L'exemple suivant présente encore une version du programme `principal` avec des procédures externes, mais qui comporte aussi les interfaces explicites du sous-programme `integr` (lignes 12 à 18) dans le programme principal et de la fonction symbolique `f` (lignes 29 à 34) au sein du sous-programme `integr`. Ces deux interfaces sont facultatives ici, mais vivement conseillées.

```

1  ! version avec procédures externes et interfaces
2  PROGRAM principal
3      IMPLICIT NONE
4      REAL :: x0 = 0., x1 = 1., s
5      INTEGER :: n = 100
6      INTERFACE
7          FUNCTION fonct(x)            ! interface obligatoire de la fonction effective
8              IMPLICIT NONE
9              REAL, INTENT(IN)         :: x
10             REAL                     :: fonct
11     END FUNCTION fonct
12     SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! interface facultative
13         IMPLICIT NONE
14         REAL, INTENT(IN)              :: xmin, xmax
15         REAL, INTENT(OUT)             :: somme
16         INTEGER, INTENT(IN)           :: n_points
17         REAL                          :: f
18     END SUBROUTINE integr
19 END INTERFACE
20 CALL integr(fonct, x0, x1, s, n)
21 WRITE(*,*) 'somme = ', s
22 END PROGRAM principal

```

```

23 ! sous-programme externe
24 SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! f = argument muet
25   IMPLICIT NONE
26   REAL, INTENT(IN)           :: xmin, xmax
27   REAL, INTENT(OUT)          :: somme
28   INTEGER, INTENT(IN)        :: n_points
29   INTERFACE
30     ! interface facultative de la fonction muette
31     FUNCTION f(x)
32       REAL, INTENT(IN)       :: x
33       REAL                   :: f
34     END FUNCTION f
35   END INTERFACE
36   REAL                       :: x, pas
37   INTEGER                     :: i
38   pas = (xmax - xmin) / n_points
39   somme = pas * (f(xmin) + f(xmax)) / 2.
40   DO i = 2, n_points
41     x = xmin + (i - 1) * pas
42     somme = somme + pas * f(x)
43   END DO
44 ! fonction externe
45 END SUBROUTINE integr
46 ! fonction externe
47 FUNCTION fonct(x)           ! définition de la fonction effective
48   IMPLICIT NONE
49   REAL, INTENT(IN)         :: x
50   REAL                     :: fonct
51   fonct = x                 ! par exemple
END FUNCTION fonct

```

Enfin, pour éviter la lourdeur des interfaces explicites, on préfère souvent intégrer les procédures dans des modules. Avec les modules `mod_fonct` (lignes 2 à 11) pour la fonction effective et `mod_integr` (lignes 13 à 38) pour le sous-programme, seule demeure l'interface explicite (lignes 23 à 28) de la fonction symbolique `f` (on peut la remplacer par une simple déclaration de `f`, ligne 21).

```

1 ! version avec modules
2 MODULE mod_fonct           ! module définissant la fct effective
3   CONTAINS
4     ! fonction interne de module
5     FUNCTION fonct(x)      ! définition de la fonction effective
6       IMPLICIT NONE
7       REAL, INTENT(IN)    :: x
8       REAL                :: fonct
9       fonct = x           ! par exemple
10    END FUNCTION fonct
11 END MODULE mod_fonct
12 !
13 MODULE mod_integr         ! module définissant le sous-programme
14   CONTAINS
15     ! sous-programme interne de module
16     SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! f = argument muet
17       IMPLICIT NONE
18       REAL, INTENT(IN)    :: xmin, xmax
19       REAL, INTENT(OUT)   :: somme
20       INTEGER, INTENT(IN) :: n_points
21       ! REAL               :: f
22       ! cette déclaration peut remplacer l'interface explicite de f qui suit
23       INTERFACE           ! interface conseillée de la fonction muette
24         FUNCTION f(x)
25           REAL, INTENT(IN) :: x
26           REAL             :: f
27         END FUNCTION f
28       END INTERFACE
29       REAL                 :: x, pas
30       INTEGER              :: i

```

```

31     pas = (xmax - xmin) / n_points
32     somme = pas * (f(xmin) + f(xmax)) / 2.
33     DO i = 2, n_points
34         x = xmin + (i - 1) * pas
35         somme = somme + pas * f(x)
36     END DO
37     END SUBROUTINE integr
38 END MODULE mod_integr
39 ! version du programme principal avec modules
40 PROGRAM principal
41     USE mod_fonct           ! chargement du module de la fonction effective
42     USE mod_integr         ! chargement du module du sous-programme
43     IMPLICIT NONE
44     REAL :: x0 = 0., x1 = 1., s
45     INTEGER :: n = 100
46     CALL integr(fonct, x0, x1, s, n)
47     WRITE(*,*) 'somme = ', s
48 END PROGRAM principal

```

6.5.4 Procédures récursives

Lorsqu'une procédure s'appelle elle-même, de façon directe ou indirecte, au travers d'une chaîne d'autres procédures, on la qualifie de récursive. Contrairement au fortran 77, le fortran 90 permet la récursivité directe ou indirecte des procédures, par la déclaration explicite via le mot-clef `RECURSIVE`. Mais, dans le cas d'une fonction récursive, le nom de la fonction ne peut plus être utilisé pour désigner le résultat : on doit alors introduire, après le mot-clef `RESULT`, le nom de la variable qui stockera le résultat. Dans une procédure récursive, une variable avec l'attribut `SAVE` (cf. 6.1.2, p. 52) est partagée par toutes les instances de la procédure, alors que ses variables automatiques sont spécifiques de chaque instance.

L'archétype de la *récursivité directe* est le calcul de la factorielle : $n! = n \times (n - 1)!$

```

INTEGER RECURSIVE FUNCTION fact(n) RESULT(factorielle)
! le type entier s'applique en fait au résultat : factorielle
IMPLICIT NONE
INTEGER, INTENT(IN)      :: n
IF ( n > 0 ) THEN
    factorielle = n * fact(n-1)    ! provoque un nouvel appel de fact
ELSE
    factorielle = 1
END IF
END FUNCTION fact

```

On peut aussi programmer la factorielle comme sous-programme :

```

RECURSIVE SUBROUTINE fact(n, factorielle)
IMPLICIT NONE
INTEGER, INTENT(IN)      :: n
INTEGER, INTENT(OUT)    :: factorielle
IF ( n > 0 ) THEN
    CALL fact(n-1, factorielle)  ! appel récursif
    factorielle = n * factorielle
ELSE
    factorielle = 1
END IF
END SUBROUTINE fact

```

On peut rencontrer la *réversivité indirecte* lors des calculs numériques d'intégrale double, si on utilise la même procédure `integr` (cf. 6.5.3, p. 64) pour intégrer selon chacune des variables : la fonction de y , $F(y) = \int_{x_0}^{x_1} f(x, y) dx$, fait appel, dans son évaluation numérique pour chaque valeur de y , à la procédure `integr`, et l'intégrale double sera évaluée en appliquant `integr` à $F(y)$. La procédure `integr` devra donc être déclarée récursive. De plus, il faudra masquer le paramètre y lors de l'intégration en x pour simuler le passage d'une fonction d'une seule variable à la procédure d'intégration : il sera transmis à F par partage via un module (cf. 6.4.2, p. 60).

```

1  MODULE mod_fonct                ! module définissant la fct effective
2  IMPLICIT NONE
3  REAL :: yy                      ! à y fixé
4  CONTAINS
5  FUNCTION fonct(x)              ! définition de la fonction effective de x
6  REAL, INTENT(IN)              :: x
7  REAL                          :: fonct
8  fonct = x**2 - yy**2
9  END FUNCTION fonct
10 END MODULE mod_fonct
11 !
12 MODULE mod_param_y             ! module des paramètres cachés à fonct
13 IMPLICIT NONE
14 REAL :: x0, x1                 ! bornes en x
15 INTEGER :: nx                   ! nombre de points en x
16 END MODULE mod_param_y
17 !
18 MODULE mod_integr              ! module définissant le sous-programme
19 IMPLICIT NONE
20 CONTAINS
21 SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! f = argument muet
22 REAL, INTENT(IN)              :: xmin, xmax ! bornes
23 REAL, INTENT(OUT)             :: somme      ! résultat
24 INTEGER, INTENT(IN)           :: n_points  ! nombre de points
25 INTERFACE
26   FUNCTION f(x)
27     REAL, INTENT(IN)          :: x
28     REAL                      :: f
29   END FUNCTION f
30 END INTERFACE
31 REAL                          :: x, pas
32 INTEGER                       :: i
33 pas = (xmax - xmin) / n_points
34 somme = pas * (f(xmin) + f(xmax)) / 2.
35 DO i = 2, n_points
36   x = xmin + (i - 1) * pas
37   somme = somme + pas * f(x)
38 END DO
39 END SUBROUTINE integr
40 END MODULE mod_integr
41
42 MODULE mod_fy
43 USE mod_param_y                ! module des paramètres cachés à fonct
44 USE mod_fonct, only: yy, fonct
45 USE mod_integr                 ! module définissant le sous-programme
46 IMPLICIT NONE
47 CONTAINS
48 FUNCTION fy(y)                 ! intégrale en x de la fonction effective
49 REAL                          :: fy
50 REAL, INTENT(IN)              :: y
51 REAL                          :: s
52 yy = y                         ! transfert de y à fonct via mod_fonct
53 CALL integr(fonct, x0, x1, s, nx) ! intégration en x à y fixé

```

```

54     fy = s
55     END FUNCTION fy
56 END MODULE mod_fy
57 !
58 PROGRAM principal
59     USE mod_param_y           ! accès aux paramètres utilisés par fy
60     USE mod_fy, only:x0, x1, fy, nx ! accès au module de la fct effective de y
61     USE mod_integr           ! accès au module d'intégration 1D
62     IMPLICIT NONE
63     REAL    :: y0 = 0., y1 = 1., s
64     INTEGER :: ny = 100
65     ! paramètres de l'intégration en x : passés via mod_fy (ne pas redéclarer)
66     x0 = 0.
67     x1 = 1.
68     nx = 50
69     CALL integr(fy, y0, y1, s, ny)
70     WRITE(*,*) 'somme de f(x, y) = x**2 -y**2 : ', s
71 END PROGRAM principal

```

f95/2003 6.5.5 Procédures pures

Dans certaines circonstances de parallélisation des codes, comme à l'intérieur d'une structure `FORALL` (cf. 7.4.6, p. 79), seules sont admises les procédures sans effet de bord. Un attribut `PURE` permet de certifier qu'une procédure est sans effet de bord, c'est à dire :

- qu'elle ne modifie pas de variables non locales : en particulier, s'il s'agit d'une fonction, tous ses arguments doivent posséder l'attribut `INTENT(IN)` ;
- qu'elle ne modifie pas de variables non locales soit visibles car la procédure est interne, soit rendues visibles par un `USE` ;
- qu'elle ne fait pas de lecture ou d'écriture sur un fichier externe : en particulier, elle ne peut rien lire au clavier ou afficher ;
- qu'elle ne contient pas de variable locale avec l'attribut `SAVE` : en particulier, pas de variable initialisée à la déclaration (cf. 2.5, p. 14) ;
- qu'elle ne comporte pas d'instruction `STOP` ;

Ces conditions imposent qu'une procédure pure n'appelle que des procédures pures. Toutes les procédures intrinsèques sont pures. Les procédures élémentaires (cf. 6.5.6, p. 69) possèdent automatiquement l'attribut `PURE`.

f95/2003 6.5.6 Procédures élémentaires

Une procédure est qualifiée d'élémentaire (attribut `ELEMENTAL`) si elle peut être appelée avec des arguments tableaux conformes de la même façon qu'avec des arguments scalaires. La majorité des procédures intrinsèques (cf. annexe A, p. 125) sont élémentaires.

Une procédure élémentaire est automatiquement pure, mais de plus tous ses arguments muets (ainsi que la valeur retournée pour une fonction) doivent être des scalaires sans l'attribut `POINTER`. Une procédure récursive (cf. 6.5.4, p. 67) ne peut pas être élémentaire. Une procédure élémentaire non-intrinsèque ne peut pas être passée en argument *effectif* d'une autre procédure.

```

ELEMENTAL REAL FUNCTION my_log(x)
! une fonction log protégée pour les arguments négatifs
REAL, INTENT(IN) :: x
IF ( x > 0 ) THEN
    my_log = LOG (x)
ELSE
    my_log = - 1000.
END IF
END FUNCTION MY_LOG

```

Chapitre 7

Tableaux

MCours.com

7.1 Généralités

Un *tableau* (array) est un ensemble « rectangulaire¹ » d'éléments de même type² (plus précisément de même variante de type), repérés au moyen d'indices entiers. Par opposition, on qualifie de *scalaire* un élément isolé. Les éléments d'un tableau sont rangés selon un ou plusieurs « axes » appelés *dimensions* du tableau. Dans les tableaux à une dimension (qui permettent de représenter par exemple des vecteurs au sens mathématique du terme), chaque élément est repéré par un seul entier, mais le fortran accepte des tableaux jusqu'à 7 dimensions³, où chaque élément est désigné par un 7-uplet d'entiers⁴.

7.1.1 Terminologie des tableaux

- *rang* (rank) d'un tableau : nombre de ses dimensions
- *étendue* (extent) d'un tableau selon une de ses dimensions : nombre d'éléments dans cette dimension
- *bornes* (bounds) d'un tableau selon une de ses dimensions : limites inférieure et supérieure des indices dans cette dimension. La borne inférieure par défaut vaut 1⁵.
- *profil* (shape) d'un tableau : vecteur dont les composantes sont les étendues du tableau selon ses dimensions ; sa taille est le rang du tableau.
- *taille* (size) d'un tableau : nombre total des éléments qui le constituent, c'est-à-dire le produit des éléments du vecteur que constitue son profil.
- deux tableaux sont dits *conformants* (conformable) s'ils ont le même profil

La déclaration d'un tableau s'effectue grâce à l'attribut `DIMENSION` qui indique le profil du tableau, mais aussi éventuellement les bornes, séparées par le symbole « : ».

<code>INTEGER, DIMENSION(5)</code>	<code>:: vecteur</code>	<code>! tableau à une dimension de 5 entiers</code> <code>! indicés de 1 à 5</code>
<code>REAL, DIMENSION(-2:2)</code>	<code>:: table</code>	<code>! tableau à une dimension de 5 réels</code> <code>! indicés de -2 à +2</code>
<code>REAL, DIMENSION(2,3)</code>	<code>:: m</code>	<code>! tableau à 2 dimensions de 2 x 3 = 6 réels</code> <code>! son profil a 2 éléments : 2 et 3</code>

1. L'assemblage « non-rectangulaire » d'éléments du même type est possible grâce aux pointeurs, cf. 11.3, p. 109.
2. L'assemblage d'éléments de types différents est possible au sein d'une structure, cf. chapitre 9, p. 92. Mais les éléments constitutifs d'un tableau peuvent être des structures, et ... les éléments de ces structures peuvent eux-mêmes comporter des tableaux !

3. Cette limite est portée à 15 en fortran 2008.

4. Certains termes employés en informatique à propos des tableaux peuvent avoir un sens profondément différent de leur sens en mathématiques dans le domaine des espaces vectoriels. C'est le cas, par exemple, de la notion de dimension : un vecteur à trois dimensions de la géométrie dans l'espace peut être représenté par un tableau monodimensionnel, de rang un et de taille trois.

5. Ne pas confondre avec le C où l'indexation des tableaux commence à zéro.

Les tableaux `vecteur` et `table` sont de dimension 1 et ont même profil : ils sont conformants.

Un élément (scalaire) d'un tableau est désigné en précisant entre parenthèses les indices de toutes ses dimensions, séparés par des virgules. `vecteur(2)`, `table(-1)` et `m(2,2)` sont des éléments des tableaux définis précédemment. Cependant, le fortran distingue un tableau monodimensionnel à un seul élément de l'élément qu'il contient :

```

INTEGER, DIMENSION(1)  :: t          ! tableau à un élément
INTEGER                 :: i          ! entier
i = t(1)                ! affectation autorisée
i = t                ! affectation illicite

```

7.1.2 Ordre des éléments dans un tableau multidimensionnel

Le caractère multidimensionnel des tableaux n'est pas respecté dans la mémoire des calculateurs, où les tableaux sont en général stockés dans une mémoire à un seul indice d'adressage. En fortran, l'ordre « classiquement » utilisé pour stocker les éléments des tableaux est celui où *le premier indice varie le plus vite*⁶, c'est à dire que deux éléments dont seul le premier indice diffère d'une unité sont contigus en mémoire.

Pour des tableaux de dimension 2, cela signifie que le rangement s'effectue colonne par colonne. Par exemple, la matrice m de 2 lignes par 3 colonnes, (au sens mathématique) est stockée dans le tableau `m` dimensionné par l'attribut `dimension(2,3)` de la manière suivante :

matrice

$$\begin{pmatrix} \boxed{1} & \boxed{3} & \boxed{5} \\ m_{11} & m_{12} & m_{13} \\ \boxed{2} & \boxed{4} & \boxed{6} \\ m_{21} & m_{22} & m_{23} \end{pmatrix}$$

tableau associé

1	2	3	4	5	6
m(1,1)	m(2,1)	m(1,2)	m(2,2)	m(1,3)	m(2,3)

Mais la norme fortran 90 ne spécifie pas l'ordre de rangement des éléments d'un tableau, laissé à la disposition du compilateur, entre autres pour des raisons d'efficacité de calcul suivant les processeurs utilisés : on ne pourra donc pas s'appuyer sur cet ordre de stockage pour le passage de tableaux multidimensionnels en argument de procédures⁷.

Cependant, cet ordre est respecté dans les instructions d'entrée-sortie et, par exemple, par la fonction `RESHAPE` de restructuration des tableaux. En particulier, une lecture globale de matrice carrée dans un fichier par un seul ordre `READ` stockera dans le tableau la *transposée* de la matrice écrite sous la forme habituelle dans le fichier.

```

INTEGER, DIMENSION(2,2)  :: matrice  ! matrice 2 lignes x 2 colonnes
...
OPEN(10, file='mat.dat', STATUS='old')  ! ouverture du fichier en lecture
READ(10, *) matrice(:, :)                ! lecture globale en format libre
WRITE(*, *) matrice(:, :)                ! affichage global en format libre
CLOSE(10)

```

Après les ordres précédents, et si le fichier `mat.dat` contient les deux lignes $\begin{matrix} 11 & 12 \\ 21 & 22 \end{matrix}$, le tableau

`matrice` contiendra : $\begin{bmatrix} 11 & 21 \\ 12 & 22 \end{bmatrix}$. En particulier `matrice(1,2) = 21` et `matrice(2,1) = 12`.

Mais l'affichage global en format libre reproduira la présentation du fichier initial, bien que le premier indice du tableau `matrice` soit celui des *colonnes* affichées.

6. Contrairement au choix fait en langage C, où l'indice qui varie le plus rapidement est celui de la dernière dimension.

7. En fortran 77, où l'ordre de stockage était garanti, on mettait à profit la disposition des éléments des tableaux pour passer des sections de tableaux multidimensionnels en tant qu'arguments effectifs à des procédures dans lesquelles l'argument muet était déclaré multidimensionnel : il suffisait que la section de tableau s'étende uniquement selon la dernière dimension du tableau.

Pour lire la matrice en affectant chaque ligne du fichier à une ligne (au sens mathématique) de la matrice, on doit expliciter la boucle sur les lignes, alors que celle sur les colonnes peut être implicite ; on fera de même pour afficher le tableau selon la disposition mathématique de la matrice.

```

INTEGER, DIMENSION(2,2)      :: matrice      ! matrice 2 lignes x 2 colonnes
INTEGER                      :: ligne, colonne
...
OPEN(10, file='mat.dat', STATUS='old')      ! ouverture du fichier en lecture
DO ligne = 1, 2
  READ(10, *) matrice(ligne,:)              ! lecture ligne par ligne
END DO
DO ligne = 1, 2
  WRITE(*, *) matrice(ligne,:)              ! affichage ligne par ligne
END DO
CLOSE(10)

```

7.1.3 Constructeurs de tableaux

Lors de sa déclaration, il est possible d'initialiser un tableau de dimension 1 grâce à un *constructeur de tableau* qui est un vecteur encadré par les délimiteurs (/ et /), ou (en fortran 2003) les délimiteurs [et], et dont les éléments peuvent être des scalaires ou des vecteurs, tous du même type, séparés par des virgules. Le vecteur pourra être produit par une ou des boucles implicites.

```

INTEGER, DIMENSION(3)      :: vecteur = (/ 3, 5, 1 /)
INTEGER, DIMENSION(3)      :: vecteur = [ 3, 5, 1 ] ! en fortran 2003
INTEGER                    :: i
INTEGER, DIMENSION(7)      :: v = (/ (2*i, i=1, 4), vecteur /) ! boucle implicite

```

Le tableau `v` contient alors les entiers suivants : 2, 4, 6, 8, 3, 5 et 1.

Pour les tableaux de rang supérieur, on pourra utiliser la fonction `RESHAPE` (cf. 7.4.3, p. 77) qui restructure le tableau mono-dimensionnel `v` en tableau multi-dimensionnel `matrice` suivant le vecteur de profil `profil`.

7.2 Sections de tableaux

Une *section de tableau* est un sous-tableau, de rang inférieur ou égal à celui du tableau.

7.2.1 Sections régulières

Une *section régulière* (regular section) est un sous-tableau dont les indices dans le tableau initial sont en progression arithmétique. Une section régulière peut donc être définie, pour chaque dimension, par un triplet `<debut>:<fin>:<pas>`⁸. Chacun des éléments du triplet est optionnel et les valeurs par défaut sont :

- `<debut>` : la borne inférieure selon la dimension, telle qu'elle a été spécifiée lors de la déclaration (donc 1 si elle n'a pas été précisée dans la déclaration)
- `<fin>` : la borne supérieure selon la dimension, telle qu'elle a été spécifiée lors de la déclaration
- `<pas>` : 1

Comme dans les boucles `DO`, si le pas est différent de 1, il est possible que la limite supérieure ne soit pas atteinte.

```

REAL, DIMENSION(3,6)      :: matrice      ! 3 lignes par 6 colonnes
REAL, DIMENSION(3)       :: colonne

```

8. Ne pas confondre avec `scilab` ou `matlab` dans lesquels l'ordre est `début:pas:fin`.

```

REAL, DIMENSION(3,3)  :: sous_matrice  ! 3 lignes par 3 colonnes
colonne(:) = matrice(:,5)  ! colonne est la 5ème colonne
sous_matrice(:, :) = matrice(:,2:6:2)  ! extraction des colonnes 2, 4 et 6
! de matrice

```

matrice(1:2,2:6:2)

	x		x		x
	x		x		x

matrice(:,2,2:6:2)

	x		x		x
	x		x		x

matrice(2:,3::2)

		x		x	
		x		x	

Remarque : à chaque fois que l'on fixe un indice, on diminue d'une unité le rang du tableau, mais si on restreint l'intervalle d'un des indices de façon à ne sélectionner qu'une valeur, le rang reste inchangé. Par exemple, si on déclare un tableau de rang deux par :

```
REAL, DIMENSION(3,6) :: matrice
```

la section de tableau `matrice(2,:)` est un tableau de rang 1⁹ à 6 éléments; mais la section de tableau `matrice(2:2,:)` reste un tableau de rang 2 de profil (/1, 6/) contenant les mêmes 6 éléments.

♠ 7.2.2 Sections non-régulières

Les *sections non régulières* sont obtenues par indexation indirecte, grâce à un vecteur d'indices.

```

INTEGER, DIMENSION(3)  :: vecteur_indice = (/ 3, 5, 1 /)
INTEGER, DIMENSION(4,6) :: m
INTEGER, DIMENSION(2,3) :: sous_matrice
sous_matrice = m(1:3:2, vecteur_indice)

```

Le sous-tableau `sous_matrice` est alors constitué des éléments suivants de `m` :

m(1,3)	m(1,5)	m(1,1)
m(3,3)	m(3,5)	m(3,1)

7.3 Opérations sur les tableaux

7.3.1 Extension des opérations élémentaires aux tableaux

Les opérateurs agissant sur des types « scalaires » peuvent être appliqués à des tableaux, par extension de leur fonction scalaire à chacun des éléments des tableaux. De telles opérations sont qualifiées d'opérations *élémentaires*. Dans le cas des opérateurs binaires, les tableaux opérands doivent être conformables pour que cette opération puisse s'effectuer terme à terme. Un scalaire est considéré comme conformable avec un tableau quelconque. Dans le cas où les types des éléments de deux tableaux diffèrent, il y a conversion implicite avant application de l'opérateur comme pour des opérations scalaires.

Pour des raisons de lisibilité, on indiquera systématiquement le rang des tableaux manipulés par une notation du type `tab(:, :)` par exemple pour un tableau à deux dimensions, évitant la notation plus concise `tab` qui occulte le caractère multidimensionnel des variables manipulées, ⇐ ♡

```

REAL, DIMENSION(2,3)  :: tab, tab1
REAL, DIMENSION(0:1,-1:1)  :: tab2
tab(:, :) = tab(:, :) + 1  ! le scalaire 1 est ajouté à chaque terme du tableau
tab(:, :) = - tab(:, :)

```

9. Il est d'ailleurs très rare en fortran de déclarer des tableaux dont l'étendue est limitée à 1 sur un des axes, contrairement à ce qui se fait sous `scilab` ou `matlab` qui distinguent vecteurs lignes et vecteurs colonnes tous les deux représentés par des matrices avec un seul élément selon un des deux axes.

```

tab(:, :) = tab(:, :) * tab(:, :)    ! attention: produit terme à terme
                                       ! (et non produit matriciel)
tab(:, :) = cos(tab(:, :))           ! cosinus de chacun des termes du tableau
tab(:, :) = tab1(:, :) + tab2(:, :) ! tab(1,1) = tab1(1,1) + tab2(0,-1) ...
                                       ! seul compte le profil

```

La multiplication matricielle de deux tableaux peut par exemple s'effectuer des deux façons suivantes (en pratique, on utilisera la fonction intrinsèque MATMUL, cf p. 76) :

```

REAL, DIMENSION(2,3) :: a           ! matrice 2 lignes x 3 colonnes
REAL, DIMENSION(3,4) :: b           ! matrice 3 lignes x 4 colonnes
REAL, DIMENSION(2,4) :: c = 0.      ! matrice 2 lignes x 4 colonnes initialisée à 0
INTEGER                :: i, j, k
DO i = 1, 2                          ! boucle sur les lignes de c
  DO k = 1, 4                          ! boucle sur les colonnes de c
    somme: DO j = 1, 3                  ! calcul de la somme
      c(i,k) = c(i,k) + a(i,j) * b(j,k)
    END DO somme
  END DO
END DO
c(:, :) = 0.                          ! réinitialisation de c
DO k = 1, 4                            ! boucle sur les colonnes de c
  somme_v: DO j = 1, 3                  ! calcul vectoriel des sommes
    c(:,k) = c(:,k) + a(:,j) * b(j,k)
  END DO somme_v
END DO

```

♠ Affectation de sections de tableau avec recouvrement

Dans une instruction « vectorielle », il est possible d'affecter une section de tableau à une section de tableau avec recouvrement entre les membre de gauche et de droite de l'opération d'affectation : le résultat peut être différent de celui qu'aurait donné une boucle explicite avec affectation scalaire. En effet,

Le second membre d'une instruction d'affectation dans un tableau est complètement évalué avant l'affectation elle-même.

```

INTEGER                :: i
INTEGER, DIMENSION(5) :: tab = (/ (i, i = 1, 5) /)
tab = tab(5:1:-1)      ! permet d'inverser l'ordre des éléments de tab

```

Une version scalaire aurait nécessité l'emploi d'une variable tampon supplémentaire pour éviter d'écraser des valeurs :

```

INTEGER                :: i
INTEGER, DIMENSION(5) :: tab = (/ (i, i = 1, 5) /)
INTEGER                :: tampon
do i = 1, 5/2          ! i=1, 2 ici : on s'arrête à la moitié du tableau
  tampon = tab(i)
  tab(i) = tab(6-i)
  tab(6-i) = tampon
end do

```

f95/2003 **7.3.2 Instruction WHERE**

Lorsque les opérations à effectuer sur les éléments d'un tableau dépendent d'une condition portant sur ce tableau ou un autre tableau conforme, on peut utiliser l'instruction `WHERE`, ou la structure `WHERE ... END WHERE`, bien adaptée au calcul parallèle¹⁰.

```
[<nom>:] WHERE (<expression_logique_tableau>)
  <bloc d'affectation de tableau(x) conforme(s)>
  ! exécuté pour les éléments où l'expression est vraie
  [ELSEWHERE
  <bloc d'affectation de tableau(x) conforme(s)>]
  ! exécuté pour les éléments où l'expression est fausse
END WHERE [<nom>]
```

```
REAL, DIMENSION(10) :: a, b, c
WHERE ( a > 0. ) b = sqrt(a)
WHERE ( a > 0. )
  c = log(a)           ! pour tous les éléments positifs du tableau a
ELSEWHERE
  c = -1.e20          ! pour tous les éléments négatifs ou nuls du tableau a
END WHERE
```

♠ **7.3.3 Affectation d'une section non-régulière**

Dans le cas où l'opération provoque l'affectation d'une section non-régulière de tableau, il est interdit d'affecter plusieurs fois le même élément, c'est à dire qu'*il ne peut y avoir de recouvrement à l'intérieur du membre de gauche* de l'opération d'affectation¹¹.

```
INTEGER, DIMENSION(3) :: i = (/ 1, 3, 1 /), v = (/ -1, -2, -3 /), w
w = v(i)           ! opération licite qui donne w = (/ -1, -3, -1 /)
w(i) = v       ! opération illicite car double affectation pour w(1)
```

7.4 Fonctions intrinsèques particulières aux tableaux

La majorité des procédures intrinsèques sont élémentaires et peuvent donc être appelées avec des arguments tableaux. Mais il existe des fonctions intrinsèques spécifiquement consacrées à la manipulation des tableaux (cf. A.5, p. 130).

7.4.1 Fonctions d'interrogation

- `SIZE` donne la taille d'un tableau
- `SHAPE` donne le vecteur profil d'un tableau
- `LBOUND` et `UBOUND` fournissent un vecteur constitué des bornes inférieures et supérieures des indices selon chacune des dimensions du tableau. Si une dimension est précisée comme argument optionnel, elles fournissent un scalaire¹² indiquant les bornes des indices selon cette dimension.

10. Depuis le fortran 95, il est possible d'imbriquer des structures `WHERE`.

11. Comme l'ordre d'affectation n'est pas garanti à l'exécution, on conçoit qu'autoriser l'affectation multiple de certains éléments introduirait une ambiguïté dans le résultat. Il est donc logique d'interdire cette possibilité, mais ... parfois difficile de prévoir quand elle pourrait intervenir dans le cas où les indices déterminant les sections sont variables.

12. Attention, même si on appelle `LBOUND` ou `UBOUND` d'un tableau de rang 1, le résultat reste un tableau à un seul élément ; pour obtenir un résultat scalaire, il faut utiliser l'argument optionnel `DIM=1`.

- MINLOC et MAXLOC fournissent le vecteur des indices du *premier* élément respectivement minimum ou maximum du tableau, mais en indexant le tableau à *partir d'une borne inférieure égale à 1* selon chaque dimension, quelles que soient les bornes déclarées.

```

INTEGER, DIMENSION(0:3)      :: v = (/ 3, 2, 2, 3 /)
INTEGER, DIMENSION(-2:2,3)   :: tab
INTEGER                       :: i
tab(:,1) = (/ (i, i=-2, 2) /) ! | -2 -1 0 1 2 |
tab(:,2) = (/ (2*i, i=-2, 2) /) ! | -4 -2 0 2 4 | (transposée de tab)
tab(:,3) = (/ (3*i, i=-2, 2) /) ! | -6 -3 0 3 6 |

```

La matrice associée à `tab` est alors :

-2	-4	-6
-1	-2	-3
0	0	0
1	2	3
2	4	6

SIZE(v)	4
SHAPE(v)	(/ 4 /)
LBOUND(v)	(/ 0 /)
LBOUND(v, DIM=1)	0
MINLOC(v)	(/ 2 /)
MAXLOC(v)	(/ 1 /)
MAXLOC(v, DIM=1)	1

SIZE(tab)	15
SIZE(tab,1)	5
SIZE(tab,2)	3
SHAPE(tab)	(/ 5, 3 /)
LBOUND(tab)	(/ -2, 1 /)
UBOUND(tab)	(/ 2, 3 /)
LBOUND(tab,1)	-2
MINLOC(tab)	(/ 1, 3 /)
MAXLOC(tab)	(/ 5, 3 /)

7.4.2 Fonctions de réduction

Les fonctions de réduction retournent un tableau de rang inférieur (éventuellement un scalaire) à celui passé en argument. Si on spécifie une dimension, seule cette dimension sera affectée par l'opération de réduction et le tableau final sera de rang $r - 1$, où r est le rang du tableau initial.

- SUM et PRODUCT calculent respectivement la somme et le produit des éléments du tableau.
- MAXVAL et MINVAL fournissent respectivement le maximum et le minimum d'un tableau.
- DOT_PRODUCT donne le produit scalaire de deux vecteurs :
 - si u et v sont des vecteurs de réels (ou d'entiers), $\text{DOT_PRODUCT}(u, v) = \text{SUM}(u*v)$.
 - si u et v sont des vecteurs de complexes, $\text{DOT_PRODUCT}(u, v) = \text{SUM}(\text{CONJG}(u)*v)$.
- MATMUL effectue le *produit matriciel* de deux tableaux¹³. Elle suppose que les profils de ses arguments permettent l'opération :
 - produit d'une matrice de n lignes et p colonnes par une matrice de p lignes et q colonnes donnant une matrice de n lignes et q colonnes
 - produit d'une matrice de n lignes et p colonnes par un vecteur de taille p donnant un vecteur de taille n
 - produit d'un vecteur de taille n par une matrice de n lignes et p colonnes donnant un vecteur de taille p .

PRODUCT(tab, DIM=2)	(/ -48, -6, 0, 6, 48)/
SUM(tab, DIM=2)	(/ -12, -6, 0, 6, 12)/
SUM(tab)	0
MAXVAL(tab, DIM=2)	(/ -2, -1, 0, 3, 6)/
MAXVAL(tab, DIM=1)	(/ 2, 4, 6)/
MAXVAL(tab)	6
MINVAL(tab, DIM=2)	(/ -6, -3, 0, 1, 2)/
DOT_PRODUCT(tab(:,1), tab(:,3))	30

13. Rappel : le produit de deux matrices en tant qu'opérateur $*$ agissant sur deux tableaux n'est applicable que sur deux tableaux conformes et effectue le produit *terme à terme* et non le produit matriciel.

Par exemple, les calculs de moments d'une série statistique stockée dans un tableau monodimensionnel peuvent s'exprimer de façon très concise à l'aide de ces fonctions.

```

INTEGER, PARAMETER          :: n = 1000
REAL, DIMENSION(n)         :: x, y
REAL                        :: moyenne_x, moyenne_y, variance_x, covariance
moyenne_x = SUM( x(:) ) / REAL( SIZE( x(:) ) )
moyenne_y = SUM( y(:) ) / REAL( SIZE( y(:) ) )
variance_x = SUM( (x(:) - moyenne_x) ** 2 ) / REAL(SIZE(x(:)) - 1)
covariance = SUM((x(:)-moyenne_x) * (y(:)-moyenne_y)) / REAL(SIZE(x(:)) - 1)

```

7.4.3 Fonctions de transformation

- `RESHAPE(source, shape[, pad] [, order])` restructure les éléments du tableau `source` suivant le profil `shape`. Si le tableau source est de taille inférieure au tableau cible, les éléments manquants sont pris dans le tableau `pad`. L'ordre « traditionnel » de rangement dans le tableau cible peut être modifié via l'argument (optionnel) `order`, qui est un tableau de rang 1.
- `PACK(tab, tabmask [, vecteur])` réarrange les éléments du tableau `tab` sélectionnés selon le masque `tabmask` (tableau de booléen conforme à `tab`) dans un tableau de rang 1. Si l'argument `vecteur` (tableau de rang 1) est fourni, il prescrit l'étendue du tableau résultant et doit comporter au moins autant d'éléments que `tabmask` comporte de valeurs `.true.` : les éléments de `vecteur` sont utilisés pour compléter le résultat si le nombre d'éléments vrais du masque est inférieur à l'étendue de `vecteur`. Les éléments du tableau source `tab` sont traités dans l'ordre habituel, en incrémentant les indices en partant de la gauche (par colonnes pour une matrice).
- `UNPACK(vecteur, tabmask, default)` recopie les éléments du tableau de rang 1 `vecteur` dans un tableau sous le contrôle du masque `tabmask` conformant avec le tableau résultat. Si l'élément du masque est vrai, l'élément correspondant du tableau résultat est pris dans `vecteur`, sinon, on lui donne la valeur `default`.
- `SPREAD(source, dim, ncopies)` crée, par duplication (`ncopies` fois selon la dimension `dim`) du scalaire ou tableau `source`, un tableau de rang immédiatement supérieur à `source`.
- `MERGE(tab_vrai, tab_faux, tab_mask)` fusionne les tableaux `tab_vrai` et `tab_faux` selon le masque `tab_mask` : les trois arguments et le résultat sont des tableaux conformants.
- `CSHIFT(tab, shift[, dim])` et `EOSHIFT(tab, shift[, boundary] [, dim])` effectuent des décalages de `shift` des éléments du tableau selon la dimension éventuellement spécifiée (si le paramètre `dim` n'est pas précisé, cela implique `dim=1`¹⁴). Si `shift` est positif, les décalages se font vers les indices décroissants et réciproquement. `CSHIFT` effectue des décalages circulaires (sans perte d'éléments) alors que `EOSHIFT` remplace les éléments perdus par des « zéros » du type des éléments de `tab` ou par `boundary` si cet argument optionnel est précisé.
- `TRANSPOSE` calcule la transposée d'une matrice.

```

INTEGER          :: i
INTEGER, DIMENSION(8)  :: v = (/ i, i = 1, 8 /), w
INTEGER, DIMENSION(3,4)  :: tab1 = RESHAPE(v, (/ 3, 4 /), pad = (/ 0 /)) &
                          , tab2, tab3

w = CSHIFT(v, 2)
tab2 = CSHIFT(tab1, SHIFT=-1 )
tab3 = CSHIFT(tab1, (/ 1, -2, 2 /), DIM=2)

```

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

RESHAPE(v, (/3,4/), pad=(/0/)) ⇒

1	4	7	0
2	5	8	0
3	6	0	0

14. Les décalages sont appliqués à chacun des vecteurs constituant le tableau et non globalement.

1	4	7	0	PACK(tab1,mod(tab1,2)==1) ⇒	1	3	5	7
2	5	8	0					
3	6	0	0					

1	2	3	4	UNPACK(v(1:4),mod(tab1,2)==1, -1) ⇒	1	-1	4	-1	
						-1	3	-1	-1
						2	-1	-1	-1

1	2	3	SPREAD(/1, 2, 3/),dim=1,ncopies=2) ⇒	1	2	3
					1	2

1	2	3	SPREAD(/1, 2, 3/),dim=2,ncopies=2) ⇒	1	1
				2	2
				3	3

1	4	7	-1	-4	-7	F	T	F	MERGE ⇒	-1	4	-7
2	5	8	-2	-5	-8	T	F	T		2	-5	8
3	6	9	-3	-6	-9	F	T	F		-3	6	-9

1	2	3	4	5	6	7	8	CSHIFT(v,SHIFT=2) ⇒	3	4	5	6	7	8	1	2

1	2	3	4	5	6	7	8	EOSHIFT(v,SHIFT=2) ⇒	3	4	5	6	7	8	0	0

1	4	7	0	CSHIFT(tab1,SHIFT=-1) ⇒	3	6	0	0
2	5	8	0		1	4	7	0
3	6	0	0		2	5	8	0

1	4	7	0	CSHIFT(tab1, (/ 1, -2, 2 /), DIM=2) ⇒	4	7	0	1
2	5	8	0		8	0	2	5
3	6	0	0		0	0	3	6

♠ 7.4.4 Notion de masque

Parfois, un traitement a priori vectoriel via une fonction intrinsèque ne doit pas être appliqué à tous les éléments d'un tableau, mais une expression logique tableau peut servir à décider quels éléments seront concernés par l'instruction tableau. Certaines procédures intrinsèques portant sur des tableaux acceptent un tel tableau booléen conforme à l'argument traité, appelé *masque* (mask), en argument optionnel (de mot-clef MASK=).

```

INTEGER          :: i, produit, somme
INTEGER, DIMENSION(5)  :: v = (/ i, i = -2, 2 /), w = 2 * v
produit = PRODUCT(v, MASK = v /= 0) ! produit des éléments non nuls de v
somme = SUM(w, MASK = v /= 0) ! somme des éléments de w(i) tels que v(i) non nul

```

7.4.5 ALL, ANY, COUNT

Trois fonctions intrinsèques opèrent sur des tableaux de booléens pour vérifier la validité d'une expression logique sur les éléments d'un tableau :

- ALL(<mask>) rend .TRUE. si tous les éléments du tableau booléen <mask> sont vrais, .FALSE. sinon ; elle réalise un ET logique entre les éléments du tableau
- ANY(<mask>) rend .TRUE. si un au moins des éléments du tableau booléen <mask> est vrai, .FALSE. si tous sont à .FALSE. ; elle réalise un OU logique entre les éléments du tableau
- COUNT(<mask>) rend le nombre (entier) des éléments du tableau <mask> qui valent .TRUE..

En pratique, <mask> est produit par application d'un test à un tableau (ou des tableaux conformes).

```

INTEGER                :: i, nb_positifs
INTEGER, DIMENSION(5)  :: v = (/ i, i = -2, 2 /)
REAL, DIMENSION(5)     :: u, u_log
...
nb_positifs = COUNT (v > 0)      ! nombre d'éléments positifs (2 ici)
IF (ALL(u > 0)) u_log = log(u)   ! calculé si tous les éléments de u sont > 0

```

f95/2003 7.4.6 Instruction et structure FORALL

Certaines affectations de tableaux, par exemple combinant deux indices, ne peuvent pas s'exprimer de façon globale sur le tableau, malgré leur facilité à être traitées en parallèle. Par exemple, une affectation entre tableaux de rang différent :

```

DO i = 1, n
  tab(i, i) = v(i) ! copie du vecteur v sur la diagonale de tab
END DO

```

peut s'exprimer à l'aide de l'instruction FORALL :

```

FORALL (i = 1:n) tab(i, i) = v(i)

```

Les expressions de droite sont alors calculées en parallèle pour toutes les valeurs de l'indice, comme si les résultats intermédiaires étaient stockés dans des variables temporaires jusqu'à la fin des calculs en attendant l'affectation finale du membre de gauche. En particulier, au contraire de ce qui se passe dans une boucle classique, l'ordre de parcours des indices est indifférent ; plus précisément il peut être choisi par le processeur pour optimiser (vectoriser) le calcul. Cette construction s'avère donc très adaptée aux calculs de filtrage numérique linéaire. Par exemple, l'instruction suivante calcule bien une différence finie pour estimer une dérivée, alors qu'une boucle dans le sens croissant des indices ne donnerait pas le même résultat.

```

FORALL (i = 1: size(v)-1 ) v(i+1) = v(i+1) - v(i)

```

La structure FORALL ... END FORALL permet d'exprimer une série d'affectations successives portant sur les mêmes indices¹⁵.

```

FORALL (i=1:n, j=1:p)
  a(i, j) = i + j           ! tous les a(i,j) sont calculés
  b(i, j) = a(i, j) ** i   ! avant de calculer b(i,j)
END FORALL

```

La structure FORALL admet de plus un masque (*cf.* 7.4.4, p; 78) avec une condition logique permettant de sélectionner les indices et le corps de la structure peut contenir aussi des constructions WHERE (*cf.* 7.3.2, p. 75).

7.5 Tableaux et allocation dynamique

7.5.1 Tableaux et procédures

Bien entendu, si on travaille avec des tableaux de profil fixe (fixe au moins à l'exécution, mais dont les dimensions peuvent être des expressions constantes, grâce à l'attribut PARAMETER), on peut déclarer leur profil à la fois dans le programme principal et les procédures appelées : on dit que ces tableaux sont à *profil explicite*. Rappelons que, dans ce cas, seul compte le profil du tableau et non ses bornes.

15. Chaque instruction est alors exécutée sur tous les indices avant de passer à la suivante.

```

REAL, DIMENSION(-2:2)      :: tab0
REAL, DIMENSION(5)        :: tab1
CALL SUB(tab0)
CALL SUB(tab1)
...
CONTAINS
  SUBROUTINE SUB(tab)
    REAL, DIMENSION(5)      :: tab
    ...                    ! tab(1) contient tab0(-2) au premier appel
    ...                    ! tab(1) contient tab1(1) au deuxième appel
  END SUBROUTINE SUB
...

```

Mais, il est plus fréquent que des procédures nécessitent l'usage de tableaux de rang connu, mais de profil variable d'un appel à l'autre. Si, dans le programme principal, les tableaux passés en argument doivent toujours être dimensionnés avec des expressions entières constantes, il est recommandé de leur attribuer, en tant qu'arguments muets de la procédure, un *profil implicite*. Pour pouvoir appeler une procédure utilisant comme argument muet des tableaux de profil implicite, la procédure appelante doit connaître l'interface de la procédure appelée; trois possibilités sont envisageables, dont la dernière est préférable :

- utiliser une procédure interne;
- utiliser une procédure externe avec interface explicite;
- utiliser une procédure de module (*cf.* 6.4, p. 58).

De plus, des tableaux locaux à la procédure peuvent être dimensionnés à l'aide d'expressions entières non constantes : on les nomme *tableaux automatiques*.

```

PROGRAM principal
INTEGER, PARAMETER          :: lignes = 3, colonnes = 4
REAL, DIMENSION(lignes,colonnes)  :: tab
REAL, DIMENSION(2*colonnes,2*lignes)  :: tab2
...
CALL sub(tab)
CALL sub2(tab)
CALL sub(tab2)
CALL sub2(tab2)
...
CONTAINS
  SUBROUTINE sub(tab)
    REAL, DIMENSION(:,:)          :: tab ! profil implicite
    ...
    RETURN
  END SUBROUTINE sub

  SUBROUTINE sub2(tab)
    REAL, DIMENSION(:,:)          :: tab ! profil implicite
    REAL, DIMENSION(SIZE(tab,1),SIZE(tab,2))  :: aux ! tableau local automatique
    ...
    RETURN
  END SUBROUTINE sub2
...
END PROGRAM principal

```

♠ 7.5.2 Tableaux dynamiques

Lorsque, par exemple, le profil d'un tableau n'est connu qu'à l'exécution, on peut lui allouer dynamiquement la mémoire nécessaire. Il faut alors déclarer ce tableau en précisant son rang¹⁶ et l'attribut `ALLOCATABLE`, qui en fait un tableau à *profil différé* (deffered-shape-array). Une fois le profil connu, l'instruction `ALLOCATE` permettra d'allouer effectivement la mémoire requise, avant toute opération d'affectation sur ce tableau. Après avoir utilisé le tableau, il est recommandé de libérer la mémoire allouée par l'instruction `DEALLOCATE`, en particulier si on souhaite réutiliser le même identificateur de tableau avec une autre taille.

```

REAL, DIMENSION(:, :), ALLOCATABLE :: tab    ! rang 2, profil différé
INTEGER                               :: lignes, colonnes, erreur
READ (*, *) lignes, colonnes              ! saisie du profil du tableau
ALLOCATE(tab(lignes, colonnes), stat=erreur) ! allocation de la mémoire
IF( erreur /= 0 ) THEN
    WRITE(*, *) 'erreur allocation de tab ', &
               lignes, ' lignes x ', colonnes, ' colonnes'
    STOP
END IF
READ (*, *) tab(:, :)                      ! affectation du tableau
...
DEALLOCATE(tab)                            ! libération de la mémoire

```

Dans un souci de fiabilité des programmes, on prendra soin, avant toute opération d'allocation ou de libération de mémoire, de tester l'état des tableaux grâce à la fonction d'interrogation `ALLOCATED` qui fournit un résultat booléen.

```

REAL, DIMENSION(:), ALLOCATABLE :: tab    ! rang 1, profil différé
INTEGER                               :: n, erreur
...
IF(.NOT.ALLOCATED(tab)) THEN              ! si tab n'est pas déjà alloué
    READ(*, *) n                           ! saisie de la taille
    ALLOCATE(tab(n), stat=erreur)          ! on l'alloue
    IF( erreur /= 0 ) THEN
        WRITE(*, *) 'erreur allocate: mémoire insuffisante'
        STOP
    END IF
ELSE                                       ! si tab est déjà alloué
    n = SIZE(tab)
END IF
...
tab (:) = ...                             ! travail sur tab
...
IF(ALLOCATED(tab)) THEN                  ! si tab est alloué
    DEALLOCATE(tab, stat=erreur)          ! on libère la mémoire
    IF( erreur /= 0 ) THEN
        WRITE(*, *) 'erreur deallocate'
        STOP
    END IF
END IF

```

Un tableau dynamique *local à une procédure* et alloué dans cette procédure est automatiquement libéré à la sortie de la procédure (via `END` ou `RETURN`), sauf s'il est rendu permanent par l'attribut `SAVE`.

16. Mais sans indiquer sa taille.

f95/2003 **Tableau dynamique en paramètre** Un tableau dynamique peut aussi être passé en argument d'une procédure, mais, en fortran 95, il doit être déjà alloué avant l'appel de la procédure et n'est pas considéré comme allouable dans la procédure. Ces restrictions sont levées en fortran 2003, ainsi qu'avec les extensions spécifiques de certains compilateurs fortran 95¹⁷. Le tableau doit alors posséder l'attribut `ALLOCATABLE` dans la procédure, qui peut se charger de l'allocation. S'il s'agit d'un argument de sortie (`INTENT(OUT)`), il est désalloué si nécessaire lors de l'appel de la procédure.

```

1  MODULE m_alloc_sub
2  CONTAINS
3    SUBROUTINE init_tab(tab, m, n)
4      IMPLICIT NONE
5      INTEGER, INTENT(in) :: m, n
6      INTEGER, ALLOCATABLE, INTENT(out) :: tab(:, :)
7      INTEGER :: ok
8      INTEGER :: i, j
9      IF(.NOT.ALLOCATED(tab)) THEN
10     ALLOCATE(tab(m, n), stat = ok)
11     IF (ok /= 0) THEN
12       WRITE(*, *) 'erreur allocation'
13       STOP
14     END IF
15     DO i = 1, m
16       DO j = 1, n
17         tab(i, j) = 1000 * i + j
18       END DO
19     END DO
20   ELSE
21     ! ne devrait pas se produire en fortran 2003 car tab est INTENT(OUT)
22     WRITE(*, *) 'tableau déjà alloué'
23     STOP
24   END IF
25 END SUBROUTINE init_tab
26 END MODULE m_alloc_sub
27 PROGRAM alloc_proc
28   USE m_alloc_sub
29   IMPLICIT NONE
30   INTEGER, ALLOCATABLE :: t(:, :)
31   INTEGER :: m, n
32   INTEGER :: i
33   WRITE (*, *) 'entrer les dimensions du tableau: m et n'
34   READ (*, *) m, n
35   ! appel du sous programme qui alloue et initialise le tableau
36   CALL init_tab(t, m, n)
37   DO i = 1, m
38     WRITE (*, *) t(i, :)
39   END DO
40   m = 2 * m
41   n = 2 * n
42   ! deuxième appel du sous programme avec tab déjà alloué, mais
43   ! nécessitant une réallocation car les diemsnions ont changé
44   CALL init_tab(t, m, n)
45   DO i = 1, m
46     WRITE (*, *) t(i, :)
47   END DO
48   IF(ALLOCATED(t)) THEN
49     DEALLOCATE(t)
50   END IF
51 END PROGRAM alloc_proc

```

17. En particulier xlf d'IBM, le compilateur NAG et g95 avec l'option `-std=f2003` ou `-std=f95` assortie de `-ftr15581` (cf. E.5.1, p. 153) autorisent l'allocation de l'argument tableau dans la procédure.

f2003 7.5.3 Allocation au vol de tableaux dynamiques par affectation

En fortran 2003, un tableau allouable peut être (ré-)alloué automatiquement si on lui affecte une expression tableau :

- s'il n'était pas alloué auparavant, il est alloué avec le profil de l'expression qui lui est affectée ;
- s'il était déjà alloué avec un profil différent, il est réalloué pour ajuster son profil à celui du membre de droite de l'affectation.

La réallocation automatique peut être inhibée si on précise dans le membre de gauche le symbole «:» pour chaque dimension. Dans ce cas, les éléments de l'expression de droite en dehors du profil du tableau de gauche sont ignorés.

Par exemple, pour des tableaux de rang 1, le programme suivant :

```

1  ! allocation automatique par affectation
2  ! fortran 2003 seulement
3  ! ok avec NAG Fortran Compiler Release 5.2(643) le 12 fev 2009
4  PROGRAM t_alloc_affect
5  INTEGER, DIMENSION(:), ALLOCATABLE :: v1, v2, v3
6  v1 = [1, 2] ! allocation de v1 par affectation
7  WRITE(*,*) "taille de v1=", SIZE(v1), " v1=", v1
8  v2 = [-3, -2, -1 ] ! allocation de v2 par affectation
9  WRITE(*,*) "taille de v2=", SIZE(v2), " v2=", v2
10 v3 = v1 ! allocation implicite de v3 => 2 éléments
11 WRITE(*,*) "taille de v3=", SIZE(v3), " v3=", v3
12 v1 = v2 ! réallocation implicite de v1 => 3 éléments
13 WRITE(*,*) "taille de v1=", SIZE(v1), " v1=", v1
14 v3(:) = v2 ! pas de réallocation de v3 => v2(3) inutilisé
15 WRITE(*,*) "taille de v3=", SIZE(v3), " v3=", v3
16 DEALLOCATE(v1, v2, v3)
17 END PROGRAM t_alloc_affect

```

affiche :

```

taille de v1= 2  v1= 1 2
taille de v2= 3  v2= -3 -2 -1
taille de v3= 2  v3= 1 2
taille de v1= 3  v1= -3 -2 -1
taille de v3= 2  v3= -3 -2

```

Chapitre 8

Chaînes de caractères

Ce court chapitre regroupe les outils, dont certains déjà introduits dans les chapitres précédents, permettant la manipulation des chaînes de caractères en fortran : le type chaîne (*cf.* chapitre 2), les expressions de type chaîne (*cf.* chapitre 3), les sous-chaînes, les fonctions manipulant des chaînes, les entrées/sorties de chaînes (*cf.* chapitre 5), les tableaux de chaînes et le passage d'arguments chaînes dans les procédures.

8.1 Le type chaîne de caractères

Le langage fortran comporte le type intrinsèque CHARACTER pour stocker les chaînes de caractères. À chaque objet de type CHARACTER est associée une longueur LEN qui est le nombre des caractères de la chaîne.

8.1.1 Les constantes chaînes de caractères

Les constantes chaînes de caractères sont délimitées par des apostrophes simples «'» ou¹ des guillemets «"». À l'intérieur d'un des types de délimiteur, l'autre est considéré comme un caractère quelconque, ce qui permet d'écrire par exemple "aujourd'hui". Toutefois, on peut aussi introduire le délimiteur à l'intérieur de la chaîne à condition de le dupliquer, comme dans 'aujourd'hui'.

Pour écrire une constante chaîne sur plusieurs lignes, il faut terminer toutes les lignes sauf la dernière par le symbole & de continuation de ligne. Par défaut tous les espaces en début de ligne de suite sont alors significatifs (*cf.* 1.4.2 p. 7); mais on peut adopter une présentation plus agréable en spécifiant par un symbole & supplémentaire le début de la suite de la chaîne, ignorant ainsi tous les espaces situés avant.

```
WRITE (*,*) 'chaîne de caractères comportant plusieurs lignes dans &  
&le programme source sans espace superflu'
```

8.1.2 Les déclarations de chaînes de caractères

CHARACTER(LEN=<longueur>) :: <chaîne>

permet de déclarer une chaîne comportant *longueur* caractères. La longueur de la chaîne est bien le nombre de caractères², espaces inclus, qui la constituent. Un caractère est simplement une chaîne de longueur 1. La longueur d'un scalaire chaîne de caractères *constant nommé* (c'est-à-dire avec l'attribut PARAMETER) peut être calculée par le compilateur si on spécifie³ LEN=* dans la déclaration.

1. En langage C, ces délimiteurs sont spécialisés : le délimiteur des constantes chaînes de caractères est le guillemet «"» et l'apostrophe «'» sert à délimiter les constantes de type caractère individuel.

2. En langage C au contraire, le tableau de caractères doit comporter un élément supplémentaire pour indiquer la fin de la chaîne.

3. On retrouve la même propriété qu'en langage C où la taille d'un tableau de caractères constant peut être calculée par le compilateur lors de l'initialisation.

```
CHARACTER(LEN=*), PARAMETER :: chaine_fixe="mot"
```

f2003 8.1.3 Les variantes du type chaînes de caractères

Fortran 90 doit donner accès à un type de caractères par défaut, et peut aussi fournir d'autres jeux de caractères dépendant du compilateur de la machine.

Mais fortran 2003 a introduit la possibilité d'accéder à des variantes du type chaîne associées à différents codage des caractères, notamment pour les caractères UNICODE. La fonction `SELECTED_CHAR_KIND` rend un entier indiquant le numéro du sous-type chaîne demandé en paramètre : parmi les sous-types possibles on trouve `DEFAULT`, `ASCII` et `ISO_10646` pour les caractères Unicode (encore noté `UCS_4` assimilable à `UTF-32`)⁴, mais seul `DEFAULT` est requis par la norme. Si la variante demandée n'est pas disponible, la fonction `SELECTED_CHAR_KIND` rend -1. La variante `ISO_10646` sera nécessaire pour lire des fichiers texte codés en `UTF-8` (cf. 5.3.2, p. 38).

Les constantes chaînes de variante de type autre que le défaut sont désignées en les *préfixant* par le numéro du sous-type donné par `SELECTED_CHAR_KIND` suivi du symbole souligné⁵. Par exemple, si le codage par défaut est `ISO-8859-1` :

```
INTEGER, PARAMETER :: defchar = SELECTED_CHAR_KIND('DEFAULT')
CHARACTER(LEN=3, KIND=defchar) :: mot=defchar_"été"
```

Pour un désigner caractère unicode en `UTF-32`, on peut utiliser la fonction `CHAR` en lui fournissant les codes-points unicode comme des entiers en hexadécimal (cf. 2.4, p. 14), par exemple :

```
INTEGER, PARAMETER :: utf32=SELECTED_CHAR_KIND('ISO_10646') ! UCS_4 = UTF-32
CHARACTER(LEN=1, KIND=utf32) :: car1_utf32, car2_utf32
car1_utf32 = CHAR(INT(Z'00E6'), KIND=utf32) ! ligature ae æ
car2_utf32 = CHAR(INT(Z'FB03'), KIND=utf32) ! ligature ffi (3 octets en UTF-8)
```

8.2 Expressions de type chaîne de caractères

Une expression de type chaîne de caractères peut être construite par concaténation de variables ou de constantes chaînes, par extraction de sous-chaîne ou enfin à l'aide de fonctions à résultat chaîne de caractères.

8.2.1 Concaténation de chaînes

L'opérateur `//` permet de concaténer⁶ deux chaînes de caractères. Par exemple, l'expression `"bonjour" // " au revoir"` est évaluée comme `"bonjour au revoir"`.

8.2.2 Sous-chaînes

De façon assez semblable aux sous-sections des tableaux, le langage fortran permet de manipuler des sous-chaînes de caractères, selon la syntaxe suivante⁷ :

-
4. Par exemple, le compilateur NAG possède quatre variantes du type chaîne, de paramètres `KIND` :
 - 1 pour l'`ASCII` sur 1 octet qui comporte en fait le codage `ISO-8859-1` ;
 - 2 sur deux octets pour le codage `UCS_2`, qui coïncide avec `UTF-16` ;
 - 3 pour le codage `JIS X 0213` permettant d'accéder aux caractères japonais ;
 - 4 sur quatre octets pour `UCS_4`, assimilable à `UTF-32`.

5. Ainsi, pour les chaînes, on préfixe la variante de type, alors que pour les autres types (entiers, réels), on suffixe la variante de type.

6. En langage C, la concaténation de chaînes de caractères est obtenue par simple juxtaposition, sans opérateur explicite.

7. On notera que le symbole « : » n'est pas facultatif, même si on veut désigner une sous-chaîne réduite à un caractère.

```
chaine1([<deb>]:[<fin>])
```

désigne la sous-chaîne extraite de `chaine1` commençant au caractère numéro `deb` et terminant au caractère numéro `fin` sachant que par défaut `deb` vaut 1 et `fin` est la longueur de `chaine1`, soit `LEN(chaine1)`. Si `deb` est supérieur à `fin`, la sous-chaîne est vide. Avec les déclarations suivantes :

```
CHARACTER(LEN=*) , PARAMETER :: mot1 = "bonjour", mot2 = "└─bonjour└─└─└─"
```

`mot1(1:3)` vaut "bon" ainsi que `mot1(:3)` et `mot1(4:7)` vaut "jour" de même que `mot1(4:)` mais `mot2(:4)` vaut "└─bon" et `mot2(5:)` vaut "jour└─└─└─".

8.2.3 Affectation

L'affectation globale d'expressions chaînes de caractères est possible en fortran⁸ sachant que le membre de gauche peut être une chaîne ou une sous-chaîne et le membre de droite une expression de type chaîne. Au cas où la longueur des chaînes de part et d'autre du signe = diffère,

- si l'expression chaîne du membre de droite est plus longue que la variable à laquelle on doit l'affecter, la chaîne de droite est tronquée côté droit ;
- si l'expression chaîne du membre de droite est plus courte que la variable à laquelle on doit l'affecter, la chaîne de droite est complétée par des espaces pour aboutir à la longueur de la chaîne de gauche.

Avec les déclarations et les affectations suivantes :

```
CHARACTER(LEN=7) :: mot1 = "bonjour"
CHARACTER(LEN=9) :: mot2 = "au revoir"
CHARACTER(LEN=17) :: mot3
CHARACTER(LEN=7) :: mot4
CHARACTER(LEN=5) :: mot5
mot3 = mot1 // " " // mot2
mot4 = mot2(1:3) // mot1(4:)
mot5 = mot1(4:7)
mot2(1:2) = "A "
```

on obtient `mot3="bonjour au revoir"`, `mot4="au jour"`, `mot5="jour "` et `mot2="A revoir"`.

8.3 Les fonctions opérant sur les chaînes de caractères

8.3.1 Suppression des espaces terminaux avec TRIM

La fonction `TRIM(String=ch)` d'argument et de résultat de type chaîne de caractère supprime les blancs à droite de la chaîne passée en argument. Elle rend donc une chaîne de longueur `LEN_TRIM(ch)` inférieure au égale à celle `LEN(ch)` de `ch`.

`TRIM(" ab cd ")` donne " ab cd"

8.3.2 Justification à gauche avec ADJUSTL

La fonction élémentaire `ADJUSTL(String=ch)` supprime tous les blancs en début de la chaîne `ch` et en ajoute autant à droite de façon à rendre une chaîne *de même longueur* que son argument, mais justifiée à gauche.

`ADJUSTL(" ab cd ")` donne "ab cd "

`TRIM(ADJUSTL(chaine))` enlève donc les espaces à la fois à gauche et à droite.

8. Les chaînes de caractères en C étant des tableaux de caractères, il n'est pas possible de les affecter globalement et on doit recourir à des fonctions de copie de chaînes à cette fin.

8.3.3 Justification à droite avec ADJUSTR

De façon symétrique, la fonction élémentaire `ADJUSTR(String=ch)` supprime tous les blancs en fin de la chaîne `ch` et en ajoute autant à gauche de façon à rendre une chaîne *de même longueur* que son argument, mais justifiée à droite.

`ADJUSTR(" ab cd ")` donne " ab cd"

8.3.4 Les fonctions LEN et LEN_TRIM

La fonction `LEN` d'argument chaîne et de résultat entier rend le nombre de caractères de la chaîne telle qu'elle a été déclarée à la compilation, indépendamment du contenu de la dite chaîne. La variante `LEN_TRIM` ne tient pas compte des espaces terminaux, mais elle compte ceux en début de chaîne : `LEN_TRIM(chaine) = LEN(TRIM(chaine))`. Ainsi, avec les constantes chaînes suivantes,

```
CHARACTER(LEN=*), PARAMETER :: mot1 = "bonjour", mot2 = " bonjour  "
WRITE(*,*) "longueurs totales de mot1 et mot2 ", LEN(mot1), LEN(mot2)
WRITE(*,*) "longueurs sans espaces terminaux ", LEN_TRIM(mot1), LEN_TRIM(mot2)
```

on affichera des longueurs respectives de 7 et 11, mais des longueurs après suppression des espaces finaux de 7 et 8.

Contrairement à `LEN`, `LEN_TRIM` est une fonction élémentaire (*cf.* 8.5, p. 89).

8.3.5 Recherche de sous-chaîne avec INDEX

La fonction élémentaire `INDEX(String=ch1, SUBSTRING=ch2)` rend un entier donnant la position du début de la sous-chaîne `ch2` dans la chaîne `ch1`. Seule la première occurrence est recherchée et la fonction `INDEX` rend 0 si la sous-chaîne cherchée n'est pas présente dans la chaîne `ch1`.

Par défaut, la recherche se fait de gauche à droite dans `ch1`. Mais cette fonction admet un paramètre optionnel `BACK` de type booléen qui, lorsqu'il est fourni avec la valeur `.true.`, précise que la recherche se fait en commençant par la droite : dans ce cas, si la sous-chaîne `ch2` n'est pas présente dans `ch1`, `INDEX` rend `LEN(ch1) + 1`.

`INDEX("attente", "te")` donne 3

`INDEX("attente", "te", BACK=.true.)` donne 6

8.3.6 Recherche des caractères d'un ensemble avec SCAN

La fonction élémentaire `SCAN(String=ch1, SET=ch2)` recherche dans la chaîne `ch1` l'un des caractères de l'ensemble `ch2` (peu importe l'ordre) et rend un entier donnant la position du premier trouvé. Si aucun des caractères de `ch2` n'est présent dans `ch1`, elle rend 0.

Par défaut, la recherche se fait de gauche à droite dans `ch1`. Mais cette fonction admet un paramètre optionnel `BACK` de type booléen qui, lorsqu'il est fourni avec la valeur `.true.`, précise que la recherche se fait en commençant par la droite : dans ce cas, l'entier rendu est donc la position dans `ch1` du dernier caractère de l'ensemble `ch2`.

`SCAN("aujourd'hui", "ru")` donne 2 pour le premier u

`SCAN("aujourd'hui", "ru", BACK=.true.)` donne 10 pour le dernier u

8.3.7 Recherche des caractères hors d'un ensemble avec VERIFY

La fonction élémentaire `VERIFY(String=ch1, SET=ch2)` recherche dans la chaîne `ch1` le premier caractère *hors de* l'ensemble `ch2` (peu importe l'ordre) et rend un entier donnant la position du premier trouvé. Si tous les caractères de `ch1` appartiennent à `ch2`, elle rend 0.

Par défaut, la recherche se fait de gauche à droite dans `ch1`. La fonction `VERIFY` admet, comme `SCAN`, un argument optionnel booléen `BACK`, qui, s'il est positionné à `.true.`, précise que la recherche se fait de droite à gauche.

`VERIFY("aujourd'hui", "aeiou")` donne 3 pour le j

`VERIFY("aujourd'hui", "aeiou", BACK=.true.)` donne 9 pour le h

Les fonctions `VERIFY` et `SCAN` jouent des rôles identiques, à ceci près que leurs arguments `SET=ch2` devraient être des ensembles complémentaires (relativement à l'ensemble des caractères présents dans `ch1`) pour qu'elles donnent le même résultat.

`SCAN("aujourd'hui", "dhjr'")` donne 3 pour le j

`VERIFY("aujourd'hui", "aiou")` donne 3 pour le j

8.3.8 Duplication de chaînes avec `REPEAT`

La fonction `REPEAT(STRING=ch, NCOPIES=n)` duplique n fois la chaîne `ch` et concatène les copies, rendant une chaîne de longueur n fois celle de `ch`.

`REPEAT(" ab ", 3)` donne " ab ab ab "

8.3.9 Conversion de caractère en entier avec `ICHAR` et `IACHAR`

Les fonctions élémentaires `ICHAR(c)` et `IACHAR(c)` renvoient l'entier donnant le numéro du caractère `c` respectivement dans la variante de type de `c` ou dans le code ASCII. Le résultat dépend de la machine et du compilateur sauf avec `IACHAR` pour les 128 premiers caractères du code ASCII. Par exemple,

`IF ((IACHAR(c) >= IACHAR('0')) .AND. (IACHAR(c) <= IACHAR('9')))` permet de tester si `c` est un caractère numérique.

8.3.10 Conversion d'entier en caractère avec `CHAR` et `ACHAR`

Les fonctions élémentaires `CHAR(i [, KIND=var_char])` et `ACHAR(i)` renvoient le caractère correspondant à l'entier i respectivement dans la variante de type indiquée par `var_char` (la variante `DEFAULT` si le paramètre de variante de type n'est pas précisé) ou dans le code ASCII. Le résultat dépend de la machine et du compilateur sauf avec `ACHAR` en ASCII pour $1 \leq i \leq 127$.

8.3.11 Comparaison de chaînes de caractères avec `LLT`, `LLE`, `LGE` et `LGT`

Les fonctions élémentaires de comparaison selon l'ordre lexicographique `LLT`, `LLE`, `LGE` et `LGT` admettent deux arguments chaînes de caractères et donnent un résultat booléen. La comparaison s'appuie sur l'ordre lexicographique de l'ASCII alors que les opérateurs logiques associés effectuent la comparaison selon l'ordre lexicographique du jeu de caractères par défaut, qui dépend de la machine. Ces fonctions sont donc plus portables que les opérateurs associés. Si les deux chaînes à comparer n'ont pas la même longueur, la plus courte est complétée par des espaces à droites avant comparaison.

<code>LLT</code>	lexically lower than	<
<code>LLE</code>	lexically lower or equal	<=
<code>LGE</code>	lexically greater or equal	>=
<code>LGT</code>	lexically greater than	>

`IF (LGE(c, '0') .AND. LLE(c, '9'))` permet aussi de tester si le caractère `c` est numérique.

8.3.12 Le caractère de fin de ligne `NEW_LINE`

f2003 ⇒

Pour permettre l'écriture de fichiers en mode `stream` (cf. 5.2.4, p.35), fortran 2003 a introduit la fonction `NEW_LINE(CHAR=ch)` qui rend le caractère de fin de ligne (new-line associé à `\n` du C) dans la variante de type caractère de son argument.

8.4 Les entrées-sorties de chaînes de caractères

8.4.1 Les entrées-sorties de chaînes formatées

- Le descripteur de conversion des chaînes de caractères est soit (cf. 5.4.1, p.42) :
- `A` : dans ce cas, le nombre de caractères est celui de la variable associée.

- An où n est un entier spécifiant le nombre de caractères. Si n est plus grand que la longueur de la chaîne à écrire, on complète par des blancs à gauche. Si à l'inverse, n est plus petit que la longueur de la chaîne à écrire, celle-ci est tronquée⁹ aux n premiers caractères.

Par exemple, si on écrit "salut" avec différentes largeurs, on obtient :

format	résultat
A	[salut]
A5	[salut]
A8	[salut]
A2	[sa]

8.4.2 Les entrées de chaînes en format libre

En format libre, la lecture des chaînes de caractères peut se faire :

- avec des délimiteurs « " » ou « ' », ce qui permet de lire des chaînes comportant des séparateurs comme l'espace, la virgule ou le « / ».
- sans délimiteur et alors la lecture s'arrête au premier séparateur (espace, virgule ou « / »).

8.4.3 Les fichiers internes : codage en chaîne de caractères et décodage

L'écriture de données numériques sur des chaînes de caractères ou la lecture de données dans des chaînes de caractères¹⁰ mettent en œuvre les méthodes de conversion évoquées plus haut, où les chaînes dans lesquelles on écrit ou on lit sont qualifiées de fichiers internes (cf. 5.2.1, 33).

Par exemple, le programme suivant convertit i et j en chacun 3 caractères qu'il écrit dans la variable `f_int`; puis il lit dans cette même variable (ou fichier interne) l'entier k sur 6 chiffres obtenus par concaténation de ceux de i et j .

```

1 PROGRAM fich_int
2 IMPLICIT NONE
3 CHARACTER(len=6) :: f_int
4 INTEGER :: i=100 , j=200 , k
5 WRITE(*,*) 'i=', i, 'j=', j
6 WRITE(f_int(1:6), '(2i3.0)') i, j ! écriture dans le fichier interne f_int
7 ! les chiffres de i et j sont juxtaposés dans la chaîne f_int
8 READ(f_int(1:6), '(i6.0)') k ! lecture de k sur le fichier interne f_int
9 WRITE(*,*) 'i=', i, 'j=', j, 'k=', k
10 END PROGRAM fich_int

```

Il affiche :

```

i= 100 j= 200
i= 100 j= 200 k= 100200

```

On utilise notamment la conversion d'entiers en chaînes de caractères pour composer des formats variables pour effectuer des opérations d'entrée-sortie (cf. 5.4.5, p. 45).

8.5 Les tableaux de chaînes de caractères

L'assemblage de plusieurs chaînes de caractères sous forme de tableau de chaînes n'est possible que si toutes les chaînes ont la même longueur, conformément à la structure « rectangulaire » des tableaux.

9. Ce comportement diffère du cas de l'écriture de données numériques, où la conversion ne se fait pas si la largeur du champ est insuffisante : on écrit alors n symboles « * ».

10. En langage C, ces opérations se font grâce aux fonctions `sprintf` et `sscanf`.

CHARACTER(LEN=<longueur>), DIMENSION(<dim>) :: <tab_ch>

Si la syntaxe d'accès aux éléments du tableau respecte la syntaxe générale des tableaux, il est aussi possible de référencer directement des sous-chaînes d'un élément du tableau. On remarquera que la fonction LEN appliquée à un tableau de chaînes renvoie *un scalaire* entier, qui est la longueur *commune* de chacun de ses éléments, alors que la fonction *élémentaire* LEN_TRIM renvoie un tableau d'entiers conforme au tableau de chaînes.

```

1 PROGRAM tab_chaines
2 IMPLICIT NONE
3 CHARACTER(LEN=4), DIMENSION(3) :: tab_ch
4 CHARACTER(LEN=4) :: ch
5 INTEGER :: i
6 tab_ch(1:2) = (/ "abcd", "ABCD" /) ! avec constructeur
7 tab_ch(3) = "1234"
8 WRITE(*,*) "lgueur commune des chaines ", LEN(tab_ch)
9 WRITE(*,*) "affichage en colonnes "
10 DO i=1, SIZE(tab_ch)
11   WRITE(*,*) tab_ch(i)
12 END DO
13 WRITE(*,*) "... en ligne: ", tab_ch
14 ! extraction de sous-chaînes pour construire une chaîne
15 ch = tab_ch(1)(1:2) // tab_ch(2)(3:4)
16 WRITE(*,*) "déduite par extraction ", ch
17 ! autre formulation avec des sous-chaînes
18 ch(1:2) = tab_ch(1)(1:2)
19 ch(3:4) = tab_ch(2)(3:4)
20 WRITE(*,*) "déduite par extraction ", ch
21 ! remplacer les fins de chaîne par 1, 2 ou 3 blancs
22 DO i=1, SIZE(tab_ch)
23   tab_ch(i)(5-i:) = repeat(" ", i)
24 END DO
25 WRITE(*,*) "des blancs en fin "
26 WRITE(*,*) "affichage en colonnes entre [] "
27 DO i=1, SIZE(tab_ch)
28   WRITE(*,*) "[" , tab_ch(i), "]"
29 END DO
30 WRITE(*,*) "LEN(tab_ch) ", LEN(tab_ch)
31 WRITE(*,*) "LEN_TRIM(tab_ch) ", LEN_TRIM(tab_ch)
32 END PROGRAM tab_chaines

```

affichage associé

```

lgueur commune des chaines  4
affichage en colonnes
abcd
ABCD
1234
... en ligne: abcdABCD1234
déduite par extraction abCD
déduite par extraction abCD
des blancs en fin
affichage en colonnes entre []
[abc ]
[AB  ]
[1   ]
LEN(tab_ch)  4
LEN_TRIM(tab_ch)  3 2 1

```

8.6 Chaînes et procédures

Les chaînes de caractères peuvent être passées en argument de procédures sans que leur longueur soit indiquée explicitement dans l'argument formel de type chaîne, comme on peut le faire pour le profil des tableaux (si le rang est connu). Au moment de l'appel, la longueur de la chaîne passée en argument effectif est déterminée dans l'appelant et transmise de façon implicite à l'appelé.

Il est aussi possible de déclarer des chaînes de caractères *automatiques* dans les procédures, chaînes dont la longueur dépend d'un argument passé à la procédure.

L'exemple suivant montre une fonction `create_ch` qui rend une chaîne dont la longueur est passée en argument, ainsi qu'un sous-programme `imprime_tab` qui admet en argument un tableau 1D de taille quelconque constitué de chaînes de caractères de longueur quelconque (mais identique).

```

1  MODULE m_proc_ch
2  IMPLICIT NONE
3  CONTAINS
4  FUNCTION create_ch(deb, n)
5     CHARACTER(LEN=n) :: create_ch
6     INTEGER, INTENT(IN) :: deb, n
7     CHARACTER(LEN=n) :: ch
8     INTEGER :: k
9     DO k=1, n ! pris dans l'ordre ascii
10      ch(k:k) = ACHAR(k + deb -1 )
11    END DO
12    create_ch = ch
13  END FUNCTION create_ch
14  SUBROUTINE imprime(ch)
15     CHARACTER(LEN=*), INTENT(IN) :: ch
16     WRITE(*,*) "ch de lg ", LEN(ch), "[", ch, "]"
17     RETURN
18  END SUBROUTINE imprime
19  SUBROUTINE imprime_tab(tab_ch)
20     CHARACTER(LEN=*), DIMENSION(:), INTENT(IN) :: tab_ch
21     CHARACTER(LEN=LEN(tab_ch)+2) :: ch
22     INTEGER :: i
23     WRITE(*,*) "tableau de ", SIZE(tab_ch), " chaines"
24     WRITE(*,*) " de longueur ", LEN(tab_ch)
25     DO i=1, SIZE(tab_ch)
26      ch = "["// tab_ch(i) // "]"
27      WRITE(*,*) ch
28    END DO
29    RETURN
30  END SUBROUTINE imprime_tab
31  END MODULE m_proc_ch
32
33  PROGRAM proc_chaines
34  USE m_proc_ch
35  CHARACTER(LEN=4) :: ch4
36  CHARACTER(LEN=6) :: ch6
37  CHARACTER(LEN=4), DIMENSION(3) :: tab_ch
38  ch4 = create_ch(33, 4)
39  ch6 = create_ch(33, 6)
40  CALL imprime(ch4)
41  CALL imprime(ch6)
42  tab_ch = (/ "abcd", "ABCD", "1234" /)
43  CALL imprime_tab(tab_ch)
44  END PROGRAM proc_chaines

```

affichage associé

```

ch de lg  4 [!#$]
ch de lg  6 [!#$%&]
tableau de 3 chaines
de longueur  4
[abcd]
[ABCD]
[1234]

```

Chapitre 9

Types dérivés ou structures

Pour représenter des objets plus sophistiqués que ceux représentables avec les types prédéfinis, même avec les tableaux (dont les éléments sont tous de même type) on peut définir des *types dérivés* (derived types) ou *structures*, constitués de plusieurs éléments ou *champs* encore appelés *composantes* dont les types peuvent être différents. Si un tableau ne permet de regrouper que des éléments homogènes désignés par leur indice, un type dérivé permet de regrouper des éléments hétérogènes désignés par leur identificateur. Le nombre de champs d'un type dérivé est généralement plus faible que le nombre d'éléments d'un tableau, mais il est possible d'imbriquer des types dérivés voire de les étendre (cf. 9.3, page 94). alors qu'on ne peut pas construire des tableaux de tableaux. Mais il est possible de construire des tableaux de types dérivés et d'insérer des tableaux dans des types dérivés (cf. 9.4, page 94).

9.1 Définition d'un type dérivé

On peut, par exemple, définir un type dérivé `point` dont les trois composantes sont : une lettre d'identification `lettre`, suivie des coordonnées (réelles) `abscisse` et `ordonnee` du point.

```
TYPE point
  CHARACTER(LEN=1) :: lettre
  REAL :: abscisse
  REAL :: ordonnee
END TYPE point
```

Plus généralement, la syntaxe de définition d'un type dérivé, bien souvent encapsulée dans un module (avant l'instruction `CONTAINS`), est :

```
TYPE [, <accès> ::] <nom_du_type>
  [SEQUENCE]
  <définition d'un champ>
  [<définition d'un champ>] ...
END TYPE [<nom_du_type>]
```

où *accès* permet de préciser sa visibilité `PRIVATE` ou `PUBLIC`. Noter qu'un type dérivé public peut comporter des composantes privées.

9.1.1 L'attribut SEQUENCE

La définition d'un type dérivé n'implique pas en général un ordre ni une taille¹ pour le stockage en mémoire des composantes, laissés libres pour le compilateur. Dans certains cas, notamment si

1. Ne pas croire que la taille d'une variable de type dérivé soit toujours la somme des tailles de ses composantes. En général, pour des raisons d'efficacité, le compilateur procède à un alignement des composantes sur des mots par

on souhaite transmettre des données de type dérivé à une procédure écrite dans un autre langage (hormis le C, pour lequel d'autres techniques d'interopérabilité existent, *cf.* chap. 12), on peut imposer, par l'instruction `SEQUENCE`, le stockage des composantes par concaténation en respectant l'ordre de définition. S'il s'agit de structures imbriquées, il est évidemment nécessaire que toutes les sous-structures aient été définies avec l'attribut `SEQUENCE`.

9.2 Référence et affectation des structures

9.2.1 Constructeur de type dérivé

Une fois le type dérivé `point` défini, on peut déclarer les variables `a` et `b` du type `point`, et leur affecter une valeur en utilisant le *constructeur de structure*, nommé comme le type dérivé. Appeler le constructeur est par exemple nécessaire pour initialiser une structure lors de sa déclaration (comme pour `b` ci-après) :

```
TYPE(point) :: a, b = point('B', 1., -1.)
a = point('A', 0., 1.)
```

En fortran 95, les composantes doivent être passées au constructeur en respectant leur position dans le type dérivé, mais depuis fortran 2003, elles peuvent aussi être désignées par mot-clef. ⇐ f2003

```
TYPE(point) :: a
a = point(abscisse = 0., lettre = 'A', ordonnee = 1.) ! fortran 2003
```

9.2.2 Sélecteur de champ

Références et affectations peuvent concerner la structure globale :

```
a = b          ! suppose a et b de même type dérivé
WRITE(*, *) b ! imprime (dans l'ordre) les champs qui constituent la structure b
```

affecte au point `a` les mêmes composantes que celles du point `b`.

Mais on peut aussi accéder individuellement à chacun des champs qui composent la structure grâce au sélecteur de champ `%` pour les affecter :

```
a%lettre = 'A'
a%abscisse = 0.
a%ordonnee = 1.
```

ou pour les référencer, y compris dans des expressions :

```
CHARACTER(LEN=1) :: c
c = a%lettre
WRITE(*, *) a%abscisse, a%ordonnee
WRITE(*, *) 'abscisse du centre :', rond%centre%abscisse
WRITE(*, *) 'périmètre :', 2*3.14*rond%rayon
```

exemple de 32 bits par adjonction de zéros (zero padding). On a donc intérêt à déclarer les composantes les plus volumineuses en tête de structure pour limiter ce remplissage. Enfin, il existe des options de compilation comme `-fpack-derived` sous `g95` ou `gfortran` pour l'éviter, au détriment des performances,

9.3 Imbrication et extension de structures

9.3.1 Imbrication de structures

Les structures peuvent comporter des composantes qui sont elles-mêmes des structures à condition que les types des composantes aient été définis préalablement. On peut par exemple définir une structure `cercle` dont un champ est une structure `point` :

```
TYPE cercle
  TYPE(point) :: centre ! de type défini au préalable
  REAL :: rayon
END TYPE cercle
```

et déclarer ensuite une variable de type `cercle` et lui affecter une valeur, par exemple grâce à son constructeur.

```
TYPE(cercle) :: rond
rond = cercle(b, 2.) ! où b est de type point
```

9.3.2 Extension d'un type dérivé

f2003 ⇒

La norme fortran 2003 prévoit aussi que l'on puisse étendre un type dérivé en lui adjoignant des composantes, grâce à l'attribut `EXTENDS`². Ce mécanisme permet d'hériter des outils conçus pour le type parent. Le constructeur du type étendu admet deux syntaxes (voir exemple ci-après) :

- la syntaxe classique (pour `circle1`) où l'on passe toutes les composantes ;
- le passage d'un objet du type parent (`a` de type `point`) et des seules composantes nécessaires (ici le rayon) pour l'étendre.

```
TYPE, EXTENDS(point) :: pt_ext_cerc
  REAL :: rayon
END TYPE pt_ext_cerc
TYPE(pt_ext_cerc) :: circle1, circle2
circle1 = pt_ext_cerc('B', 2., -0., 2.) ! constructeur du type étendu
circle2 = pt_ext_cerc(a, 2.) ! constructeur à partir du type parent
```

N.-B. : un type dérivé portant l'attribut `SEQUENCE` (cf. 9.1.1, p. 92) ou `BIND` (cf. 12.2 p. 116) n'est pas extensible.

L'accès aux composantes du type parent peut se faire soit directement, soit via le type parent :

```
WRITE(*,*) "circle1%lettre = ", circle1%lettre ! accès direct
WRITE(*,*) "circle1%point%lettre = ", circle1%point%lettre ! via type parent
```

9.4 Structures et tableaux

On peut définir des tableaux de structures :

```
TYPE(point), DIMENSION(10) :: courbe
```

définit un tableau de 10 éléments de type `point`. Alors `courbe%lettre` est un tableau de 10 caractères, alors que `courbe(2)` est une structure de type `point`.

On peut aussi définir des types dérivés dont certains champs sont des tableaux. En fortran 95, le profil de ces tableaux doit être connu au moment de la compilation.

2. Mais cette possibilité n'est pas encore implémentée dans tous les compilateurs, notamment dans `g95`.

```

TYPE point3d
  CHARACTER(LEN=1) :: lettre
  REAL, DIMENSION(3) :: coordonnees
END TYPE point3d
TYPE(point3d) :: m
m = point3d('M', (/1,2,3/))
PRINT *, m
PRINT *, 'coordonnée 2 de m ', m%coordonnees(2)

```

On peut aussi définir des tableaux de structures de ce type :

```

TYPE(point3d), DIMENSION(10) :: courbe3d

```

Alors `courbe3d%coordonnees(1)` est le tableau d'étendue 10 des coordonnées x des points de la courbe en trois dimensions.

9.5 Types dérivés à composantes allouables dynamiquement

Pour manipuler des structures comportant des tableaux de dimensions allouables dynamiquement en fortran 95, il était nécessaire de faire appel aux pointeurs (cf. chap. 11). Mais ces limitations ont disparu avec la norme fortran 2003 ainsi qu'avec les options spécifiques de certains compilateurs fortran 95³. ← f2003

Dans ce cas, l'appel du constructeur ainsi que l'opérateur d'affectation se chargent de l'allocation des composantes allouables. Mais il n'est pas possible de lire ou d'écrire globalement une variable de type dérivé à composante allouable ou d'attribut pointeur via `READ` ou `WRITE`.

```

TYPE point
  CHARACTER(LEN=1) :: lettre
  REAL, DIMENSION(:), ALLOCATABLE :: coord
END TYPE point
...
TYPE(point) :: a, b ! déclaration
a = point('A', (/1.,-1./)) ! construction de a => allocation de a%coord
WRITE(*,*) 'a ', a%lettre, a%coord ! affichage de a composante par composante
! WRITE(*,*) 'a ', a ! pas autorisé en fortran 2003
b = a ! affectation globale => allocation de b%coord

```

9.6 Procédures et opérateurs agissant sur les structures

La transmission de structures en argument de procédures externes suppose que les types dérivés soient connus de l'ensemble des procédures qui les manipulent⁴. La méthode la plus fiable consiste à insérer la définition du type dérivé dans un module (cf. 6.4, page 58) qui sera appelé par toutes les procédures utilisant ce type, de façon à éviter de dupliquer les définitions des structures (susceptibles de ne pas rester cohérentes lors de la mise au point du code).

```

1 MODULE m_point ! module de définition du type point
2   IMPLICIT NONE
3   TYPE point
4     CHARACTER(len=1) :: lettre

```

3. En particulier `xlf` d'IBM, le compilateur `NAG` et `g95` avec l'option `-std=f2003` ou `-std=f95` assortie de `-ftr15581` (cf. E.5.1, p. 153) autorisent les composantes allouables dans les structures.

4. Pour ce faire, on peut employer des interfaces explicites, mais il faut alors préciser l'attribut `SEQUENCE` lors de la définition du type dérivé.

```
5     REAL :: abscisse
6     REAL :: ordonnee
7 END TYPE point
8 END MODULE m_point
9 MODULE m_sym ! module de la procédure de calcul du symétrique
10  USE m_point
11  IMPLICIT NONE
12  CONTAINS
13  SUBROUTINE sym(m, n) ! sous programme de calcul du symétrique
14      TYPE(point), INTENT(in) :: m
15      TYPE(point), INTENT(out) :: n
16      n = point ( m%lettre, m%ordonnee, m%abscisse)
17  END SUBROUTINE sym
18 END MODULE m_sym
19 PROGRAM sym_point      ! programme principal
20  USE m_point ! pour connaître le type point
21  USE m_sym   ! pour connaître l'interface de la procédure sym
22  IMPLICIT NONE
23  TYPE(point) :: a, b ! déclaration
24  a = point('A', 1. , -1.) ! construction de a
25  WRITE(*,*) 'a ', a      ! affichage de a
26  CALL sym(a, b)         ! appel de sym
27  WRITE(*,*) 'b ', b     ! affichage de b
28 END PROGRAM sym_point
```

Tout l'intérêt des structures apparaît quand on peut les manipuler globalement par des procédures et des opérateurs. Ces possibilités comme la surdéfinition d'opérateurs (*cf.* 10.2.1, p.99) constituent une des richesses du fortran 90 en tant que langage de haut niveau.

Chapitre 10

Généricité, surcharge d'opérateurs

Dans ce chapitre, on introduit, à partir de quelques exemples, certains aspects novateurs du fortran 90, comme la généricité et la surcharge d'opérateurs, qui, associés aux modules, au contrôle de visibilité et aux pointeurs, permettent d'aborder les fonctionnalités d'un langage orienté objet.

10.1 Procédures génériques et spécifiques

Le fortran 77 comportait déjà la notion de procédure générique, mais elle était réservée aux procédures intrinsèques. Avec les versions 90 et au delà, cette notion devient accessible pour les procédures définies par l'utilisateur. En fortran¹, de nombreuses fonctions intrinsèques numériques invoquables sous leur nom *générique* font en réalité appel, suivant le type de leurs paramètres d'appel, à une version *spécifique* particulière. Par exemple, la fonction générique `MAX` référence les fonctions spécifiques :

- `MAX0` si les arguments sont entiers ;
- `AMAX1` si les arguments sont réels ;
- `DMAX1` si les arguments sont double précision.

Pour regrouper sous une même procédure générique plusieurs procédures spécifiques, il faut déclarer une interface commune obligatoirement *nommée* et lister ensuite les procédures spécifiques qui seront effectivement appelées, avec chacune leur interface. Au moment de l'appel de la procédure générique le choix de la procédure spécifique sera fait en fonction :

- du nombre des arguments effectifs d'appel ;
- et du type de ces arguments.

Par exemple dans le cas de fonctions² :

```
INTERFACE <nom_générique>
  FUNCTION <nom_spécifique_1(...)>
    ...
  END FUNCTION <nom_spécifique_1>
  ...
  FUNCTION <nom_spécifique_n(...)>
    ...
  END FUNCTION <nom_spécifique_n>
END INTERFACE <nom_générique>
```

1. En langage C, seule la norme C99 permet grâce à `#include <tgmath.h>`, d'utiliser un nom générique pour les fonctions **mathématiques**. Par exemple suivant le type de son argument, la fonction générique `exp` appellera la fonction spécifique `expf` pour un `float`, `exp` pour un `double` et `expl` pour un `long double`. La généricité est en revanche parfaitement disponible en C++.

2. Depuis fortran 95, il est possible de rappeler le nom générique à la fin de l'instruction `END INTERFACE`.

Pour définir une fonction générique `moy` qui calcule la moyenne (en flottant) d'un tableau de réels ou d'entiers, il faut définir deux fonctions spécifiques `moy_reel` et `moy_int` que l'on peut placer dans un module :

```

1  MODULE m_moyenne
2  IMPLICIT NONE
3  INTERFACE moy ! interface générique pour une famille de procédures spécifiques
4      MODULE PROCEDURE moy_reel ! version avec tableau de réels
5      MODULE PROCEDURE moy_int  ! version avec tableau d'entiers
6  ! le choix se fera sur le type de l'argument
7  END INTERFACE moy ! nommage possible en fortran 95 seulement
8  CONTAINS
9      FUNCTION moy_reel(tab_r) ! version réels
10     REAL :: moy_reel
11     REAL, DIMENSION(:), INTENT(in) :: tab_r
12     moy_reel = SUM(tab_r(:)) / SIZE(tab_r(:))
13  END FUNCTION moy_reel
14     FUNCTION moy_int(tab_i)  ! version entiers
15     REAL :: moy_int
16     INTEGER, DIMENSION(:), INTENT(in) :: tab_i
17     moy_int = REAL(SUM(tab_i(:))) / REAL(SIZE(tab_i(:)))
18  END FUNCTION moy_int
19  END MODULE m_moyenne
20
21  PROGRAM t_moyenne
22     USE m_moyenne
23     IMPLICIT NONE
24     INTEGER :: i
25     INTEGER, DIMENSION(10) :: ti = (/ (i, i=1, 10) /)
26     REAL, DIMENSION(10) :: tr
27     tr(:) = REAL(ti(:)) / 10.
28     ! appel de la procédure avec le nom générique (moy)
29     WRITE(*,*) moy( tr(:) ) ! argument tableau de réels => appelle moy_reel
30     WRITE(*,*) moy( ti(:) ) ! argument tableau d'entiers => appelle moy_int
31  END PROGRAM t_moyenne

```

Dans le cas de procédures externes dont le code source ne peut pas être modifié, il suffit de fournir l'interface des procédures spécifiques pour pouvoir les intégrer dans une interface générique. C'est ce qui est pratiqué pour les bibliothèques.

10.2 L'interface-opérateur

En fortran, les opérateurs intrinsèques sont soit monadiques, comme `+`, `-` et `.not.` suivis de leur opérande, soit plus souvent dyadiques et alors entourés par leurs deux opérands. À partir de fortran 90, il est possible :

- de définir de nouveaux opérateurs;
- et d'étendre la portée des opérateurs intrinsèques du fortran (*surcharge*) à des types pour lesquels leur action n'est pas définie par le langage.

Les procédures spécifiques qui permettent de définir l'action d'un opérateur rendent un résultat qui peut être utilisé dans une expression; elles doivent donc être des *fonctions* :

- à un argument (opération monadique);
- ou deux arguments (opération dyadique).

```

INTERFACE OPERATOR (<opérateur>)
    FUNCTION nom_spécifique_1(...)

```

```

...
END FUNCTION nom_spécifique_1
...
FUNCTION nom_spécifique_n(...)
...
END FUNCTION nom_spécifique_n
END INTERFACE

```

10.2.1 Surcharge d'opérateurs

Pour surcharger un opérateur, seules doivent être fournies les fonctions qui définissent le rôle de l'opérateur sur les types pour lesquels il n'est pas intrinsèquement défini. Le choix entre les fonctions spécifiques est fait en fonction du nombre des arguments (1 ou 2) et de leur type.

À titre d'exemple, l'opérateur // réalise la concaténation entre deux chaînes de caractères et on peut l'étendre, en tant qu'opérateur dyadique³ sur des réels, au calcul de la résistance équivalente à une association de deux résistances en parallèle.

```

1 MODULE m_resist
2 ! l'opérateur // est défini entre deux chaînes de caractères seulement
3 ! on le surcharge pour les réels (représentant ici des résistances par ex)
4 INTERFACE OPERATOR(//)
5     MODULE PROCEDURE parallele ! liste des fonctions qui étendent l'opérateur
6 END INTERFACE
7 CONTAINS
8     FUNCTION parallele(r1,r2)
9         IMPLICIT NONE
10        REAL :: parallele
11        REAL, INTENT(in) :: r1, r2
12        parallele = 1. / (1./r1 + 1./r2)
13        RETURN
14    END FUNCTION parallele
15 END MODULE m_resist
16 !
17 PROGRAM elec
18     USE m_resist
19     IMPLICIT NONE
20     REAL :: r1, r2
21     WRITE(*,*) ' entrer deux valeurs de résistances '
22     READ *, r1, r2
23     WRITE(*,*) 'r1 = ', r1, ' r2 = ', r2
24     WRITE(*,*) 'r1 '//série r2 = ', r1+r2 ! utilisation de // natif
25     WRITE(*,*) 'r1 // r2 = ', r1//r2 ! utilisation de // surchargé
26 END PROGRAM elec

```

10.2.2 Création d'opérateurs

Un opérateur nouveau est introduit (identificateur entouré par des points, comme les opérateurs de comparaison) suivant une syntaxe similaire à celle de la surcharge des opérateurs existants.

Par exemple, on peut définir l'opérateur `.para.` d'association en parallèle :

- pour les réels, en particulier pour les résistances ;
- pour un type dérivé `capacite` représentant les capacités.

3. L'opérateur // ne pourrait pas être défini comme monadique.

```

1 MODULE m_parallele
2 IMPLICIT NONE
3 TYPE capacite
4     REAL :: capa
5 END TYPE capacite
6 INTERFACE OPERATOR(.para.)
7 ! liste des fonctions qui définissent l'opérateur
8     MODULE PROCEDURE parallele_res    ! pour les réels (dont les résistances)
9     MODULE PROCEDURE parallele_capa  ! pour les capacités (type dérivé)
10 END INTERFACE
11 CONTAINS
12     FUNCTION parallele_res(r1,r2)
13         REAL :: parallele_res
14         REAL, INTENT(in) :: r1, r2
15         parallele_res = 1. / (1./r1 + 1./r2)
16         RETURN
17     END FUNCTION parallele_res
18     FUNCTION parallele_capa(c1,c2)
19         TYPE(capacite) :: parallele_capa
20         TYPE(capacite), INTENT(in) :: c1, c2
21         parallele_capa%capa = c1%capa + c2%capa
22         RETURN
23     END FUNCTION parallele_capa
24 END MODULE m_parallele
25 !
26 PROGRAM elec
27     USE m_parallele
28     IMPLICIT NONE
29     REAL :: r1, r2
30     TYPE(capacite) :: c1, c2, c3
31     WRITE(*,*) ' entrer deux valeurs de résistances '
32     READ *, r1, r2
33     WRITE(*,*) 'r1 = ', r1, ' r2 = ', r2
34     WRITE(*,*) 'r1 // r2 = ' , r1.para.r2    ! .para. appelle parallele_res
35     WRITE(*,*) ' entrer deux valeurs de capacités '
36     READ *, c1%capa, c2%capa
37     WRITE(*,*) 'c1 = ', c1%capa, ' c2 = ', c2%capa
38     c3 = c1.para.c2                          ! .para. appelle parallele_capa
39     WRITE(*,*) 'c1 // c2 = ' , c3%capa
40 END PROGRAM elec

```

10.3 L'interface-affectation

En fortran le symbole = n'est pas un opérateur de comparaison, mais le symbole de l'affectation d'une expression à une variable. Si le type de la variable est différent du type de l'expression, cette affectation se fait grâce à une conversion implicite de type.

Il est possible d'étendre la portée du symbole d'affectation à des types pour lesquels son sens n'est pas défini intrinsèquement. On parle alors de *surcharge de l'affectation* et on utilise une syntaxe particulière qui regroupe les interfaces des *sous-programmes* à deux arguments (le premier est la variable à affecter, le second l'expression qui lui sera affectée) qui définissent le sens de cette affectation. Le choix du sous-programme spécifique à appeler se fait en fonction du type des deux arguments.

Dans l'exemple suivant, on définit un type `hms`, heures, minutes, secondes. En surchargeant l'affectation (*cf.* ligne 10 et suivantes), on construit les conversions en secondes et heures fractionnaires

et les conversions réciproques. Via ces conversions, il est possible de surcharger les opérateurs +, - (monadique `opp_hms`, ligne 92 et dyadique `diff_hms`, ligne 82), * et >.

```

1  ! module de définition du type hms (heures, minutes, secondes)
2  MODULE m_hms
3  IMPLICIT NONE
4  TYPE hms
5     INTEGER :: heures
6     INTEGER :: minutes
7     INTEGER :: secondes
8  END TYPE hms
9  ! interfaces des opérateurs pour le type hms
10 ! surcharge de l'affectation pour conversion
11 INTERFACE ASSIGNMENT (=)
12     MODULE PROCEDURE hms2hfrac ! hms -> heures fractionnaires (type réel)
13     MODULE PROCEDURE hms2s    ! hms -> secondes (type entier)
14     MODULE PROCEDURE hfrac2hms ! heures fractionnaires (type réel) -> hms
15     MODULE PROCEDURE s2hms    ! secondes (type entier) -> hms
16 END INTERFACE
17 INTERFACE OPERATOR (+) ! surcharge de l'addition
18     MODULE PROCEDURE som_hms
19 END INTERFACE
20 INTERFACE OPERATOR (-) ! surcharge dyadique et monadique de "-"
21     MODULE PROCEDURE diff_hms
22     MODULE PROCEDURE opp_hms
23 END INTERFACE
24 INTERFACE OPERATOR (*) ! surcharge de la multiplication: entier par hms
25     MODULE PROCEDURE mult_hms
26 END INTERFACE
27 INTERFACE OPERATOR (>) ! surcharge de l'opérateur >
28     MODULE PROCEDURE sup_hms
29 END INTERFACE
30 CONTAINS
31     ! surcharges de l'affectation => par des sous-programmes
32     SUBROUTINE hfrac2hms(t_hms, h)
33     ! conversion des heures fractionnaires en heures, minutes, secondes
34     TYPE(hms), INTENT(out) :: t_hms
35     REAL, INTENT(in) :: h
36     REAL :: tmp
37     t_hms%heures = INT(h)
38     tmp = 60 * (h - t_hms%heures)
39     t_hms%minutes = FLOOR(tmp)
40     tmp = 60 * (tmp - t_hms%minutes)
41     t_hms%secondes = FLOOR(tmp)
42     RETURN
43 END SUBROUTINE hfrac2hms
44     SUBROUTINE hms2hfrac(h, t_hms)
45     ! conversion des heures, minutes, secondes en heures fractionnaires
46     TYPE(hms), INTENT(in) :: t_hms
47     REAL, INTENT(out) :: h
48     h = t_hms%secondes + 60*(t_hms%minutes + 60*t_hms%heures)
49     h = REAL(h)/3600.
50     RETURN
51 END SUBROUTINE hms2hfrac
52     SUBROUTINE hms2s(s, t_hms)
53     ! conversion des heures, minutes, secondes en secondes

```

```

54     INTEGER, INTENT(out):: s
55     TYPE(hms), INTENT(in) :: t_hms
56     s = t_hms%secondes + 60*(t_hms%minutes + 60*t_hms%heures)
57     RETURN
58 END SUBROUTINE hms2s
59 SUBROUTINE s2hms(t_hms, s)
60 ! conversion des secondes en heures, minutes, secondes
61     TYPE(hms), INTENT(out) :: t_hms
62     INTEGER, INTENT(in):: s
63     REAL :: tmp
64     t_hms%heures = INT(s/3600)
65     tmp = s - 3600*t_hms%heures
66     t_hms%minutes = INT(tmp/60)
67     tmp = tmp - 60*t_hms%minutes
68     t_hms%secondes = INT(tmp)
69     RETURN
70 END SUBROUTINE s2hms
71 ! surcharges des opérateurs => par des fonctions
72 FUNCTION som_hms(x, y)
73     TYPE(hms) :: som_hms
74     TYPE(hms), INTENT(in) :: x, y
75     INTEGER :: sx, sy, ss
76     sx = x ! conversion en secondes
77     sy = y ! conversion en secondes
78     ss = sx + sy
79     som_hms = ss ! conversion en hms
80     RETURN
81 END FUNCTION som_hms
82 FUNCTION diff_hms(x,y) ! "-" dyadique
83     TYPE(hms) :: diff_hms
84     TYPE(hms), INTENT(in) :: x, y
85     INTEGER :: sx, sy, ss
86     sx = x ! conversion en secondes
87     sy = y ! conversion en secondes
88     ss = sx - sy
89     diff_hms = ss ! conversion en hms
90     RETURN
91 END FUNCTION diff_hms
92 FUNCTION opp_hms(x) ! "-" monadique
93     TYPE(hms) :: opp_hms
94     TYPE(hms), INTENT(in) :: x
95     INTEGER :: sx
96     sx = x ! conversion en secondes
97     opp_hms = -sx ! conversion en hms
98     RETURN
99 END FUNCTION opp_hms
100 FUNCTION mult_hms(n, x)
101     TYPE(hms) :: mult_hms
102     INTEGER, INTENT(in) :: n
103     TYPE(hms), INTENT(in) :: x
104     INTEGER :: sx
105     sx = x ! conversion en secondes
106     mult_hms = n * sx ! conversion en hms
107     RETURN
108 END FUNCTION mult_hms
109 FUNCTION sup_hms(x, y)

```



```

110     LOGICAL :: sup_hms
111     TYPE(hms), INTENT(in) :: x, y
112     INTEGER :: sx, sy
113     sx = x ! conversion en secondes
114     sy = y ! conversion en secondes
115     sup_hms = sx > sy
116     RETURN
117 END FUNCTION sup_hms
118 END MODULE m_hms
119 !
120 PROGRAM t_hms
121 ! traitement des temps en heures, minutes, secondes
122 USE m_hms
123 IMPLICIT NONE
124 TYPE(hms) :: hms1, hms2, hms3, hms4, hms5
125 REAL :: h1, h2
126 INTEGER :: s1, s2
127 WRITE(*,*) 'entrez deux durées en heures minutes secondes'
128 READ *, hms1, hms2
129 h1 = hms1 ! conversion implicite en fraction d'heures
130 h2 = hms2 ! conversion implicite en fraction d'heures
131 s1 = hms1 ! conversion implicite en secondes
132 s2 = hms2 ! conversion implicite en secondes
133 hms3 = hms1 + hms2 ! addition de deux types hms
134 hms4 = hms1 - hms2 ! soustraction de deux types hms
135 hms5 = - 2*hms1 ! opposé d'un multiple entier
136 WRITE(*,*) ' en heures fractionnaires'
137 WRITE(*,*) ' h1 = ', h1, ' h2 = ', h2
138 WRITE(*,*) ' en secondes '
139 WRITE(*,*) ' s1 = ', s1, ' s2 = ', s2
140 WRITE(*,*) ' en heures minutes secondes'
141 WRITE(*,*) ' hms1 = ', hms1, ' hms2 = ', hms2
142 WRITE(*,*) ' somme en h m s ', hms3
143 WRITE(*,*) ' différence en h m s ', hms4
144 WRITE(*,*) ' -2 fois h1 en h m s ', hms5
145 IF (hms1 > hms2) THEN ! comparaison
146     WRITE(*,*) 'hms1 > hms2'
147 ELSE
148     WRITE(*,*) 'hms1 <= hms2'
149 END IF
150 END PROGRAM t_hms

```

Pour fiabiliser l'usage de ce type dérivé, il faudrait contraindre les composantes heures, minutes, secondes à posséder un signe commun. On pourrait par exemple interdire l'accès direct aux composantes (attribut `PRIVATE`) du type `hms`, quitte à définir des procédures de saisie et de communication des données `hms` précisément chargées de cet accès restreint.

Chapitre 11

Pointeurs

Historiquement, la notion de pointeur a été introduite dans d'autres langages, en particulier le C¹, pour manipuler les données via les adresses de la mémoire qu'elles occupent : dans cette vision, un pointeur est une variable typée ; le type permet de connaître la taille de la zone mémoire nécessaire et le mécanisme de codage et de décodage² pour stocker l'objet (variable, structure, ...) pointé³ ; le pointeur est destiné à stocker des valeurs qui sont les adresses des objets pointés, mais ces adresses seules seraient inutilisables sans la connaissance du type de l'objet pointé.

Mais la notion de pointeur en fortran s'apparente plus à un descripteur comportant non seulement une adresse, mais aussi d'autres informations sur la nature et la structure de la cible que le pointeur permet de désigner, notamment afin d'en assurer une gestion dynamique. La cible peut être une donnée de type quelconque (scalaire, tableau, structure, voire pointeur), ou une procédure, mais doit avoir été déclarée comme cible potentielle. D'autre part, fortran ne permet pas d'accéder explicitement à l'adresse que stocke le pointeur. D'ailleurs, il n'existe en fortran ni opérateur d'adresse⁴, ni d'opérateur d'indirection⁵ et la cible est désignée par le pointeur lui-même.

Dans la norme fortran 95, les pointeurs étaient nécessaires pour gérer les tableaux dynamiques dans certains contextes en tant que résultats de fonctions, arguments formels de procédures ou composantes de types dérivés. La norme 2003 apporte une gestion plus performante des tableaux dynamiques sans faire appel explicitement aux pointeurs. On réservera donc les pointeurs à des usages plus spécifiques comme la manipulation de structures dynamiques telles que les arbres ou les listes chaînées, ou, dans le cadre de la programmation objet, l'incorporation de méthodes dans les types dérivés grâce aux pointeurs de procédures.

11.1 Pointeurs, cibles et association

Une variable est considérée comme un *pointeur* si on précise l'attribut `POINTER` lors de sa déclaration ; noter que cette déclaration ne réserve alors pas d'espace pour stocker les valeurs qu'elle est susceptible de pointer.

1. Le tableau annexe (cf. D.7, p. 147) résume approximativement les syntaxes employées dans les deux langages pour manipuler des pointeurs.

2. Le type ne renseigne pas seulement sur la taille, mais aussi sur le codage de la valeur : par exemple si, sur une machine 32 bits, 4 octets permettent de stocker un entier par défaut et aussi un réel simple précision, le motif binaire qui représente par exemple la valeur numérique 10 ne sera pas le même en entier et en réel.

3. C'est cette information qui permet en C de faire de l'arithmétique sur les pointeurs : incrémentation, différence...

4. Si `var` est une variable, son adresse est désignée en C par `&var`. D'ailleurs, dans le cadre de l'interopérabilité entre fortran et C (cf. 12.3, p. 116), il faut faire appel en fortran, à une fonction spécifique du module `ISO_C_BINDING`, `c_loc` pour accéder à l'adresse d'une variable au sens du C.

5. Si `ptr` est un pointeur, la cible pointée est désignée par `*ptr` en C alors qu'en fortran, `ptr` désigne la cible.

11.1.1 Cible nommée

Un pointeur peut être *associé à une variable* de même type (et éventuellement sous-type) qualifiée de cible si celle-ci a été déclarée avec l'attribut `TARGET`⁶. L'instruction

```
ptr => trgt
```

associe le pointeur *ptr* à la cible *trgt*. Tant que cette association restera en vigueur, toute mention du pointeur *ptr* dans une expression fera en fait référence à la cible *trgt*.

Une variable pointeur peut aussi être associée à une autre variable pointeur de même type : dans ce dernier cas, par transitivité, l'association est faite avec la cible ultime, d'attribut `TARGET` c'est à dire non pointeur.

```

1 PROGRAM pointeurs
2 IMPLICIT NONE
3 INTEGER, TARGET :: i=1 , j=2 ! target obligatoire pour pointer vers i ou j
4 INTEGER, POINTER :: pi, pj, pk
5 ! association entre pointeur et cible : nécessite le même type
6 pi => i      ! pi pointe sur i
7 pj => j      ! pj pointe sur j
8 pk => i      ! pk pointe aussi sur i
9 WRITE(*, *) "associations : i, j = ", i, j, " pi, pj, pk = ", pi, pj, pk
10 ! affectation : conversions implicites possibles
11 pi = 3.5     ! affecte int(3.5) à la variable i, pointée par pi
12 pj = pi + 10 ! ajoute 10 à la variable pointée (i) et place le résultat dans j
13 pk => j      ! maintenant pk pointe sur j et plus sur i
14 WRITE(*, *) "affectations : i, j = ", i, j, " pi, pj, pk = ", pi, pj, pk
15 ! association d'un pointeur à un autre (en fait sa cible)
16 pi => pj     ! pi pointe maintenant sur la cible de pj, soit j
17 WRITE(*, *) "association d'un pointeur à un autre : pi, pj, pk = ", pi, pj, pk
18 END PROGRAM pointeurs

```

affichera

```

associations : i, j =  1 2  pi, pj, pk =  1 2 1
affectations : i, j =  3 13 pi, pj, pk =  3 13 13
association d'un pointeur à un autre : pi, pj, pk =  13 13 13

```

Fonctions concernant l'association des pointeurs

La fonction intrinsèque `ASSOCIATED` qui rend un booléen permet de s'informer sur l'état d'association d'un pointeur. Elle admet deux formes d'appel :

- `ASSOCIATED(<ptr>)` rend `.true.` si le pointeur `<ptr>` est associé ;
- `ASSOCIATED(<ptr>, <target>)` rend `.true.` si le pointeur `<ptr>` est associé à la cible `<target>`.

La déclaration d'un pointeur lui donne par défaut un statut d'association indéterminé. Il est donc recommandé de le désassocier explicitement dès la déclaration⁷ par `<ptr> => NULL()` ou, en début de programme par l'instruction `NULLIFY(<ptr>)`. ⇐ ♥

6. Cette restriction permet d'éviter de manipuler accidentellement des variables via des pointeurs d'une part et de laisser le compilateur optimiser plus efficacement les instructions manipulant des variables non accessibles via des pointeurs d'autre part. À l'inverse, le langage C n'impose pas cette précaution, mais la notion de pointeur restreint introduite en C99 a un objectif similaire.

7. L'option `-fpointer=null` du compilateur g95 (cf. E.5.1, p. 153) permet aussi d'initialiser à `null` les pointeurs non initialisés dans le code, mais ce n'est qu'une prudence supplémentaire.

```

REAL, POINTER :: ptr1 => NULL()
REAL, TARGET  :: r1, r2
WRITE(*, *) ASSOCIATED(ptr1)      ! affichera .false.
ptr1 => r1
WRITE(*, *) ASSOCIATED(ptr1)      ! affichera .true.
WRITE(*, *) ASSOCIATED(ptr1, r1) ! affichera .true.
WRITE(*, *) ASSOCIATED(ptr1, r2) ! affichera .false.

```

11.1.2 Cible dynamique anonyme

Une autre façon d'associer un pointeur est de lui allouer une zone mémoire, par l'instruction `ALLOCATE` (cf. ligne 3 du programme suivant), qui, dans le cas d'un pointeur, réserve l'espace mémoire et associe le pointeur à la mémoire réservée. Comme cette zone mémoire n'est pas désignée par une variable cible, la cible dynamique est qualifiée d'anonyme. Elle n'est accessible qu'au travers des pointeurs dont elle est la cible.

L'association cesse dès que l'on libère la zone mémoire par une instruction `DEALLOCATE` (cf. ligne 8). L'instruction `DEALLOCATE(<ptr>)` appliquée à un pointeur a donc deux effets :

- elle libère la mémoire allouée pour la cible anonyme ;
- elle désassocie le pointeur `<ptr>`.

Si on a, entre temps, associé d'autres pointeurs à cette même cible (cf. ligne 6), seul celui qui a permis la libération de mémoire est désassocié par l'instruction `DEALLOCATE` : les autres continuent de pointer vers une zone mémoire dont rien ne garantit qu'elle ne soit occupée par d'autres données...

♡ ⇒ Il est donc fortement conseillé de désassocier (cf. ligne 12), *tous* les pointeurs qui se partagent cette cible immédiatement après la désallocation.

```

1 PROGRAM alloc_assoc_pointeurs
2 REAL, POINTER :: ptr => NULL(), ptr2 => NULL()
3 ALLOCATE(ptr) ! reservation de mémoire et association de ptr
4 WRITE(*, *) "ptr associé ? ", ASSOCIATED(ptr) ! affichera .true.
5 ptr = .25 ! affectation d'une valeur à la cible
6 ptr2 => ptr ! ptr2 pointe aussi vers la cible allouée
7 WRITE(*, *) "ptr2 associé à ptr ?", ASSOCIATED(ptr2, ptr) ! affichera .true.
8 DEALLOCATE(ptr) ! libère la mémoire et desassocie ptr
9 WRITE(*, *) "ptr associé ? ", ASSOCIATED(ptr) ! affichera .false.
10 WRITE(*, *) "ptr2 associé ?", ASSOCIATED(ptr2) ! affichera .true.
11 WRITE(*, *) "ptr2 associé à ptr ?", ASSOCIATED(ptr2, ptr) ! affichera .false.
12 NULLIFY(ptr2) ! fortement conseillé
13 ! si ptr2 est désassocié, l'instruction suivante provoquerait une erreur
14 ! si ptr2 n'était pas désassocié, elle donnerait un résultat aléatoire
15 ! WRITE(*, *) "cible de ptr2 ", ptr2
16 END PROGRAM alloc_assoc_pointeurs

```

Mais si on désassocie *tous* les pointeurs associés à une cible anonyme allouée sans instruction `DEALLOCATE`, la zone mémoire allouée reste réservée, mais n'est définitivement plus accessible : on parle alors de fuite de mémoire (memory leak). Si cette fuite de mémoire se produit de façon répétitive dans une boucle par exemple, elle peut provoquer un dépassement de capacité de la mémoire. On veillera donc à éviter de telles pertes de mémoire dans les procédures.

```

1 PROGRAM ALLOC_PTR
2 IMPLICIT NONE
3 REAL, POINTER :: ptr => NULL()
4 ALLOCATE(ptr) ! allocation du pointeur
5 WRITE(*, *) ASSOCIATED(ptr) ! affichera .true.
6 ptr = 2 ! utilisation de la mémoire allouée
7 NULLIFY(ptr) ! désassociation avant désallocation ! => memory leak

```

```

8 WRITE(*, *) ASSOCIATED(ptr) ! affichera .false.
9 END PROGRAM ALLOC_PTR

```

Lors de l'exécution, le programme précédent compilé avec g95 affichera T (true) puis F (false) et signalera cette fuite de mémoire : Remaining memory: 4 bytes allocated at line 4

Noter qu'affecter une valeur à un pointeur non associé n'est pas autorisé, car rien ne spécifie alors la zone mémoire où stocker la valeur.

11.2 Pointeurs sur des tableaux

L'attribut pointeur peut être donné à une variable tableau à condition que seul son rang et non son profil (*cf.* 7.1.1, 70) soit spécifié.

```

INTEGER, DIMENSION(10), POINTEUR :: ptab ! est interdit
INTEGER, DIMENSION(:), POINTEUR  :: ptab ! est autorisé

```

Dans le cas de tableaux dont l'indice ne commence pas à 1, on distinguera les deux formes d'association :

```

INTEGER, DIMENSION(-2:2), TARGET :: tab
INTEGER, DIMENSION(:), POINTER   :: pt1, pt2
pt1 => tab      ! qui conserve l'indexation initiale
pt2 => tab(:)   ! qui renumérote à partir de 1 car tab(:) est une section de tab

```

Plus généralement, il est enfin possible de pointer vers une section régulière de tableau, de rang inférieur ou égal à celui du tableau dont la section est issue.

```

1 PROGRAM section_ptr
2 INTEGER, DIMENSION(3,5), TARGET :: mat
3 INTEGER, DIMENSION(:), POINTER  :: pt1 => NULL() ! rang 1
4 INTEGER, DIMENSION(:, :), POINTER :: pt2 => NULL() ! rang 2
5 INTEGER :: i, j
6 WRITE(*,*) "matrice initiale"
7 DO i=1, 3
8     mat(i,:) = i* (/ (j, j=1, 5) /)
9     WRITE(*,*) mat(i,:)
10 END DO
11 pt1 => mat(2,:)
12 pt2 => mat(1:3:2, 2:4)
13 WRITE(*,*) "deuxième ligne de mat"
14 WRITE(*,*) pt1
15 WRITE(*,*) "sous-matrice de mat (lignes 1 et 3 , colonnes 2 à 4)"
16 DO i=1, 2
17     WRITE(*,*) pt2(i,:)
18 END DO
19 END PROGRAM section_ptr

```

affiche

```

matrice initiale
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15

```

```

deuxième ligne de mat
2 4 6 8 10
sous-matrice de mat (lignes 1 et 3 , colonnes 2 à 4)
2 3 4
6 9 12

```

Il y aura alors réindexation à partir de 1 de la section pointée sauf si on désigne l'indice de départ du pointeur⁸.

```
pt2(1:, 2:) => mat(1:3:2, 2:4)
```

Exemple de fonction à valeur pointeur

Une fonction peut produire un résultat tableau avec l'attribut pointeur⁹ : cela lui permet de rendre un tableau dont le profil dépend de calculs effectués dans la fonction. Dans ce cas, il ne faut pas oublier (après usage) de désallouer dans l'appelant le tableau alloué par la fonction.

```

1 MODULE m_fct_ptr
2 IMPLICIT NONE
3 CONTAINS
4 FUNCTION positifs(x) ! fonction à valeur pointeur sur un tableau de rang 1
5 INTEGER, DIMENSION(:), POINTER :: positifs
6 INTEGER, DIMENSION(:), INTENT(in) :: x
7 INTEGER :: p, ok, i, j
8 p = COUNT( x>0 ) ! comptage du nombre de positifs
9 ALLOCATE (positifs(p), stat=ok) ! allocation du pointeur
10 IF(ok /= 0) STOP 'erreur allocation'
11 j = 0
12 DO i=1, SIZE(x)
13   IF (x(i) <= 0 ) CYCLE
14     j = j +1
15     positifs(j) = x(i) ! affectation du pointeur
16 END DO
17 END FUNCTION positifs
18 END MODULE m_fct_ptr
19
20 PROGRAM ppal
21 USE m_fct_ptr
22 IMPLICIT NONE
23 INTEGER :: i
24 INTEGER, DIMENSION(10) :: t=(/(i, i=-5, 4) /)
25 INTEGER, DIMENSION(:), POINTER :: tp => NULL()
26 WRITE(*, *) "tableau initial ", t(:)
27 tp => positifs(t) ! appel de la fonction qui alloue le pointeur
28 WRITE(*, *) size(tp), " éléments positifs :", tp(:)
29 DEALLOCATE(tp) ! pour libérer la mémoire
30 END PROGRAM ppal

```

qui affiche

```

tableau initial  -5 -4 -3 -2 -1 0 1 2 3 4
4 positifs : 1 2 3 4

```

8. Cette possibilité n'est pas implémentée sous gfortran 4.4.3 ni sous g95.

9. C'était la seule méthode en fortran 95 standard qui n'autorisait pas les fonctions à résultat tableau allouable. Mais ces restrictions n'existent plus en fortran 2003, ou grâce à certaines extensions propres des compilateurs, en particulier `-ftr15581` pour le compilateur `g95` (cf. E.5.1, p. 153).

Les pointeurs permettent de nombreuses autres applications dans le domaine des données dynamiques, notamment pour gérer les listes chaînées.

11.3 Tableaux de types dérivés contenant des pointeurs

Lorsqu'un pointeur possède un attribut de dimension, son profil qui détermine la taille de la cible ne peut pas être fixé à la déclaration (on parle de profil différé). Il n'est donc pas possible de définir directement des tableaux de pointeurs. Mais il suffit d'encapsuler le pointeur dans un type dérivé pour pouvoir simuler un tableau de pointeurs avec un tableau d'éléments du type dérivé qui ne contient que le pointeur. Il est ainsi possible de représenter des assemblages « non-rectangulaires » d'éléments du même type, qui ne sont pas représentables sous forme de tableaux. Cela vaut par exemple pour les parties triangulaires inférieure et supérieure d'une matrice.

```

1 PROGRAM lower_upper_alloc
2 IMPLICIT NONE
3 ! pas de tableau de pointeurs => les encapsuler dans un type dérivé
4 TYPE iptr
5   INTEGER, DIMENSION(:), POINTER :: pt => null()
6   ! pointeur de tableau 1D d'entiers
7 END TYPE iptr
8 TYPE(iptr), DIMENSION(:), POINTER :: lower=> null(), upper => null()
9 ! tableaux d'éléments de type iptr pour stocker les parties triang. inf et sup
10 INTEGER, ALLOCATABLE, DIMENSION(:, :) :: mat
11 INTEGER :: ligne, colonne, n
12 DO
13   WRITE(*,*) "entrer la dimension de la matrice carrée"
14   READ(*,*) n
15   IF (n <= 0) EXIT
16   ALLOCATE(mat(n,n))
17   DO ligne = 1, n
18     DO colonne = 1, n
19       mat(ligne, colonne) = 10*ligne + colonne
20     END DO
21   END DO
22   WRITE(*,*) "matrice complète"
23   DO ligne = 1, n
24     WRITE(*,*) mat(ligne, :)
25   END DO
26   ALLOCATE(lower(n)) ! allocation du tableau-pointeur de n lignes
27   ALLOCATE(upper(n)) ! allocation du tableau-pointeur de n lignes
28   DO ligne = 1, n
29     ALLOCATE(lower(ligne)%pt(ligne)) ! une allocation par ligne
30     ALLOCATE(upper(ligne)%pt(n-ligne +1)) ! une allocation par ligne
31     lower(ligne)%pt = mat(ligne, 1:ligne) ! affectation ligne à ligne
32     upper(ligne)%pt = mat(ligne, ligne:n) ! affectation ligne à ligne
33   END DO
34   DEALLOCATE(mat) ! libération de la matrice
35   WRITE(*,*) "partie triangulaire inférieure"
36   DO ligne = 1, n
37     WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
38   END DO
39   WRITE(*,*) "partie triangulaire supérieure (non alignée)"
40   DO ligne = 1, n
41     WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
42   END DO
43   DO ligne = 1, n
44     DEALLOCATE(lower(ligne)%pt, upper(ligne)%pt)
45   END DO

```

```

46 DEALLOCATE(lower, upper)
47 END DO
48 END PROGRAM lower_upper_alloc

```

```

    entrer la dimension de la matrice carrée
4
matrice complète
11 12 13 14
21 22 23 24
31 32 33 34
41 42 43 44
partie triangulaire inférieure
11
21 22
31 32 33
41 42 43 44
partie triangulaire supérieure (non alignée)
11 12 13 14
22 23 24
33 34
44

```

Dans l'exemple précédent (programme `lower_upper_alloc`), les tableaux de pointeurs `lower` et `upper` sont alloués (ligne 26, p. 109), après le choix de la matrice `mat`; puis, les lignes des parties triangulaires inférieure et supérieure sont allouées (ligne 29, p. 109), et enfin les éléments correspondants sont recopiés (ligne 31, p. 109). Il y a donc duplication de l'espace mémoire de la matrice. On peut alors libérer l'espace alloué pour la matrice (ligne 34, p. 109) et continuer à utiliser les parties inférieure et supérieure, `lower` et `upper`.

Mais on peut aussi, comme dans la version `lower_upper_ptr` qui suit, se contenter de désigner les éléments correspondants de la matrice comme cibles (ligne 29, p. 111) des pointeurs des tableaux `lower` et `upper` : dans ce cas, ils suivent les modifications (ligne 40, p. 111) des coefficients de la matrice. Il est alors prudent de les désassocier à la libération de la matrice (ligne 50, p. 111).

```

1 PROGRAM lower_upper_ptr
2 IMPLICIT NONE
3 ! pas de tableau de pointeurs => les encapsuler dans un type dérivé
4 TYPE iptr
5   INTEGER, DIMENSION(:), POINTER :: pt => null()
6   ! pointeur de tableau 1D d'entiers
7 END TYPE iptr
8 TYPE(iptr), DIMENSION(:), POINTER :: lower=> null(), upper => null()
9 ! tableaux d'éléments de type iptr pour stocker les parties triang. inf et sup
10 INTEGER, ALLOCATABLE, DIMENSION(:,:), TARGET :: mat
11 INTEGER :: ligne, colonne, n
12 DO
13   WRITE(*,*) "entrer la dimension de la matrice carrée"
14   READ(*,*) n
15   IF (n <= 0) EXIT
16   ALLOCATE(mat(n,n))
17   DO ligne = 1, n
18     DO colonne = 1, n
19       mat(ligne, colonne) = 10*ligne + colonne
20     END DO
21   END DO
22   WRITE(*,*) "matrice complète"
23   DO ligne = 1, n

```



```

24     WRITE(*,*) mat(ligne, :)
25 END DO
26 ALLOCATE(lower(n)) ! allocation du tableau-pointeur de n lignes
27 ALLOCATE(upper(n)) ! allocation du tableau-pointeur de n lignes
28 DO ligne = 1, n      ! désignation de la cible des pointeurs
29     lower(ligne)%pt => mat(ligne, 1:ligne) !
30     upper(ligne)%pt => mat(ligne, ligne:n)
31 END DO
32 WRITE(*,*) "partie triangulaire inférieure"
33 DO ligne = 1, n
34     WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
35 END DO
36 WRITE(*,*) "partie triangulaire supérieure (non alignée)"
37 DO ligne = 1, n
38     WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
39 END DO
40 mat = - mat ! modification de la matrice
41 WRITE(*,*) "mat change de signe"
42 WRITE(*,*) "partie triangulaire inférieure"
43 DO ligne = 1, n
44     WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
45 END DO
46 WRITE(*,*) "partie triangulaire supérieure (non alignée)"
47 DO ligne = 1, n
48     WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
49 END DO
50 DEALLOCATE(mat) ! libération de la matrice après usage des parties sup/inf
51 DO ligne = 1, n ! par prudence (ils pointent alors vers une zone désallouée)
52     NULLIFY(lower(ligne)%pt, upper(ligne)%pt)
53 END DO
54 DEALLOCATE(lower, upper)
55 END DO
56 END PROGRAM lower_upper_ptr

```

```

entrer la dimension de la matrice carrée
3
matrice complète
11 12 13
21 22 23
31 32 33
partie triangulaire inférieure
11
21 22
31 32 33
partie triangulaire supérieure (non alignée)
11 12 13
22 23
33
mat change de signe
partie triangulaire inférieure
-11
-21 -22
-31 -32 -33
partie triangulaire supérieure (non alignée)
-11 -12 -13
-22 -23
-33

```

♠ 11.4 Manipulation de listes chaînées

Les composantes des types dérivés peuvent posséder l'attribut pointeur, ce qui permet la constitution de listes chaînées : les listes chaînées permettent de représenter des ensembles ordonnés d'éléments (appelés noeuds de la liste) dont non seulement le contenu, mais aussi le nombre peut évoluer en cours de l'exécution du programme. On peut en particulier insérer ou supprimer des éléments dans la liste tout en conservant l'ordre : l'allocation se fait donc élément par élément, contrairement à ce qui passe pour les tableaux où l'allocation, si elle peut être dynamique, reste globale. Mais, l'accès à un élément donné de la liste nécessite de parcourir tous les précédents alors qu'on peut accéder directement à un élément d'indice donné d'un tableau.

Dans une liste simplement chaînée, chaque élément comporte un pointeur vers l'élément suivant sauf le pointeur du dernier élément de la liste qui est nul.

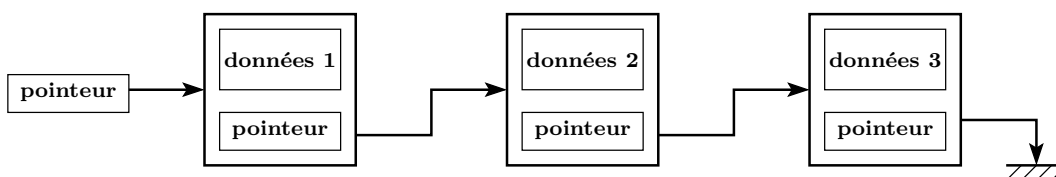


FIGURE 11.1 – Liste simplement chaînée à trois éléments : le dernier élément pointe vers NULL.

L'exemple suivant montre une liste chaînée permettant de représenter une courbe dans le plan : le type dérivé `point` comporte donc un pointeur vers une donnée de type `point` pour chaîner la liste. Pour simplifier, la courbe initialement créée est la deuxième bissectrice des axes. Les procédures d'impression `imprime_chaine` (ligne 10, p. 112) et de libération `libere_chaine` (ligne 26, p.113) de la liste sont ici implémentées sous forme récursive. Pour libérer la mémoire, il est nécessaire de commencer par la fin de la liste afin d'éviter de perdre l'accès aux éléments suivants. En revanche, l'impression de la liste se fait dans l'ordre si on imprime avant l'appel récursif (en « descendant » la récurrence) ou dans l'ordre inverse (procédure `imprime_chaine_inverse` (ligne 18, p. 112) si on imprime après (en « remontant » la récurrence). Enfin, les opérations d'insertion, `insere_point` (ligne 57, p. 113) ou de suppression, `supprime_point` (ligne 35, p. 113) d'un point intérieur se font dans une boucle qui parcourt l'ensemble de la liste jusqu'au pointeur nul signifiant l'accès au dernier point.

```

1  MODULE m_chaine
2  IMPLICIT NONE
3  TYPE point
4  ! définition du type dérivé point avec un pointeur vers le suivant
5  REAL :: x
6  REAL :: y
7  TYPE(point), POINTER :: next => null() ! f95 seulement (par prudence)
8  END TYPE point
9  CONTAINS
10 RECURSIVE SUBROUTINE imprime_chaine(courant) !
11     TYPE(point), POINTER :: courant
12     ! affichage en commençant par le début : imprimer avant la récursion
13     WRITE(*,*) courant%x, courant%y
14     IF(ASSOCIATED(courant%next)) THEN
15         CALL imprime_chaine(courant%next)
16     END IF
17 END SUBROUTINE imprime_chaine
18 RECURSIVE SUBROUTINE imprime_chaine_inverse(courant) !
19     TYPE(point), POINTER :: courant
20     ! affichage en commençant par la fin : dérouler la récursion avant
21     IF(ASSOCIATED(courant%next)) THEN

```

```

22     CALL imprime_chaine_inverse(courant%next)
23     END IF
24     WRITE(*,*) courant%x, courant%y
25     END SUBROUTINE imprime_chaine_inverse
26     RECURSIVE SUBROUTINE libere_chaine(courant)!
27     TYPE(point), POINTER :: courant
28     ! attention libérer en commençant par la fin
29     IF(ASSOCIATED(courant%next)) THEN
30         CALL libere_chaine(courant%next)
31     END IF
32     WRITE(*,*) "libération du point de coordonnées", courant%x, courant%y
33     DEALLOCATE(courant)
34     END SUBROUTINE libere_chaine
35     SUBROUTINE supprime_point(debut) !
36     ! suppression d'un point (intérieur seulement)
37     TYPE(point), POINTER :: debut
38     TYPE(point), POINTER :: courant=>null(), suivant=>null()
39     TYPE(point), POINTER :: precedent=>null()
40     INTEGER :: suppr
41     precedent => debut
42     courant => precedent%next
43     DO
44         suivant => courant%next
45         WRITE(*,*) courant%x, courant%y, "supprimer ? 1 si oui"
46         READ(*,*) suppr
47         IF (suppr == 1) THEN
48             precedent%next => courant%next ! court-circuiter le point courant
49             DEALLOCATE(courant) ! libération du point courant
50         ELSE
51             precedent => courant
52         ENDIF
53         courant => suivant
54         IF (.NOT.ASSOCIATED(suivant%next)) EXIT
55     END DO
56     END SUBROUTINE supprime_point
57     SUBROUTINE insere_point(debut) !
58     ! ajout d'un point (intérieur seulement)
59     TYPE(point), POINTER :: debut
60     TYPE(point), POINTER :: nouveau=>null(), suivant=>null()
61     TYPE(point), POINTER :: precedent=>null()
62     INTEGER :: ajout
63     REAL :: x, y
64     precedent => debut
65     DO
66         suivant => precedent%next
67         WRITE(*,*) precedent%x, precedent%y, "insérer ? 1 si oui"
68         READ(*,*) ajout
69         IF(ajout == 1) THEN
70             ALLOCATE(nouveau)
71             nouveau%next => precedent%next
72             precedent%next => nouveau
73             WRITE(*,*) "entrer x et y"
74             READ(*,*) x, y
75             nouveau%x = x
76             nouveau%y = y
77         ENDIF

```

```

78     precedent => suivant
79     IF (.NOT.ASSOCIATED(suivant%next)) EXIT
80     END DO
81     END SUBROUTINE insere_point
82 END MODULE m_chaine

```

```

84 PROGRAM linked_list
85 USE m_chaine
86 INTEGER :: i, n
87 TYPE(point), POINTER :: courant => null(), suivant=>null()
88 TYPE(point), POINTER :: debut => null()
89 ! construction d'une liste chaînée de n points
90 WRITE(*,*) 'entrer le nb de points'
91 READ(*,*) n
92 ! premier point qui permettra l'accès à toute la chaîne
93 ALLOCATE(debut)
94 courant => debut
95 DO i = 1, n-1
96     courant%x = REAL(i)
97     courant%y = -REAL(i)
98     ALLOCATE(suivant)
99     courant%next => suivant
100    courant => suivant
101 END DO
102 ! dernier point sans successeur
103 courant%x = REAL(n)
104 courant%y = -REAL(n)
105 courant%next => null() ! fin de la liste
106 WRITE(*,*) "impression dans le sens direct"
107 CALL imprime_chaine(debut) ! impression dans le sens direct
108 WRITE(*,*) "impression dans le sens inverse"
109 CALL imprime_chaine_inverse(debut) ! impression en sens inverse
110 WRITE(*,*) "suppression de points"
111 CALL supprime_point(debut) ! suppression de points intérieurs
112 WRITE(*,*) "impression dans le sens direct"
113 CALL imprime_chaine(debut) ! impression dans le sens direct
114 WRITE(*,*) "insertion de points"
115 CALL insere_point(debut) ! insertion de points intérieurs
116 WRITE(*,*) "impression dans le sens direct"
117 CALL imprime_chaine(debut) ! impression dans le sens direct
118 WRITE(*,*) "libération de la chaîne"
119 CALL libere_chaine(debut) ! libération de la chaîne
120 END PROGRAM linked_list

```

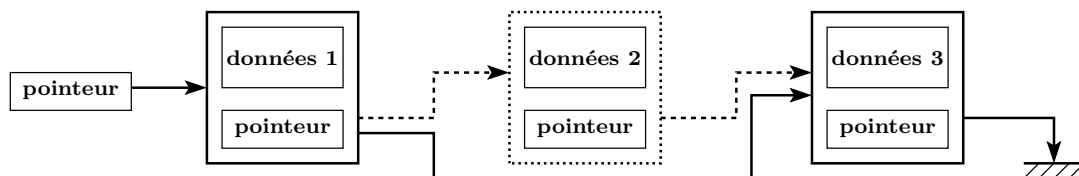


FIGURE 11.2 – Suppression d'un élément intérieur d'une liste simplement chaînée à trois éléments

Chapitre 12

Inter-opérabilité avec le C

La standardisation de l'interfaçage entre les langages C et fortran est intervenue seulement avec la norme fortran 2003. Elle permet de rendre portables les appels de fonctions en C depuis le fortran et réciproquement. Elle s'appuie sur la définition en fortran d'entités (types, variables et procédures) dites inter-opérables car équivalentes à celles du langage C.

La construction d'un exécutable à partir de code C et de code fortran nécessite une compilation séparée avec des compilateurs C et fortran compatibles, puis une édition de lien qui fournisse toutes les bibliothèques nécessaires. En particulier dans le cas où le programme principal est en C, il faut spécifier explicitement la bibliothèque du fortran

lors de l'édition de lien si elle est lancée par le compilateur C¹. Une autre solution consiste à confier l'édition de lien au compilateur fortran associé, quitte à lui indiquer par une option² que le programme principal n'est pas en fortran.

12.1 Interopérabilité des types intrinsèques

Le module intrinsèque `iso_c_binding` définit les paramètres de codage (`kind`) sous forme de constantes symboliques pour les sous-types intrinsèques inter-opérables avec le langage C. Le tableau suivant décrit les équivalences entre les principaux types inter-opérables du fortran 2003 et ceux de la norme C89 du langage C :

C89	fortran 2005
<code>char</code>	<code>CHARACTER(kind=c_char)</code>
<code>short_int</code>	<code>INTEGER(kind=c_short)</code>
<code>int</code>	<code>INTEGER(kind=c_int)</code>
<code>long int</code>	<code>INTEGER(kind=c_long)</code>
<code>long long int</code>	<code>INTEGER(kind=c_lon_long)</code>
<code>size_t</code>	<code>INTEGER(kind=c_size_t)</code>
<code>float</code>	<code>REAL(kind=c_float)</code>
<code>double</code>	<code>REAL(kind=c_double)</code>
<code>long double</code>	<code>REAL(kind=c_long_double)</code>

Ce module définit de même des types complexes et un type booléen inter-opérables respectivement avec les types complexes et le type booléen du C99.

1. Avec le compilateur `g95` sous linux et un processeur Intel 32 bits, on lancera par exemple, sous réserve que la bibliothèque fortran soit installée dans ce répertoire,
`gcc partie_c.o partie_fortran.o -L/usr/lib/g95/lib/gcc-lib/i686-pc-linux-gnu/4.0.3/ -lf95` suivi éventuellement de `-lm`

De plus on s'assurera de la compatibilité des versions des compilateurs utilisés.

2. Par exemple `gfortran` ou `g95` avec `gcc`, ou `ifort` et l'option `-nofor-main` (cf. E.4.1, page 152) avec `icc`.

C99	fortran 2005
float _Complex	COMPLEX(kind=c_float_complex)
double _Complex	COMPLEX(kind=c_double_complex)
long double _Complex	COMPLEX(kind=c_long_double_complex)
_bool	LOGICAL(kind=c_bool)

Il définit aussi des types inter-opérables équivalents aux variantes de type entier définies en C99, `int8_t`, `int16_t`, `int32_t` et `int64_t` à nombre de bits fixé ainsi que les versions à nombre minimal de bits `int_least8_t` et suivants ou les plus rapides avec un nombre de bits fixé `int_fast8_t` et suivants : le nommage en fortran est obtenu par adjonction du préfixe `c_`, par exemple `c_int8_t`.

Le module `iso_c_binding` définit enfin des constantes caractères de paramètre de codage (`kind`) `c_char` pour les caractères spéciaux du C, comme `c_null_char` ou `c_new_line`.

12.2 Inter-opérabilité des types dérivés

Les types dérivés du fortran peuvent être rendus inter-opérables avec les structures du C à condition que³ :

- les composantes du type dérivé fortran soient inter-opérables, publiques, qu'elles ne soient pas allouables et ne comportent pas de pointeur (au sens de l'attribut `pointer` en fortran⁴) ;
- Le type dérivé fortran ne soit pas une extension de type (cf. 9.3.2, p. 94), ne soit pas de type `SEQUENCE` (cf. 9.1.1, p. 92), ni un type paramétré et ne possède pas de procédures attachées ;
- l'attribut `bind(c)` soit spécifié en fortran dans la définition du type dérivé ;
- la structure du C ne comporte pas de champ de bits.

12.3 Inter-opérabilité avec les pointeurs du C

Le module `iso_c_binding` définit des types dérivés (à composantes privées) `c_ptr` et `c_funptr` inter-opérables respectivement avec les pointeurs d'objets et de fonctions du C. Les constantes nommées `c_null_ptr` et `c_null_funptr` sont les équivalents en fortran du pointeur nul du C.

Ce module définit aussi des procédures de manipulation des pointeurs du C depuis le fortran.

- `c_loc(x)` renvoie une valeur du type `c_ptr` contenant l'adresse au sens du C de son argument. Cette fonction joue le rôle de l'opérateur `&` du langage C.
- `c_funloc(f)` renvoie l'adresse d'une procédure inter-opérable.
- `c_f_pointer(cp_ptr, fp_ptr, profil)` est un sous-programme qui permet de traduire un pointeur du C et un profil éventuel en un pointeur au sens du fortran. L'argument d'entrée `cp_ptr` est du type `c_ptr`. L'argument de sortie `fp_ptr` est un pointeur vers un type inter-opérable. En sortie, `fp_ptr` est associé à la cible de `cp_ptr`. L'argument d'entrée `profil` en général optionnel, est requis si `fp_ptr` est un tableau.
- `c_f_procpointer(cp_ptr, fp_ptr)` est un sous-programme qui effectue la même conversion que `c_f_pointer` mais pour des procédures.

12.4 Inter-opérabilité des procédures

L'inter-opérabilité des procédures entre fortran et C requiert que les paramètres échangés soient inter-opérables, que l'interface de la procédure fortran soit explicite et déclarée avec l'attribut `BIND(C)`. L'attribut `BIND` permet aussi de spécifier le nom de la fonction C associée grâce au mot clef `NAME` selon la syntaxe : `BIND(C, NAME=nom_de_la_fonction_en_C)`. À défaut du paramètre `NAME` dans l'attribut `BIND` le nom en C est le même qu'en fortran, mais en minuscule⁵.

Elle exclut notamment :

- les procédures internes du fortran ;

3. L'espace occupé par les structures est soumis à des contraintes d'alignement en mémoire qui conduisent parfois à compléter une structure par des octets de remplissage. Ces contraintes diffèrent suivant les langages, les processeurs et les compilateurs et peuvent ainsi affecter l'inter-opérabilité des structures.

4. Mais un type dérivé inter-opérable peut comporter une composante de type `c_ptr` ou `c_funptr`.

5. On rappelle qu'en fortran, la casse n'est pas distinguée, cf. 1.3.2, p. 5.

- les arguments optionnels en fortran et le nombre variable d'arguments en C ;
- les résultats de fonction non scalaires en fortran ;

Les sous-programmes du fortran correspondent aux fonctions sans valeur de retour (type `void`) en C.

12.4.1 Le passage par copie de valeur (value)

Comme le passage de paramètre en C se fait par copie de valeur, alors qu'il se fait par référence en fortran, le fortran 2003 a introduit l'attribut `VALUE` pour modifier le mécanisme de passage d'argument du fortran dans le sens du C. Cette méthode permet de travailler sur la copie passée en argument sans que cela affecte la valeur côté appelant en retour. L'attribut `VALUE` peut être utilisé en dehors de l'inter-opérabilité avec le C mais n'est pas applicable aux paramètres pointeurs ou allouables et est évidemment incompatible avec les vocations `INTENT(OUT)` et `INTENT(INOUT)`.

À l'inverse, un argument sans l'attribut `VALUE` doit correspondre dans la fonction en C à un pointeur vers le type de l'argument (inter-opérable) du fortran.

12.4.2 Exemple : appel de fonctions C depuis le fortran

L'exemple suivant montre l'appel en fortran de fonctions C sans valeur de retour pour saisir des réels et les afficher. Elles sont vues du fortran comme des sous-programmes de même nom en l'absence du paramètre `NAME` dans l'attribut `BIND` (cf. lignes 6 et 12).

L'accès au type `c_float` dans l'interface fortran (cf. lignes 8 et 14) nécessite soit un `USE`, soit un `IMPORT` (cf. 6.4.4, p. 62).

Le sous-programme d'affichage `c_write_float` peut se contenter d'un passage par copie de valeur, d'où l'attribut `VALUE` en fortran (cf. ligne 15), car il ne modifie pas la valeur du paramètre. En revanche, la fonction de lecture, `c_read_float`, nécessite un passage par pointeur en C, directement compatible avec le passage d'argument utilisé par défaut en fortran.

```

----- Partie fortran 2003 -----
1  MODULE m_interf
2    USE, INTRINSIC :: iso_c_binding, ONLY:c_float
3    IMPLICIT NONE
4    INTERFACE
5      ! void c_read_float(float *pf);
6      SUBROUTINE c_read_float(f) BIND(c) !
7        ! USE, INTRINSIC :: iso_c_binding, ONLY:c_float
8        IMPORT :: c_float ! préférable au USE !
9        REAL(kind=c_float), INTENT(out) :: f
10     END SUBROUTINE c_read_float
11     ! void c_write_float(float f);
12     SUBROUTINE c_write_float(f) BIND(c) !
13       ! USE, INTRINSIC :: iso_c_binding, ONLY:c_float
14       IMPORT :: c_float ! préférable au USE !
15       REAL(kind=c_float), intent(in), value :: f ! copie de valeur
16     END SUBROUTINE c_write_float
17   END INTERFACE
18 END MODULE m_interf
19
20 PROGRAM f_appel_c_sub_io
21   USE m_interf
22   REAL(kind=c_float) :: r
23   CALL c_read_float(r)
24   WRITE (*,*) "float en fortran après lecture en C ", r
25   CALL c_write_float(r)
26 END PROGRAM f_appel_c_sub_io

```

Partie langage C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void c_read_float(float *pf);
5 void c_write_float(float f);
6
7 void c_read_float(float *pf){
8     /* *pf est saisi en C */
9     printf("saisie d'un float en C\n");
10    scanf("%g", pf);
11    printf("float tel que saisi en C %g\n", *pf);
12    return;
13 }
14
15 void c_write_float(float f){
16    printf("valeur affichée en C %g\n", f);
17    return;
18 }

```

12.4.3 Exemple : appel de sous-programmes fortran depuis le C

L'exemple suivant montre l'appel en C de sous-programmes fortran pour saisir des réels et les afficher. Ils sont vus depuis le C comme des fonctions sans valeur de retour dont le nom est donné par le paramètre NAME dans le BIND (*cf.* lignes 6 et 13). Le sous-programme d'affichage `write_float` peut se contenter d'un passage par copie de valeur, d'où l'attribut `VALUE` en fortran (*cf.* ligne 14), car il ne modifie pas la valeur du paramètre. En revanche, la fonction de lecture, `f_read_float`, nécessite un passage par pointeur en C, directement compatible avec le passage d'argument utilisé par le fortran.

Partie langage C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void f_read_float(float *px);
5 void f_write_float(float x);
6
7 int main(void){
8     float f;
9     /* f est saisi en fortran */
10    f_read_float(&f);
11    printf("valeur du float affichée en C %g\n", f);
12    f_write_float(f);
13    exit(EXIT_SUCCESS);
14 }

```

Partie fortran 2003

```

1 MODULE m_io
2     USE, INTRINSIC :: iso_c_binding, ONLY:c_float
3     IMPLICIT NONE
4     CONTAINS
5         ! void f_read_float(float *px);
6         SUBROUTINE read_float(x) BIND(c, name="f_read_float") !
7             REAL(kind=c_float), INTENT(out) :: x
8             WRITE (*, *) "saisie en fortran: entrer un float"

```



```

9      READ (*,*) x
10     WRITE (*,*) "float lu en fortran avant passage en C", x
11  END SUBROUTINE read_float
12  ! void f_write_float(float x);
13  SUBROUTINE write_float(x) BIND(c, name="f_write_float") !
14     REAL(kind=c_float), INTENT(in), value :: x ! passage par copie
15     WRITE (*,*) "affichage en fortran après passage en C "
16     WRITE (*,*) x
17  END SUBROUTINE write_float
18 END MODULE m_io

```

12.5 Inter-opérabilité des tableaux

Les tableaux ne peuvent être inter-opérables que si leurs éléments sont de type et de paramètres inter-opérables. De plus, ils doivent soit être de profil explicite, soit ne comporter qu'une dimension indéterminée :

- la dernière spécifiée par * en fortran ;
- la première spécifiée par [] en C.

Dans le cas d'un tableau de rang 1, si la taille est explicite, elle doit être précisée et identique dans les deux langages. Si elle est indéterminée en fortran, elle ne doit pas être précisée en C.

L'inter-opérabilité des tableaux de rang supérieur à 1 s'appuie sur un ordre de rangement en mémoire qui veut qu'en fortran, ce soit le premier indice qui défile le plus vite⁶ alors qu'en C, où il s'agit alors de tableaux de tableaux, c'est le dernier indice qui défile le plus vite. Il faudra donc systématiquement échanger l'ordre des tailles pour passer d'un langage à l'autre.

Par exemple le tableau fortran déclaré :

```
INTEGER(kind=c_int) :: tab(3,5,6)
```

pourra être inter-opérable avec le tableau C :

```
int tab[6][5][3];
```

S'il reste une dimension indéterminée, ce sera la dernière en fortran, `tab(3,5,*)` et la première en C `tab[][5][3]`.

12.5.1 Exemple : fonctions C manipulant des tableaux définis en fortran

Les tableaux sont ici de rang deux et de taille fixe pour le compilateur dans le programme principal en fortran afin d'éviter l'allocation dynamique. Mais les fonctions en C supposent seulement qu'une étendue est connue : celle dont la connaissance permet de calculer les décalages à effectuer dans les adresses mémoire pour accéder aux éléments du tableau, c'est à dire celle de droite en C, notée `C_N2` (cf. ligne 8, partie C), qui correspond à celle de gauche en fortran, notée `F_LIG` (cf. ligne 12, partie fortran). Cette dimension commune, fixée ici grâce au préprocesseur, devrait l'être sous un seul nom dans un fichier unique inclus par le source C et par le source fortran. Avec les conventions de nommage classiques des compilateurs (cf. annexe E), il est nécessaire de nommer le source fortran avec comme suffixe `.F90` au lieu de `.f90` pour le voir traité par le pré-processeur.

```

----- Partie fortran 2003 -----
1  ! toutes les étendues du tableau transmis sont fixes
2  ! sauf la plus à droite en fortran (qui varie le plus lentement)
3  ! donc déclarées dans l'interface, sauf la dernière *
4  ! attention: stockage inversé en C
5
6  ! instruction pré-processeur à répéter en C
7  ! ou mieux : inclure un fichier commun dans les deux codes
8
9  ! pour imposer le traitement par le pré-processeur
10 ! choisir un fichier de suffixe .F90 au lieu de .f90 pour le source

```

6. Les éléments qui ne diffèrent que d'une unité de cet indice sont supposés être contigus en mémoire.

```

11
12 #define F_LIG 3 !
13
14 MODULE m_interf
15   USE, INTRINSIC :: iso_c_binding, ONLY:c_int
16   IMPLICIT NONE
17   INTERFACE
18     SUBROUTINE c_print_tab2d(mat, f_col) BIND(c)
19       IMPORT ::c_int !
20       INTEGER(kind=c_int), INTENT(in) :: mat(F_LIG, *)
21       INTEGER(kind=c_int), value :: f_col
22     END SUBROUTINE c_print_tab2d
23     SUBROUTINE c_mult_tab2d(mat, f_col) BIND(c)
24       IMPORT ::c_int !
25       INTEGER(kind=c_int), INTENT(inout) :: mat(F_LIG, *)
26       INTEGER(kind=c_int), value :: f_col
27     END SUBROUTINE c_mult_tab2d
28   END INTERFACE
29 END MODULE m_interf
30
31 PROGRAM f_appel_sub_c_tab2d
32   USE m_interf
33   IMPLICIT NONE
34   ! f_col est connu du compilateur en fortran
35   ! mais est un paramètre pour la fonction définie en C
36   INTEGER(kind=c_int), PARAMETER :: f_col = 4, f_col2 = 2
37   INTEGER(kind=c_int) :: mat(F_LIG, f_col), mat2(F_LIG, f_col2)
38   INTEGER :: i, j
39   ! remplissage des tableaux
40   DO i = 1, F_LIG
41     DO j = 1, f_col
42       mat(i,j) = 10 * i + j
43     END DO
44   END DO
45   mat2(:, :) = - mat(:, 1:f_col2)
46   ! affichage en fortran
47   WRITE (*,*) "tableau 2d affiché en fortran: 1er indice = ligne"
48   DO i = 1, F_LIG
49     WRITE(*,*) mat(i,:)
50   END DO
51   WRITE (*,*) "2ème tableau 2d affiché en fortran: 1er indice = ligne"
52   DO i = 1, F_LIG
53     WRITE(*,*) mat2(i,:)
54   END DO
55   ! appel pour l'affichage en C
56   CALL c_print_tab2d(mat, f_col)
57   WRITE(*,*) "deuxième tableau"
58   CALL c_print_tab2d(mat2, f_col2)
59   ! on multiplie chaque ligne par son numéro
60   WRITE (*,*) "tableau 2d multiplié par l'indice de ligne fortran"
61   CALL c_mult_tab2d(mat, f_col)
62   ! appel pour l'affichage en C
63   CALL c_print_tab2d(mat, f_col)
64   END PROGRAM f_appel_sub_c_tab2d

```

Partie langage C

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* doit être cohérent avec le 1er indice du tableau fortran */
5  /* C_N2 = F_LIG ce qui pourrait être assuré par un fichier */
6  /* commun inclus dans les deux sources fortran et C */
7
8  #define C_N2 3
9
10 /* transposition des dimensions entre fortran et C */
11 /* en C toutes les dimensions sauf la première doivent être connues */
12 /* du compilateur => fixées ici par le pré-processeur */
13 /* en fortran mat(F_LIG, *) = en C mat[][F_LIG] */
14 /* et passe la dimension variable en argument */
15 /* mat(F_LIG, f_col) = mat[c_n1][C_N2] avec c_n1=f_col C_N2=f_LIG */
16
17 /* integer(kind=c_int), intent(inout) :: mat(F_LIG, *) */
18 void c_print_tab2d(int mat[][C_N2], const int c_n1);
19 void c_mult_tab2d(int mat[][C_N2], const int c_n1);
20
21
22 void c_print_tab2d(int mat[][C_N2], const int c_n1){
23     int i, j;
24     printf("affichage en C\n");
25     for (i=0; i<C_N2; i++){ /* 2e indice C = 1er fortran */
26         for (j=0; j<c_n1; j++){ /* 1er indice C = 2e fortran */
27             printf("%d ", mat[j][i]);
28         }
29         printf("\n");
30     }
31     return;
32 }
33
34 void c_mult_tab2d(int mat[][C_N2], const int c_n1){
35     int i, j;
36     for (i=0; i<C_N2; i++){ /* 2e indice C = 1er fortran */
37         for (j=0; j<c_n1; j++){ /* 1er indice C = 2e fortran */
38             mat[j][i] *= (i+1); /* mult par 1er indice fortran */
39         }
40     }
41     return;
42 }

```

12.5.2 Exemple : fortran manipulant des tableaux dynamiques alloués en C

On peut utiliser les fonctions avancées du fortran portant sur les tableaux à partir du langage C, y compris dans le cas de tableaux dynamiques alloués et désalloués en C. Dans le cas de tableaux de rang 1, on définit en C une structure *vecteur* (lignes 11 à 14) comportant un pointeur C sur le début du tableau (de float ici) et un entier spécifiant le nombre de ses éléments. On définit un type dérivé *vecteur* inter-opérable en fortran (module *m_vecteur*) constitué des mêmes composantes en utilisant les types inter-opérables *c_ptr* et *c_int* associés en fortran.

Une fois l'argument *vecteur* passé au fortran par copie⁷ (ligne 8), il s'agit de lui associer un

7. Avec le passage par valeur, on s'interdit de modifier le pointeur C et la taille du tableau, donc de libérer ou

tableau de rang 1 de la taille précisée et dont les éléments sont ceux pointés par le pointeur C. Les tableaux allouables ou les tableaux à attribut pointeur du fortran sont plus riches que les pointeurs du C, car ils comportent les informations sur le profil du tableau.

On utilise donc le sous-programme `c_f_pointer` (ligne 16) pour effectuer le transfert des informations de la structure C vers le pointeur fortran. À ce propos, noter que la conversion du paramètre de taille de `INTEGER(kind=c_int)` en entier par défaut (ligne 13) est nécessaire pour spécifier le profil du tableau de façon portable en fortran. Une fois le tableau connu du fortran, on peut lui appliquer toutes les transformations possibles avec les fonctions-tableau du fortran, tant que l'on ne modifie pas sa taille.

Définition du type dérivé en fortran 2003

```

1 MODULE m_vect
2 USE, INTRINSIC :: iso_c_binding, ONLY : c_int, c_ptr
3 IMPLICIT NONE
4 ! définition du type dérivé vecteur tel que vu en C
5 ! ses composantes sont des types fortran inter-opérables C
6 TYPE, bind(c) :: vecteur
7   INTEGER(c_int) :: n ! taille du tableau
8   TYPE(c_ptr) :: ptc_v ! pointeur C
9 END TYPE vecteur
10 END MODULE m_vect

```

Sous-programme fortran 2003

```

1 ! les vecteurs sont alloués dynamiquement et libérés par l'appelant (en C)
2 MODULE m_modif_vect
3 USE m_vect
4 IMPLICIT NONE
5 CONTAINS
6   SUBROUTINE f_modif_vect(v1) bind(c, name="f_modif_vect")
7     USE, INTRINSIC :: iso_c_binding, ONLY : c_f_pointer, c_float, c_null_ptr
8     TYPE(vecteur), INTENT(in), value :: v1 ! passage de la structure par copie
9     REAL(c_float), DIMENSION(:), POINTER :: ptf_v ! pointeur fortran
10    ! affectation du pointeur de tableau fortran
11    ! avec l'adresse au sens pointeur C et le profil
12    INTEGER, DIMENSION(1) :: loc_v1_n ! vecteur profil fortran du tableau
13    loc_v1_n(1) = v1%n ! conversion du type integer(c_int)
14    ! vers le type entier par défaut (32 ou 64 bits)
15    ! pour spécifier le profil du tableau en fortran
16    CALL c_f_pointer(cptr=v1%ptc_v, fptr=ptf_v, shape=loc_v1_n)
17    WRITE(*,*) "on double les valeurs du tableau"
18    ptf_v(:) = 2* ptf_v(:)
19    WRITE(*,*) ptf_v(:)
20    WRITE(*,*) "décalage circulaire de 2 cases dans le tableau"
21    ptf_v(:) = CSIFT(ptf_v(:), shift=-2)
22    WRITE(*,*) "résultat en fortran"
23    WRITE(*,*) ptf_v(:)
24  END SUBROUTINE f_modif_vect
25 END MODULE m_modif_vect

```

Programme principal en langage C

```

1 /* version avec passage du pointeur vers a structure
2    => permet de désallouer et réallouer en fortran */
3 /* version avec allocation dynamique et type dérivé */

```

de réallouer le tableau en fortran. Bien sûr les valeurs des éléments du tableau sont modifiables en fortran.

```
4  /* attention compiler avec gcc -lm pour inclure libm.a */
5  /*          et inclure l'entête associé math.h */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9
10 /* type dérivé vecteur */
11 typedef struct {
12     int n ;
13     float *v;
14 } vecteur;
15
16 void f_modif_vect(vecteur v) ; /* déclaration de la procédure en fortran */
17 /* passage d'un pointeur vers la structure C => elle est modifiable */
18 /* attention : aucune vérification avec l'interface fortran bien sûr ! */
19
20 int main(void) {
21     vecteur v1 = {0, NULL};
22     int i;
23
24     /* Lecture du nombre de dimensions */
25     while(
26     printf("Entrez le nombre de composantes : (0 pour terminer) \n"),
27     scanf("%d", &(v1.n)),
28     v1.n > 0){
29
30     /* Allocation mémoire en C */
31     v1.v = (float *) calloc((size_t) v1.n, sizeof (float));
32     /*          size_t: conversion nécessaire en 64 bits */
33
34     /* Affectation des composantes */
35     for (i=0; i<v1.n; i++) {
36         v1.v[i] = (float) (i+1) ;
37     }
38
39     /* Affichage des composantes */
40     printf("affichage des %d composantes\n", v1.n);
41     for (i=0; i<v1.n; i++) {
42         printf("%f ", v1.v[i]);
43     }
44     printf("\n");
45
46     /* appel de la subroutine fortran */
47     printf("appel fortran \n"),
48     f_modif_vect(v1); /* pointeur vers la structure de vecteur*/
49
50     /* Impression du résultat */
51     printf("affichage des composantes après appel fortran\n");
52     for (i=0; i<v1.n; i++) {
53         printf("%f ", v1.v[i]);
54     }
55     printf("\n");
56
57     /* Libération de la memoire dans le même langage */
58     /* on suppose que libération <=> v1.n =0 */
59     if (v1.n > 0 || v1.v != NULL ) {
```

```
60     free(v1.v);
61     v1.n = 0;
62     v1.v = NULL ;
63     printf("mémoire libérée en C\n");
64 }
65 }
66 exit (EXIT_SUCCESS);
67 }
```

Si on passe la structure `vecteur` par pointeur en C, le pointeur du tableau et le nombre de composantes seront modifiables : il est ainsi possible de libérer le tableau et de le réallouer avec éventuellement une taille différente. Cette opération peut être déclenchée par le fortran, mais en appelant une fonction C à laquelle on repasse la structure `vecteur`. En effet, plus globalement, un espace mémoire alloué dans un langage doit être désalloué dans ce même langage.

Annexe A

Procédures intrinsèques

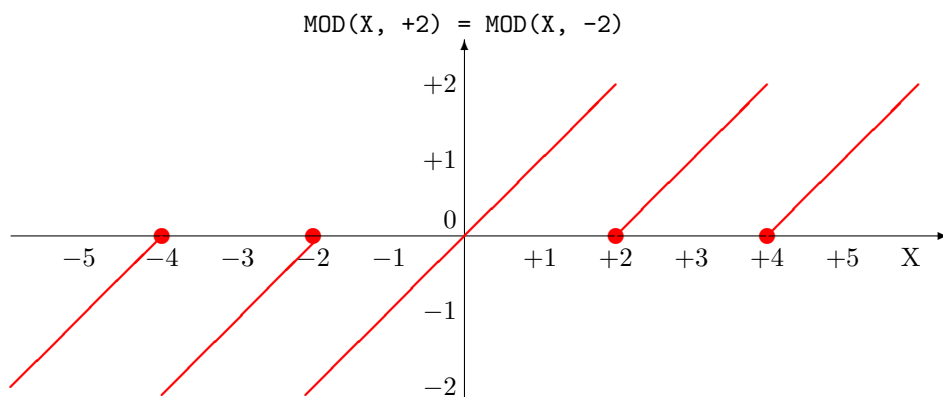
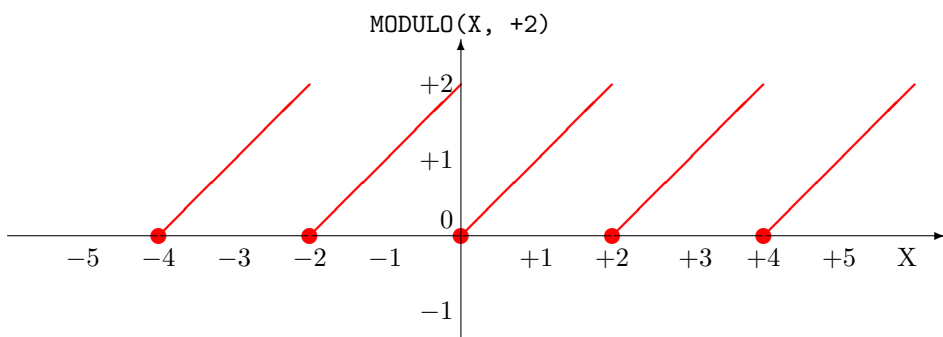
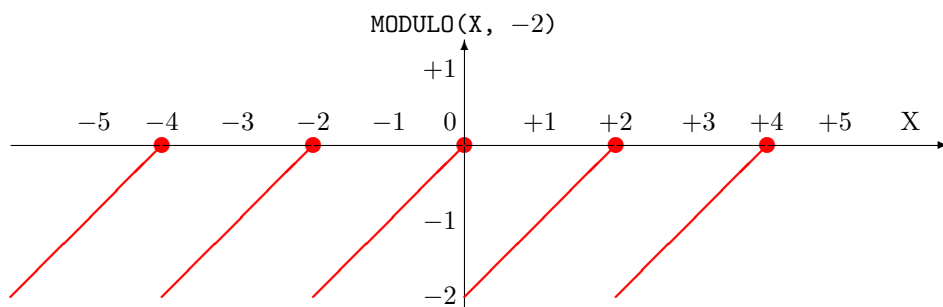
Les procédures intrinsèques sont des procédures intégrées au langage, en ce sens que leur interface est connue (aucune déclaration explicite via `USE` par exemple n'est nécessaire) et que leur code objet est inclus dans la bibliothèque du fortran (aucune autre bibliothèque n'est requise lors de l'édition de liens). La plupart des procédures intrinsèques font partie du standard du fortran 90, d'autres ont été ajoutées en fortran 95 puis en fortran 2003, et en fortran 2008 (les fonctions inverses de trigonométrie hyperbolique, les fonctions erreur et gamma ainsi que certaines fonctions de Bessel, voir [Site gfortran, de la collection de compilateurs gcc](#) par exemple). Des options de compilation (cf. [E.5.1](#), p. 153 pour `g95`, et [E.6.1](#), p. 154 pour `gfortran`) permettent d'autoriser l'emploi des fonctions intrinsèques hors standard 2003.

Cette annexe rassemble les principales procédures intrinsèques standard du fortran 90, 2003 et 2008. Les fonctions *élémentaires* (`elemental`) (cf. [6.5.6](#), p. 69) sont précédées du signe `*`.

A.1 Fonctions numériques de base

<code>*ABS(a)</code>	valeur absolue ou module (pour un complexe) de a	
<code>*ACOS(a)</code>	$\arccos(a)$ en radians	
<code>*ACOSH(a)</code>	$\operatorname{argch}(a)$	f2008
<code>*ASIN(a)</code>	$\arcsin(a)$ en radians	
<code>*ASINH(a)</code>	$\operatorname{argsh}(a)$	f2008
<code>*ATAN(a)</code>	$\arctan(a)$ en radians dans l'intervalle $[-\pi/2, \pi/2]$	
<code>*ATAN2(a, b)</code>	argument du nombre complexe $a + ib \neq 0$, en radians dans l'intervalle $[-\pi, \pi]$	
<code>*ATANH(a)</code>	$\operatorname{argth}(a)$	f2008
<code>*CONJG(a)</code>	\bar{a} complexe conjugué	
<code>*COS(a)</code>	$\cos(a)$	
<code>*COSH(a)</code>	$\cosh(a)$	
<code>*ERF(a)</code>	$\operatorname{erf}(a)$ fonction d'erreur $\frac{2}{\sqrt{\pi}} \int_0^a \exp(-t^2) dt$	f2008
<code>*ERFC(a)</code>	$\operatorname{erfc}(a)$ fonction d'erreur complémentaire $\frac{2}{\sqrt{\pi}} \int_a^\infty \exp(-t^2) dt$	f2008
<code>*EXP(a)</code>	$\exp(a)$	
<code>*GAMMA(a)</code>	$\Gamma(a)$ fonction gamma $\int_0^\infty t^{a-1} \exp -t dt$ $\Gamma(n+1) = n!$ si n entier positif	f2008
<code>*HYPOT(a, b)</code>	$\sqrt{a^2 + b^2}$ sans dépassement de capacité évitable ¹	f2008
<code>*LOG(a)</code>	$\ln(a)$ logarithme népérien	
<code>*LOG10(a)</code>	$\log(a)$ logarithme décimal	
<code>*LOG_GAMMA(a)</code>	$\ln(\Gamma(a))$ logarithme de la valeur absolue de la fonction gamma	f2008
<code>*MAX(a1, a2, ...)</code>	valeur maximum	
<code>*MIN(a1, a2, ...)</code>	valeur minimum	

*MOD(n, p)	reste de la division entière de n par p par troncature vers 0, soit $n - \text{int}(n/p)*p$, fonction impaire de n et paire de p (cf. figure A.1, p. 126) ²
*MODULO(n, p)	reste de la division entière de n par p , soit $n - \text{floor}(n/p)*p$, fonction de période $ p $ en n et son signe est celui de p (cf. figures A.2 et A.3, p. 126)
*SIGN(a, b)	$ a \times \text{sign}(b)$
*SIN(a)	$\sin(a)$
*SINH(a)	$\sinh(a)$
*SQRT(a)	\sqrt{a}
*TAN(a)	$\tan(a)$
*TANH(a)	$\tanh(a)$

FIGURE A.1 – Graphique de la fonction MOD(x, p) pour $p = \pm 2$ FIGURE A.2 – Graphique de la fonction MODULO(x, p) pour $p > 0$, par exemple $p = +2$ FIGURE A.3 – Graphique de la fonction MODULO(x, p) pour $p < 0$, par exemple $p = -2$

2. Pour éviter un dépassement avec le calcul du carré, on factorise le plus grand : $\sqrt{a^2 + b^2} = |a| \sqrt{1 + (b/a)^2}$.
 2. MOD est l'équivalent de l'opérateur % du C dans la norme C99, qui spécifie que la division entière se fait par troncature vers 0.

A.2 Conversion et troncature

*AIMAG(a)	partie imaginaire d'un complexe
*AINT(a[,kind])	valeur entière par troncature d'un réel, convertie en réel, c'est-à-dire $\text{REAL}(\text{INT}(a))$
*ANINT(a[,kind])	valeur entière par arrondi d'un réel, convertie en réel, c'est-à-dire $\text{REAL}(\text{NINT}(a))$
*CEILING(a[,kind])	conversion en type entier par excès (cf. figure A.4, p.127)
*CPLX(x,y[,kind])	conversion en type complexe
*FLOOR(a[,kind])	conversion en type entier par défaut (cf. figure A.5, p. 127)
*INT(a[,kind])	conversion en type entier par troncature (cf. figure A.6, p. 127)
*LOGICAL(a[,kind])	conversion entre variantes de booléens
*NINT(a[,kind])	conversion en type entier par arrondi (cf. figure A.7, p. 127)
*REAL(a[,kind])	conversion en type réel ou partie réelle d'un complexe

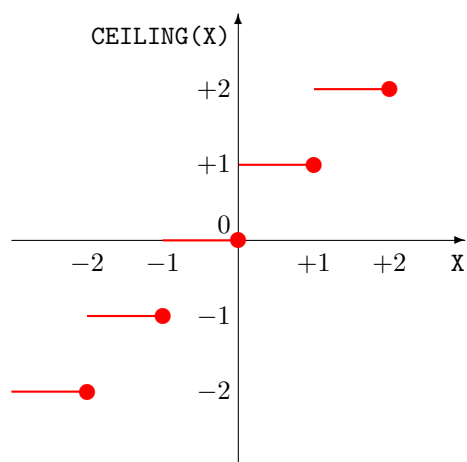


FIGURE A.4 – Graphique de la fonction CEILING

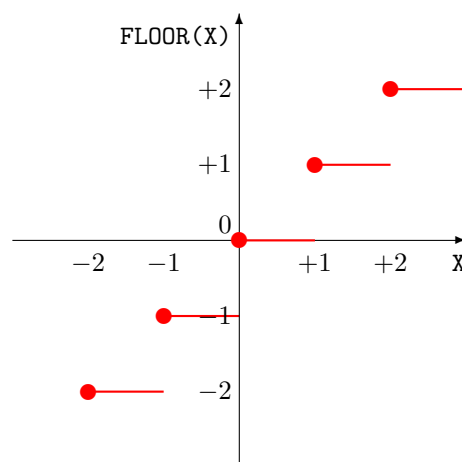


FIGURE A.5 – Graphique de la fonction FLOOR

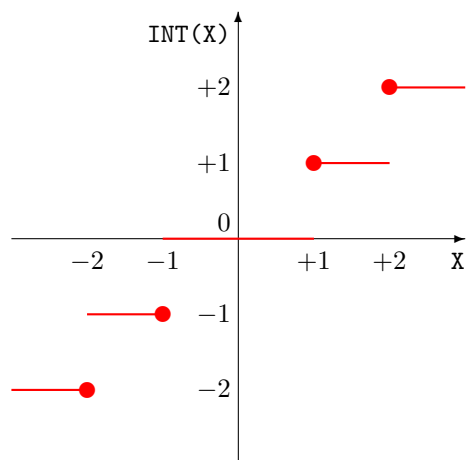


FIGURE A.6 – Graphique de la fonction INT

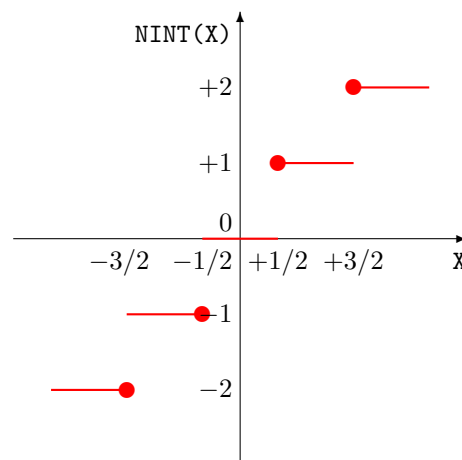


FIGURE A.7 – Graphique de la fonction NINT

A.3 Générateur pseudo-aléatoire

Les générateurs pseudo-aléatoires fournissent des tirages successifs statistiquement indépendants pris dans une suite *finie* de très longue période qui dépend du processeur et du compilateur,

mais dont l'interface d'appel `RANDOM_NUMBER` est, elle, portable. Le générateur possède une mémoire, ou tableau d'état, qui « progresse » naturellement à chaque invocation, mais qui peut aussi être manipulée grâce au sous-programme `RANDOM_SEED`. En mémorisant l'état du générateur à un instant donné, et en lui imposant³ plus tard la valeur mémorisée, on peut forcer le générateur à reprendre les tirages à partir d'un état déterminé et à reproduire ainsi plusieurs fois la même séquence.

`RANDOM_NUMBER(a)` générateur de nombres pseudo-aléatoires distribués uniformément sur $[0, 1[$; `a` est un scalaire ou un tableaux de réels.

`RANDOM_SEED(SIZE|PUT|GET)` initialisation du générateur de nombres pseudo-aléatoires¹, quatre usages possibles en fonction du mot-clef choisi :

(`C`) initialisation à des valeurs dépendant du processeur

```
call RANDOM_SEED()
```

(`SIZE`) lecture de la dimension du tableau d'état `INTENT(out)`

```
INTEGER :: n
call RANDOM_SEED(SIZE = n)
PRINT *, ' taille ', n
```

(`GET`) lecture des valeurs du tableau d'état `INTENT(out)`

```
INTEGER :: n
INTEGER, DIMENSION(n) :: tab
call RANDOM_SEED(GET = tab(1:n))
PRINT *, ' tableau d''état ', tab
```

(`PUT`) réinitialisation du tableau d'état `INTENT(in)`

```
INTEGER :: n
INTEGER, DIMENSION(n) :: tab
tab(1:n) = ... ! au moins une valeur non nulle
call RANDOM_SEED(PUT = tab(1:n))
```

Avec le compilateur `xlf` d'IBM, un autre mot-clef, `GENERATOR` est disponible, qui permet de choisir entre le générateur de base (`GENERATOR=1`, valeur par défaut) ou un générateur optimisé (`GENERATOR=2`), plus rapide et de période plus grande.

1. La taille du tableau d'état du générateur aléatoire n'est pas spécifiée par le standard. Elle dépend du compilateur utilisé. Sous LINUX et processeur INTEL ou AMD 32 bits, elle est de 1 avec le compilateur `f95` de NAG, de 2 avec le traducteur `f90` (de fortran 90 vers fortran 77) Vast (Pacific Sierra) et le compilateur `ifort` d'INTEL, de 4 avec `g95` et de 34 avec `pgf90` de Portland. Avec un processeur AMD 64 bits, cette taille passe à 2 avec `g95`, qui utilise des entiers par défaut sur 64 bits, mais cela correspond au même nombre d'octets.

Avec le compilateur de Portland sous SOLARIS de SUN, elle est de 4. Sur la machine NEC SX5 de l'IDRIS, elle est de 128. Sous DEC-OSF1 et processeur ALPHA de 64 bits, elle est de 2, comme sous AIX avec le compilateur `xlf` d'IBM.

Exemple d'usages du générateur aléatoire

```
PROGRAM random_test
IMPLICIT NONE
REAL                :: x
REAL, DIMENSION(3,4) :: alea
5 INTEGER :: n_alea ! taille de la mémoire du générateur aléatoire de fortran90
INTEGER, DIMENSION(:), ALLOCATABLE :: mem_alea, mem_alea2 ! tableaux d'état
```

3. Dès que l'on force le tableau d'état, on risque de perdre l'indépendance entre des tirages successifs.

```

INTEGER :: i, erreur_alloc
! tirage d'un nombre pseudo-aléatoire entre 0 et 1
CALL RANDOM_NUMBER(x)
10 WRITE (*, *) ' valeur tirée: x = ',x
! tirage d'un tableau de nombres pseudo-aléatoires indépendants entre 0 et 1
CALL RANDOM_NUMBER(alea)
WRITE (*, *) ' tableau des valeurs tirées:'
DO i = 1, SIZE(alea, 1)
15   WRITE (*, *) alea(i, :)
END DO
! recherche de la taille du tableau d'état
CALL RANDOM_SEED(SIZE = n_alea)
WRITE(*, *) ' taille de la mémoire du générateur aléatoire : ', n_alea
20 ! réservation de mémoire pour deux tableaux d'état
ALLOCATE(mem_alea(n_alea), stat = erreur_alloc)
IF(erreur_alloc /= 0 ) STOP
ALLOCATE(mem_alea2(n_alea), stat = erreur_alloc)
IF(erreur_alloc /= 0 ) STOP
25 ! lecture et mémorisation du tableau d'état courant avant la boucle
CALL RANDOM_SEED(GET = mem_alea)
WRITE(*, *) '- tableau d''état du générateur avant la boucle:', mem_alea(:)
DO i = 1 , 3
! lecture et affichage du tableau d'état courant puis tirage
30   CALL RANDOM_SEED(GET = mem_alea2)
   WRITE(*, *) ' tableau d''état du générateur :', mem_alea2(:)
   CALL RANDOM_NUMBER(x)
   WRITE (*, *) ' valeur tirée: x = ',x
END DO
35 ! réinitialisation avec le tableau d'état mémorisé avant la boucle
! pour repartir du même état et donc obtenir la même série
CALL RANDOM_SEED(PUT = mem_alea)
WRITE (*, *) '- puis réinitialisation à l''état avant la boucle '
DO i = 1 , 3
40   ! lecture et affichage du tableau d'état courant puis tirage
   CALL RANDOM_SEED(GET = mem_alea2)
   WRITE(*, *) ' tableau d''état du générateur :', mem_alea2(:)
   CALL RANDOM_NUMBER(x)
   WRITE (*, *) ' valeur tirée: x = ',x
45 END DO
END PROGRAM random_test

```

A.4 Représentation des nombres

DIGITS(a)	nombre de digits (bits en base 2) utilisés pour représenter la mantisse a dans le type de a
EPSILON(a)	plus petite valeur dont la somme avec 1 diffère de 1 dans le type de a
*EXPONENT(a)	exposant (entier) de a dans la représentation de a sous la forme $\text{FRACTION}(a) * \text{REAL}(\text{RADIX}(a))^{*\text{EXPONENT}(a)}$
*FRACTION(a)	partie fractionnaire de a dans la représentation de a sous la forme $\text{FRACTION}(a) * \text{REAL}(\text{RADIX}(a))^{*\text{EXPONENT}(a)}$
HUGE(a)	plus grande valeur représentable dans le type de a
KIND(a)	type numérique dans lequel est représenté a
MAXEXPONENT(a)	valeur maximale de l'exposant (entier) de $\text{RADIX}(a)$ dans le type de a
MINEXPONENT(a)	valeur minimale de l'exposant (entier) de $\text{RADIX}(a)$ dans le type de a
*NEAREST(a,s)	nombre le plus proche de a <i>exactement</i> représentable dans le type de a (par excès ou défaut suivant le signe de s)

PRECISION(<i>a</i>)	nombre de chiffres décimaux significatifs dans le type de <i>a</i>
RADIX(<i>a</i>)	base de la représentation des nombres dans le type de <i>a</i> (en général 2)
RANGE(<i>a</i>)	domaine de <i>a</i> , exprimé en puissance entière de 10, soit $\text{INT}(\text{MIN}(\text{LOG}_{10}(\text{HUGE}(\mathbf{a})) , -\text{LOG}_{10}(\text{TINY}(\mathbf{a}))))$
*RRSPACING(<i>a</i>)	inverse de la distance relative entre deux nombres de même type que <i>a</i> autour de <i>a</i>
*SCALE(<i>a</i> , <i>i</i>)	$\mathbf{a} * \text{REAL}(\text{RADIX}(\mathbf{a}))^{**i}$
SELECTED_INT_KIND(<i>r</i>)	sous-type entier permettant de représenter les entiers de l'intervalle $] - 10^r, 10^r[$
SELECTED_REAL_KIND(<i>p</i> , <i>r</i>)	sous-type réel permettant de représenter les réels <i>x</i> tels que $10^{-r} < x < 10^r$ avec <i>p</i> chiffres significatifs
*SET_EXPONENT(<i>a</i>)	$\text{FRACTION}(\mathbf{a}) * \text{REAL}(\text{RADIX}(\mathbf{a}))^{**i}$
*SPACING(<i>a</i>)	plus petite distance entre <i>a</i> et la valeur de même type la plus proche
TINY(<i>a</i>)	plus petite valeur absolue représentable dans le type de <i>a</i>
BIT_SIZE(<i>i</i>)	nombre de bits des <i>entiers</i> du même type que <i>i</i>

A.5 Fonctions opérant sur des tableaux

ALL(array[,dim])	teste si tous les éléments d'un tableau booléen sont vrais
ALLOCATED(array)	teste si un tableau est alloué (résultat booléen)
ANY(array[,dim])	teste si au moins un des éléments d'un tableau booléen est vrai
COUNT(array[,dim])	compte les éléments vrais d'un tableau booléen
DOT_PRODUCT(vector_a, vector_b)	produit scalaire de deux vecteurs
LBOUND(array[,dim])	vecteur des bornes inférieures des indices d'un tableau (scalaire si la dimension concernée est précisée)
MATMUL(matrix_a, matrix_b)	multiplication matricielle
MAXLOC(array[,mask])	tableau de rang 1 donnant la position de la première occurrence de la valeur maximum d'un tableau
MAXVAL(array[,dim][,mask])	valeur maximum d'un tableau
MERGE(tsource, fsource, mask)	fusion des tableaux <i>tsource</i> (si vrai) et <i>fsource</i> (si faux) selon le masque <i>mask</i>
MINLOC(array[,mask])	tableau de rang 1 donnant la position de la première occurrence de la valeur minimum d'un tableau
MINVAL(array[,dim][,mask])	valeur minimum d'un tableau
PACK(array, mask [,vector])	range dans un tableau de rang 1 les éléments de <i>array</i> sélectionnés par le masque <i>mask</i> (conformant avec <i>tab</i>). Si <i>vector</i> est fourni, le tableau résultat est si nécessaire complété par les éléments terminaux de <i>vector</i> pour obtenir un vecteur de même étendue. <i>vector</i> doit donc posséder au moins autant d'éléments que le masque en sélectionne.
PRODUCT(array[,dim][,mask])	produit des valeurs d'un tableau
RESHAPE(source, shape[,pad][,order])	restructuration du tableau <i>source</i> selon le profil <i>shape</i>
SHAPE(array)	profil d'un tableau
SIZE(array[,dim])	taille (nombre d'éléments) d'un tableau [suivant la dimension <i>dim</i>]
SPREAD(source, dim, ncopies)	création d'un tableau de dimension supérieure par duplication (<i>ncopies</i> fois) du scalaire ou tableau <i>source</i> selon la dimension <i>dim</i>

SUM(array[,dim][,mask])	somme des valeurs d'un tableau
TRANPOSE(matrix)	transposition
UBOUND(array[,dim])	vecteur des bornes supérieures des indices d'un tableau (scalaire si la dimension concernée est précisée)
UNPACK(vector, mask, field)	déploie les éléments du tableau <code>vector</code> de rang 1 dans un tableau initialement rempli avec <code>field</code> selon le masque <code>mask</code> conformant à <code>field</code>

A.6 Manipulation de bits

Les fonctions intrinsèques suivantes permettent la manipulation directe de bits sur les entiers :

*BTEST(i, pos)	vrai si le bit <code>pos</code> de l'entier <code>i</code> vaut 1
*IAND(i, j)	vrai si tous les bits de <code>i</code> et <code>j</code> sont égaux
*IBCLR(i, pos)	rend un entier du type de <code>i</code> dont le bit <code>pos</code> a été mis à zéro.
*IBITS(i, pos, len)	rend un entier du type de <code>i</code> dont les <code>len</code> bits en commençant au bit <code>pos</code> sont ceux de <code>i</code> complétés par des zéros à gauche.
*IBSET(i, pos)	rend un entier du type de <code>i</code> dont le bit <code>pos</code> a été mis à un.
*IEOR(i, j)	rend un entier du type de <code>i</code> et <code>j</code> dont les bits sont obtenus par ou exclusif entre ceux de <code>i</code> et <code>j</code> .
*IOR(i, j)	rend un entier du type de <code>i</code> et <code>j</code> dont les bits sont obtenus par ou inclusif entre ceux de <code>i</code> et <code>j</code> .
*ISHFT(i, shift)	rend un entier du type de <code>i</code> dont les bits sont obtenus par décalage de <code>shift</code> vers la gauche (la droite si <code>shift</code> négatif) le remplissage étant assuré par des zéros.
*ISHFTC(i, shift)	rend un entier du type de <code>i</code> dont les bits sont obtenus par décalage de <code>shift</code> vers la gauche (la droite si <code>shift</code> négatif) le remplissage étant assuré par recyclage circulaire.
*NOT(i)	rend un entier du type de <code>i</code> dont les bits sont obtenus par inversion de ceux de <code>i</code> .
TRANSFER(source, mold [, size])	interprète la représentation binaire de <code>source</code> selon le type spécifié par <code>mold</code>

A.7 Pointeurs (cf. 11.1.1, p. 105)

ASSOCIATED(pointer [, target])	rend une valeur booléenne vraie si le pointeur <code>pointer</code> est associé à une cible (à la cible <code>target</code> si l'argument optionnel est fourni).
NULL([mold])	rend un pointeur désassocié du type du pointeur <code>mold</code> si cet argument est présent.

f2003 A.8 Accès à l'environnement système

GET_ENVIRONMENT_VARIABLE(name [,value] [,length] [,status] [,trim_name])	sous-programme qui récupère la valeur de la variable d'environnement <code>name</code> ⁴ dans la chaîne de caractères <code>value</code> , sa longueur dans l'entier <code>length</code> , éventuellement avec les blancs à gauche si <code>trim_name</code> vaut <code>.false</code> . Le statut de retour <code>status</code> vaut 1 si la variable demandée n'existe pas, -1 si la valeur est trop longue pour la chaîne <code>value</code> et 0 en l'absence d'erreur ou d'avertissement.
GET_COMMAND([command] [,length] [,status])	sous-programme qui rend dans la chaîne <code>command</code> la commande ayant lancé le programme et éventuellement le nombre de caractères de cette chaîne dans <code>length</code> . L'argument optionnel <code>status</code> vaut 0 quand l'appel s'est déroulé sans problème, -1 quand la chaîne passée est trop courte pour héberger la ligne de commande, et un nombre positif en cas d'erreur.

4. C'est l'équivalent de la fonction `getenv` du langage C.

COMMAND_ARGUMENT_COUNT()

fonction qui rend un entier contenant le nombre de paramètres⁵ de la commande ayant lancé le programme.

GET_COMMAND_ARGUMENT(number [,value] [,length] [,status])

sous-programme qui rend dans la chaîne *value* l'argument numéro *number*⁶ de la commande ayant lancé le programme et éventuellement sa taille dans *length*.

Exemple d'accès à l'environnement système

On peut ainsi passer des paramètres sous forme de chaînes de caractères au programme principal, de la même façon qu'en C. L'exécution de la commande `a.out test 12 "5 6"` où `a.out` est obtenu en compilant le programme suivant :

```

1 PROGRAM t_environ
2 IMPLICIT NONE
3 CHARACTER(len=20) :: nom, valeur, arg0, arg
4 CHARACTER(len=80) :: commande
5 INTEGER :: longueur, statut, n_param, i
6 WRITE(*,*) "récupération d'une variable d'environnement"
7 nom(1:) = "SHELL"
8 CALL get_environnement_variable(nom, valeur, longueur, statut)
9 IF (statut /= 0) STOP "erreur get_environnement_variable"
10 WRITE(*,*) nom(1:5), "=", valeur(1:longueur)
11 CALL get_command(COMMAND=commande, LENGTH=longueur, STATUS=statut)
12 IF (statut /= 0) STOP "erreur get_command"
13 WRITE(*,*) "longueur de la ligne de commande", longueur
14 WRITE(*,*) "ligne de commande = |", commande(1:longueur), "|"
15 CALL get_command_argument(0, arg0, longueur, statut)
16 IF (statut /= 0) STOP "erreur get_command_argument"
17 WRITE(*,*) "commande ayant lancé ce programme=", arg0(1:longueur)
18 n_param = command_argument_count()
19 WRITE(*,*) "nb de paramètres de la commande = ", n_param
20 DO i=1, n_param
21     CALL get_command_argument(i, arg, longueur, statut)
22     IF (statut /= 0) STOP "erreur get_command_argument"
23     WRITE(*,*) "paramètre ", i, " de la commande=", arg(1:longueur)
24 END DO
25 END PROGRAM t_environ

```

affiche :

```

1 récupération d'une variable d'environnement
2 SHELL=/bin/bash
3 longueur de la ligne de commande          17
4 ligne de commande = |a.out test 12 5 6|
5 commande ayant lancé ce programme=a.out
6 nb de paramètres de la commande =        3
7 paramètre          1 de la commande=test
8 paramètre          2 de la commande=12
9 paramètre          3 de la commande=5 6

```

f95/2003 A.9 Gestion du temps

DATE_AND_TIME([DATE,] [TIME,] [ZONE,] [VALUES])

sous-programme permettant d'accéder à la date et à l'heure fournis par le système d'exploitation. Les quatre arguments sont des arguments de sortie optionnels :

5. C'est à dire `argc -1` du langage C.

6. C'est le paramètre donné par `argv[number]` du langage C.

- DATE : chaîne de 8 caractères recevant la date sous forme `aaaammjj`
- TIME : chaîne de 10 caractères recevant le temps sous forme `hhmss.iii` où `iii` représentent les millisecondes
- ZONE : chaîne de 5 caractères recevant le décalage par rapport au temps universel sous la forme `+/-hhmm`
- VALUE : tableau de 8 entiers représentant successivement l'année, le mois, le jour, le décalage par rapport au temps universel exprimé en minutes, l'heure, les minutes, les secondes et les millisecondes.

Exemple d'utilisation de DATE_AND_TIME

```

1  MODULE util_time
2  IMPLICIT NONE
3  CONTAINS
4  FUNCTION deltat(td, tf)
5  ! calcul de la duree en secondes
6  ! attention : incorrect si changement de date
7  INTEGER, DIMENSION(8), INTENT(in) :: td, tf
8  REAL :: deltat
9  deltat = (tf(8) -td(8))/1000. + (tf(7) -td(7)) + &
10         60 * ( (tf(6) -td(6)) + 60 *(tf(5) -td(5)))
11 END FUNCTION deltat
12 SUBROUTINE affiche(t)
13 ! affichage de la date suivie du temps
14 ! en heures, minutes, secondes et millisecondes
15 INTEGER, DIMENSION(8), INTENT(in) :: t
16 WRITE(*,'(i2.2,a,i2.2,a,i4.4, a, i2.2,a,i2.2,a,i2.2,a,i3.3, a)') &
17     t(3),"/",t(2),"/",t(1), " ", &
18     t(5), "h", t(6), "'", t(7), "'", t(8), "ms "
19 END SUBROUTINE
20 END MODULE util_time
21
22 PROGRAM datetime
23 USE util_time
24 IMPLICIT NONE
25 ! test du sous-programme date_and_time
26 INTEGER, DIMENSION(8) :: debut, fin
27 CHARACTER(len=8) :: jour
28 CHARACTER(len=10) :: instant
29 INTEGER:: i, j
30 CALL DATE_AND_TIME(date=jour, time=instant)
31 WRITE(*,*) "date (AAAAMMJJ) : ", jour
32 WRITE(*,*) "heure (hhmss.*): ", instant
33 CALL DATE_AND_TIME(values=debut)
34 WRITE(*,'(a)', advance="no") "debut "
35 CALL affiche(debut)
36 DO i=1, 1000000 ! ajuster < huge(i)
37     j = 1 - i
38 END DO
39 CALL DATE_AND_TIME(values=fin)
40 WRITE(*,'(a)', advance="no") "fin  "
41 CALL affiche(fin)
42 WRITE(*,*) "duree de la boucle", deltat(debut, fin), 's'
43 END PROGRAM datetime

```

L'exécution de ce code donne par exemple :

```

1  date (AAAAMMJJ) : 20110824
2  heure (hhmmss.*): 135703.472
3  debut 24/08/2011 13h57'03"472ms
4  fin   24/08/2011 13h57'03"482ms
5  duree de la boucle 0.01 s

```

CPU_TIME(time)

sous-programme permettant d'accéder au temps du processeur exprimé en secondes avec une résolution de la microseconde : l'origine est arbitraire, seules les différences sont pertinentes. Ce sous-programme permet notamment d'évaluer des durées d'exécution de code. Le paramètre est un argument de sortie réel.

SYSTEM_CLOCK([count] [,count_rate] [,count_max])

sous-programme permettant d'accéder au temps exprimé en nombre de tops d'horloge de fréquence COUNT_RATE modulo une valeur maximale COUNT_MAX. Les trois arguments sont des arguments de sortie entiers optionnels.

A.10 Caractères et chaînes de caractères

*ACHAR(i)	i ^e caractère (où i est compris entre 0 et 127) dans l'ordre de l'ascii
*CHAR(i [, kind])	i ^e caractère dans l'ordre du compilateur (qui inclut souvent l'ordre ascii, mais peut aussi fournir des caractères nationaux, comme nos caractères accentués)
*IACHAR(c)	position du caractère (entre 0 et 127) c dans l'ordre ascii
*ICHAR(c)	position du caractère c dans l'ordre du compilateur (qui inclut souvent l'ordre ascii, mais peut aussi fournir des caractères nationaux, comme nos caractères accentués)
NEW_LINE(c)	caractère de nouvelle ligne dans la variante de caractères de c
SELECTED_CHAR_KIND(string)	rend le numéro de la variante de caractères dans laquelle la chaîne string est donnée ('DEFAULT', 'ASCII' ou 'ISO_10646')
<hr/>	
*ADJUSTL(string)	justification à gauche de la chaîne string
*ADJUSTR(string)	justification à droite de la chaîne string
LEN(string)	longueur de la chaîne string
*LEN_TRIM(string)	longueur de la chaîne string sans les blancs à droite
REPEAT(string,ncopies)	duplique ncopies fois la chaîne string
TRIM(string)	suppression des blancs à droite
<hr/>	
*LGE(string_a,string_b)	vrai si string_a est placé après string_b dans l'ordre lexicographique de l'ascii ou coïncide avec string_b
*LGT(string_a,string_b)	vrai si string_a est placé strictement après string_b dans l'ordre lexicographique de l'ascii
*LLE(string_a,string_b)	vrai si string_a est placé avant string_b dans l'ordre lexicographique de l'ascii ou coïncide avec string_b
*LLT(string_a,string_b)	vrai si string_a est placé strictement avant string_b dans l'ordre lexicographique de l'ascii

*INDEX(string,substring)	position de la sous-chaîne <code>substring</code> dans la chaîne <code>string</code>
*SCAN(string,set)	repère un des caractères de la chaîne <code>set</code> dans la chaîne <code>string</code>
*VERIFY(string,set)	vérifie que la chaîne <code>string</code> ne contient que des caractères de la chaîne <code>set</code> (renvoie l'entier 0), sinon donne la position du premier caractère de <code>string</code> qui ne fait pas partie de l'ensemble <code>set</code>

Codes ascii et iso-latin1

Les fonctions réciproques `achar` et `iachar` traduisent le codage ASCII des caractères explicité dans le tableau ci-après, qui indique le numéro en octal, décimal et hexadécimal de chaque caractère, soit sa représentation s'il est imprimable soit son abréviation et éventuellement la séquence d'échappement (commençant par une contre-oblique) qui le représente dans le langage C.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL \0	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL \a	107	71	47	G
010	8	08	BS \b	110	72	48	H
011	9	09	HT \t	111	73	49	I
012	10	0A	LF \n	112	74	4A	J
013	11	0B	VT \v	113	75	4B	K
014	12	0C	FF \f	114	76	4C	L
015	13	0D	CR \r	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ \
035	29	1D	GS	135	93	5D]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	&
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g

050	40	28	(150	104	68	h
051	41	29)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

Plusieurs codages des caractères sur 8 bits permettent d'étendre le code ASCII en représentant aussi les caractères dotés de signes diacritiques (accents, cédille, ...) présents dans les langages européens. Le tableau suivant indique les caractères du codage ISO-8859-1 (alphabet latin 1, pour les langues de l'Europe occidentale) qui sont imprimables et ne figurent pas dans le codage ASCII.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD	–	SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	ˉ	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPIT TWO
263	179	B3	³	SUPERSCRIPIT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPIT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK

274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF
276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA
310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ð	LATIN CAPITAL LETTER ETH
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN
370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE

371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

Annexe B

Ordre des instructions

Ce diagramme indique l'ordre à respecter entre les différentes instructions dans une unité de programme en fortran. Les instructions présentées dans des colonnes différentes à l'intérieur d'une même ligne du tableau peuvent être insérées dans un ordre quelconque.

PROGRAM, FUNCTION, SUBROUTINE, MODULE, BLOCK DATA		
USE		
IMPORT		
FORMAT, ENTRY	IMPLICIT NONE	
	PARAMETER	IMPLICIT
	PARAMETER et DATA	définition de types, blocs d'interfaces, déclarations
	DATA	instructions exécutables
CONTAINS		
procédures internes ou de module		
END		

Noter, en particulier, que l'on place dans l'ordre, d'abord USE, puis IMPLICIT, ensuite les déclarations de types ou de variables, puis les instructions exécutables. Enfin, l'instruction CONTAINS qui introduit les procédures internes se place juste avant la fin de l'unité de programme qui l'héberge.

En pratique, les instructions de formats sont soit placées juste après les ordres d'entrées-sortie qui les appellent, soit groupées en fin d'unité de programme, surtout dans le cas où elles sont utilisées par plusieurs ordres d'entrées-sortie. ⇐ ♥

Concernant les déclarations de variables dans les procédures, une pratique saine consiste à déclarer d'abord les arguments de la procédure, en précisant leur vocation (INTENT), puis les variables locales. ⇐ ♥

Annexe C

La norme IEEE 754 de représentation des flottants

C.1 Introduction

C.1.1 Représentation des réels en virgules flottante

Représenter un réel (non nul) en virgule flottante dans une base b consiste à en donner une approximation finie sous la forme :

$$r = s b^{e'} m' = s b^{e'} \sum_{i=0}^{q'} p_i b^{-i} \quad (\text{C.1})$$

où

- $s = \pm 1$ est le signe ;
- e' est un entier qualifié d'*exposant* ;
- m' est le *significande*, appelé abusivement la *mantisse* que l'on peut décomposer sur la base b selon les $q' + 1$ poids entiers p_i tels que $0 \leq p_i < b$

Plusieurs décompositions mantisse-exposant étant possibles (de la même façon qu'en décimal, on peut écrire 1.2×10^0 ou 0.12×10^1), on choisit ici l'exposant pour que $1 \leq m' < b$ (qui correspondrait à l'intervalle $[1., 10.[$ si on travaillait en décimal). Cette représentation diffère donc de celle introduite plus haut (cf. 2.2, p. 10), avec une mantisse m entre $1/b$ et 1 (qui correspondrait à l'intervalle $].1, 1.[$ si on travaillait en décimal), choix qui était plus en accord avec les fonctions intrinsèques FRACTION et EXPONENT. Ces deux représentations sont liées par $e' = e - 1$ et $q' = q - 1$.

Dans le cas de la base 2 et en prenant en compte le choix d'un premier bit p_0 de m' à 1, l'expression (C.1) prend la forme :

$$r = (-1)^s 2^{e'} \left(1 + \sum_{i=1}^{q-1} \frac{p_i}{2^i} \right) = (-1)^s 2^{e'} (1 + f) = (-1)^s 2^e \left(\frac{1}{2} + \sum_{i=1}^{q-1} \frac{p_i}{2^{i+1}} \right) \quad (\text{C.2})$$

où $f = m' - 1$ est la partie fractionnaire du significande sur $q' = q - 1$ bits.

Chaque variante du type réel (déterminée via le paramètre de codage KIND) est caractérisée par :

- un nombre M de bits dédiés au stockage de la mantisse qui fixe la précision relative des réels ;
- un nombre E de bits dédiés au stockage de l'exposant qui fixe le domaine de valeurs couvert.

signe	exposant	mantisse
1 bit	E bits	M bits

C.1.2 Arithmétique étendue

Mais la norme IEEE-754-2008 permet aussi de représenter des résultats d'opérations arithmétiques qui ne s'évaluent pas comme des réels, tels que :

- ◇ $1./0.$ soit $+\infty$ affiché **+Inf**
- ◇ $-1./0.$ soit $-\infty$ affiché **-Inf**
- ◇ $0./0.$ qui est indéterminé, ou `sqrt(-1.)` qui est une opération invalide : il est codé comme **NaN** soit « Not-A-Number ». **NaN** possède aussi une variante négative. Enfin, les résultats $\pm\text{NaN}$ peuvent donner lieu à des messages (**signaling NaN**) ou être propagés sans avertissement dans les calculs (**quiet NaN**) : un bit particulier permet de déterminer ce comportement.

L'arithmétique traditionnelle s'étend à ces valeurs spéciales de façon naturelle si on précise que dès qu'un **NaN** intervient dans une expression, le résultat est aussi représenté par un **NaN**. Par exemple, **Inf + Inf** donne **+Inf**, mais **Inf - Inf** donne **NaN**. Enfin, le codage IEEE permet de représenter un « zéro positif » et un « zéro négatif » qui interviennent naturellement dans des opérations arithmétiques étendues telles que $1./+\text{Inf}$ ou $1./-\text{Inf}$.

C.2 Les codes normaux

L'exposant e' codé sur E bits qui détermine l'intervalle couvert définissant ainsi le domaine, est de signe *a priori* quelconque entre $-2^{E-1} + 1$ et 2^{E-1} : on lui ajoute $2^{E-1} - 1$ de façon à stocker le nombre sans signe $e'' = e' + 2^{E-1} - 1$, appelé *exposant biaisé*, qui est un entier compris entre 0 et $2^E - 1$. Mais on exclut les deux valeurs extrêmes de l'exposant pour réserver ces *codes spéciaux* (où les bits de l'exposant e'' sont tous à 0 ou tous à 1) aux deux zéros signés et à l'arithmétique étendue de la norme IEEE. Ainsi, pour les *codes normaux*, $1 \leq e'' \leq 2^E - 2$, donc $-2^{E-1} + 2 \leq e' \leq 2^{E-1} - 1$.

C.2.1 Les nombres normalisés

En binaire, pour les nombres *normalisés*, le bit de plus fort poids de la mantisse m' vaut donc 1. On convient alors de ne pas le stocker pour gagner un bit sur la précision de la représentation ; on stocke donc la partie fractionnaire f de la mantisse sur $M = q' = q - 1$ bits seulement. Mais le nombre de bits de la mantisse donné par la fonction intrinsèque **DIGITS** est $M + 1 = q$ car il prend en compte ce bit caché.

Ainsi pour les flottants normalisés, $r = (-1)^s 2^{e'' - 2^{E-1} + 1} \times 1.f$ où f est la partie fractionnaire de la mantisse. Dans chaque octave $[2^n, 2^{n+1}[$, il y a 2^M nombres représentés exactement, en progression arithmétique de raison 2^{n-M} ; cette raison double à chaque changement d'octave. Si le successeur de 2^n est $2^n(1 + 2^{-M})$, son prédécesseur est $2^n(1 - 2^{-M-1})$.

Valeurs extrêmes

Le plus grand réel normalisé positif ainsi représentable, donné par la fonction **HUGE**, est donc obtenu pour $e'' = 2^E - 2$, soit $e' = 2^{E-1} - 1$ avec $p_0 = 1$ et $p_i = 1$ pour $i \geq 0$, soit $m' = \sum_{i=0}^{M-1} 2^{-i} = 2(1 - 2^{-M+1})$. Il vaut donc $2^{2^{E-1} - 1} (1 - 2^{-M-1})$. Si un calcul donne une valeur absolue supérieure à **HUGE(x)**, il y a théoriquement dépassement de capacité par valeur supérieure (**floating overflow**).

Le plus petit réel positif normalisé ainsi représentable, donné par la fonction **TINY**, est donc obtenu pour $e'' = 1$, soit $e' = -2^{E-1} + 2$ avec $p_0 = 1$ et $p_i = 0$ pour $i \geq 1$, soit $f = 0$. Il vaut donc $2^{-2^{E-1} + 2}$. Si un calcul donne une valeur absolue non nulle mais inférieure à **TINY(x)**, il y a théoriquement dépassement de capacité par valeur inférieure (**floating underflow**).

Le nombre d'octaves de réels positifs est $e''_{\max} - e''_{\min} + 1$, soit $2^E - 2$.

C.2.2 Les nombres dénormalisés

Mais, quitte à perdre en précision, on peut aussi représenter des nombres de valeur absolue inférieure à la plus petite valeur ayant un exposant décalé e'' nul : on travaille alors en virgule fixe. Si l'exposant décalé e'' est nul, les M bits suivants représentent exactement la mantisse sans bit caché à 1 : on définit ainsi les nombres *dénormalisés*. Le plus petit nombre dénormalisé positif (tous les bits à 0 sauf le dernier) vaut donc $2^{-2^{E-1} + 2 - M}$.

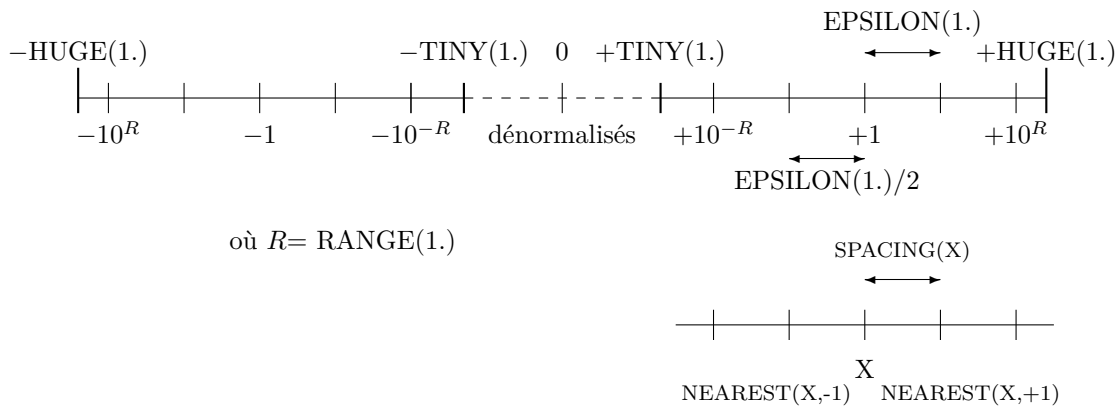


FIGURE C.1 – Représentation des réels. Même si l'échelle s'apparente à une échelle log, celle-ci n'est pas respectée afin de figurer le zéro. De plus, successeur et prédécesseur de X sont en général à égale distance de X (progression arithmétique de raison $\text{SPACING}(X)$ dans chaque octave) sauf si X est à la limite d'une octave, c'est à dire une puissance entière de 2 (dans ce cas le pas est deux fois plus faible du côté de zéro).

C.3 Les codes spéciaux

Les codes employés pour les valeurs spéciales de l'arithmétique étendue sont caractérisés par le fait que tous leurs bits d'exposant sont à 1. On note aussi les deux zéros signés avec tous les bits d'exposant à 0.

	signe	exposant	mantisse
+0	0	E bits à 0	M bits à zéro
-0	1	E bits à 0	M bits à zéro
quiet NaN > 0	0	E bits à 1	1 suivi de (M-1) bits quelconques
signaling NaN > 0	0	E bits à 1	0 suivi d'au moins 1 bit à 1
quiet NaN < 0	1	E bits à 1	1 suivi de (M-1) bits quelconques
signaling NaN < 0	1	E bits à 1	0 suivi d'au moins 1 bit à 1
+Inf	0	E bits à 1	M bits à 0
-Inf	1	E bits à 1	M bits à 0

C.3.1 Arithmétique IEEE et options de compilation

Il est cependant nécessaire de choisir les options de compilation adéquates pour imposer le respect de la norme IEEE (voir par exemple E.3.1 p. 151 pour pgf95) dans la propagation des drapeaux indiquant la sortie de l'arithmétique classique. On peut en effet constater une grande diversité de comportement par défaut entre plusieurs compilateurs sur le simple cas d'un calcul dont le résultat devrait être indéterminé.

```
PROGRAM nan_indeterm
IMPLICIT NONE
REAL :: zero, infini, indet
zero = 0.
infini = 1./zero
indet = infini * zero ! en principe indéterminé
WRITE(*,*) " 0. ,          1./0. ,          (1./0.)*0. "
WRITE(*,*) zero, infini, indet
END PROGRAM nan_indeterm
```


Pour des flottants sur 32 bits, la plus petite valeur positive en normalisé, c'est-à-dire le résultat de la fonction TINY vaut 2^{-126} soit approximativement $1.175494\text{E-}38$. La plus grande valeur, c'est-à-dire le résultat de la fonction HUGE est obtenue avec les 23 bits de la mantisse à 1, $2^{+127} \times (2 - 2^{-23}) \approx 2^{128}$, soit approximativement $3.402823\text{E}+38$. Quant à la précision relative, elle est caractérisée par la fonction EPSILON, l'écart relatif *maximal* entre deux flottants successifs, soit $2^{-23} \approx 1.192093\text{E-}7$. Dans chaque octave, de 2^n à 2^{n+1} par exemple, il y a 2^{23} , soit environ 8 millions de réels en progression arithmétique de raison 2^{n-23} . Pour conserver approximativement la précision relative, le pas double quand on progresse d'une octave en s'éloignant de zéro. Noter que le voisin gauche de 1., soit NEAREST(1.,-1.) est deux fois plus proche de 1. que son voisin de droite, soit NEAREST(1.,1.). Enfin, le nombre d'octaves¹ de réels positifs normalisés est de $2^8 - 2 = 254$.

f2008 ⇒ Pour vérifier le codage binaire, on peut utiliser le format binaire **b** (cf. 5.4.1, p. 42) des réels : en écriture, il permet de convertir un réel décimal en binaire et en lecture de convertir du binaire en réel affiché en décimal.

```

program bin_real
! format B avec des réels = norme 2008
! acceptée par gfortran et ifort, pas par g95
implicit none
real :: r0 = 1. ! réel par défaut (32 bits)
character(len=32) :: binreal
write(*, *) "mantisse sur",digits(r0),"bits (1 caché)"
! écriture d'un réel en binaire
write(*, *) r0
write(*, "(a)") "seeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee"
write(*, "(b32.32)") r0
! lecture d'un réel saisi en binaire
! "seeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee"
binreal="00111111100000000000000000000000" ! 1.
read(binreal, "(b32.32)") r0
write(*, "(es16.9)") r0
end program bin_real

```

Par exemple pour des réels sur 32 bits, le programme `bin_real` ci-contre affiche :

```

mantisse sur 24 bits (1 caché)
1.
seeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
0011111110000000000000000000000000
1.000000000E+00

```

C.4.2 Le codage IEEE des flottants sur 64 bits

Les réels en double précision sont généralement stockés sur 64 bits avec 11 bits d'exposant et 52 bits plus un bit à 1 non stocké pour la mantisse. Le décalage de l'exposant est de 1023.

signe	exposant biaisé	partie fractionnaire de la mantisse
s	e''	$f = m' - 1$
1 bit	11 bits	52 bits

$$r = (-1)^s 2^{e''-1023} \times 1.f \text{ où } f = m' - 1 \quad (\text{C.4})$$

Avec 64 bits, le domaine s'étend : TINY vaut $2^{-2^{10}+2} = 2^{-1022} \approx 2.2250738585072014 \times 10^{-308}$ alors que HUGE vaut $2^{2^{10}-1}(2 - 2^{-52}) = 2^{1024}(1 - 2^{-53}) \approx 1.7976931348623157 \times 10^{+308}$. Mais la précision est aussi améliorée avec $\text{EPSILON}=2^{-52} \approx 2.220446049250313 \times 10^{-16}$. Chaque octave comporte 2^{52} soit environ $4,510^{15}$ réels en progression arithmétique et il y a $2^{11} - 2$, soit 2046 octaves de réels positifs normalisés.

Pour plus d'informations sur la norme IEEE-754, on pourra consulter le site [WIKIPEDIA \(2010\)](#) et le chapitre 9 « Arithmétique des ordinateurs » de l'ouvrage de [STALLINGS \(2003\)](#). On pourra aussi s'exercer sur les convertisseurs hexadécimal-IEEE en ligne sur le site [VICKERY \(2009\)](#).

1. Le nombre total de réels normalisés non nuls sur 32 bits est donc $2 \times 2^{23}(2^8 - 2)$, soit $2^{32} - 2^{25}$. Si on y ajoute 2×2^{23} réels dénormalisés et 2 zéros, cela donne $2^{32} - 2^{24} + 2$ valeurs normales, parmi les 2^{32} codes disponibles. Au contraire, tous les codes sur 32 bits représentent des entiers valides, qui sont donc très légèrement plus nombreux que les réels sur 32 bits!

Annexe D

Correspondance entre la syntaxe du fortran 90 et celle du langage C

Avertissement

Les correspondances de syntaxe mises en évidence ici ne signifient absolument pas des fonctionnalités équivalentes dans les deux langages. Par exemple, les chaînes de caractères sont traitées de manière différente en fortran, où le type intrinsèque existe et en C, où elles sont représentées par des tableaux de caractères. Sans chercher l'exhaustivité dans l'analogie, elles ont simplement pour but d'aider le lecteur familier du langage C à éviter les erreurs de syntaxe en fortran.

D.1 Déclarations des types de base

fortran 90	C89	compléments C99
<code>integer :: var</code>	<code>int var;</code>	
<code>real :: var</code>	<code>float var;</code>	
<code>complex :: var</code>		<code>complex var ;</code>
<code>character :: var</code>	<code>char var;</code>	
<code>logical :: var</code>		<code>bool var ;</code>

D.2 Opérateurs algébriques

	fortran 90	langage C
addition	<code>+</code>	<code>+</code>
soustraction	<code>-</code>	<code>-</code>
multiplication	<code>*</code>	<code>*</code>
division	<code>/</code>	<code>/</code>
élévation à la puissance	<code>**</code>	<code>≈ pow(x,y)</code>
reste modulo j	<code>≈ mod(i,j) ou modulo(i,j)</code>	<code>%</code>

D.3 Opérateurs de comparaison

	fortran 90	C
inférieur à	<code><</code>	<code><</code>
inférieur ou égal à	<code><=</code>	<code><=</code>
égal à	<code>==</code>	<code>==</code>
supérieur ou égal à	<code>>=</code>	<code>>=</code>
supérieur à	<code>></code>	<code>></code>
différent de	<code>/=</code>	<code>!=</code>

D.4 Opérateurs logiques

	fortran 90	C	
		C89	C95
ET	.AND.	&& (évaluation minimale)	and
OU	.OR.	(évaluation minimale)	or
NON	.NOT.	!	not

D.5 Opérations sur les bits

Les arguments des fonctions (en fortran) et les opérandes des opérateurs (en C) agissant bit à bit sont des entiers.

fortran 90	signification	langage C
NOT(i)	négation bit à bit	~i
IAND(i, j)	et	i & j
IOR(i, j)	ou (inclusif)	i j
IEOR(i, j)	ou exclusif	i ^ j
ISHFT(i, j)	décalage à gauche de j bits	i << j
ISHFT(i, -j)	décalage à droite de j bits	i >> j

D.6 Structures de contrôle

fortran 90	C
if (expr. log.) then bloc else bloc endif	if (expr. log.) bloc else bloc
select case (expr.) case ('sélecteur') bloc case (default) bloc end select	switch (expr. entière) { case sélecteur : bloc break; default : bloc }
do entier = début, fin[, pas] bloc end do	for (expr ₁ ; expr ₂ ; expr ₃) bloc
do while (expr. log.) bloc end do	while (expr. log.) bloc
cycle	continue;
exit	break;
go to étiquette-numér.	goto étiquette;
return	return;
stop	exit;

D.7 Pointeurs

	fortran 90	C
déclarer des pointeurs	REAL, POINTER:: ptr, pts	float *ptr, *pts;
déclarer une cible	REAL, TARGET:: r	float r;
pointer vers	ptr => r	ptr = &r;
pointer vers	pts => ptr	pts = ptr;
cible pointée	ptr	*ptr
dissocier un pointeur	ptr => null()	ptr = NULL;
tester l'association	IF(ASSOCIATED(ptr))	if(ptr != NULL);
tester l'association	IF(ASSOCIATED(ptr, pts))	if(ptr == pts);
allouer un pointeur	ALLOCATE(ptr[...]) de plus, associe ptr	ptr=malloc(...); ptr=calloc(...);
désallouer un pointeur	DEALLOCATE(ptr) de plus, désassocie ptr	free(ptr); faire ptr = NULL;
conversion implicite	pt_real = pt_int	*pt_real = *pt_int
interdit/déconseillé	pt_real => entier	pt_real = &entier
interdit/déconseillé	pt_real => pt_entier	pt_real = pt_entier

D.8 Fonctions mathématiques

Les fonctions mathématiques intrinsèques du fortran sont utilisables sans faire appel à un quelconque `USE`, ni lier explicitement une bibliothèque. À l'inverse, pour utiliser les fonctions de la bibliothèque mathématique en langage C, il est nécessaire d'une part d'inclure le fichier d'entête `math.h` et d'autre part de lier la bibliothèque `libm.a`. De plus, ces fonctions sont génériques en fortran, alors qu'en C, chaque type utilise une fonction spécifique¹, d'où les variantes pour argument entier (i), double (r), voire complexe (c) en C99, et les sous-variantes pour un argument de type réel (double par défaut) suffixées `f` pour les float, et `l` pour les long double.

fortran	C89	C99	remarques
ABS(a)	abs(a) (i) fabs/f/l(a) (r)	cabs(a) (z)	abs pour les entiers en C
ACOS(a)	acos/f/l(a) (r)	cacos/f/l(a) (z)	norme C99
AIMAG(a)		cimag/f/l(a) (z)	
ASIN(a)	asin/f/l(a) (r)	casin/f/l(a) (z)	résultat flottant en C
ATAN(a)	atan/f/l(a) (r)	catan/f/l(a) (z)	
ATAN2(a,b)	atan2/f/l(a,b)		norme C99
CEILING(a)	ceil/f/l(a)		
CONJG(a)		conj/f/l(a) (z)	résultat flottant en C
COS(a)	cos/f/l(a) (r)	ccos/f/l(a) (z)	
COSH(a)	cosh/f/l(a) (r)	ccosh(a) (z)	résultat flottant en C
EXP(a)	exp/f/l(a) (r)	cexp/f/l(a) (z)	
FLOOR(a)	floor/f/l(a)		opérateur en C
LOG(a)	log/f/l(a) (r)	clog/f/l(a) (z)	
LOG10(a)	log10/f/l(a)		opérateur en C
MOD(a,p)	a%p si positifs	a%p	
MODULO(a,p)	a%p si positifs		opérateur en fortran
a**b	pow/f/l(a,b) (r)	cpow/f/l(a,b) (z)	
SIGN(a,b)		copysign/f/l(a,b)	opérateur en C
SIN(a)	sin/f/l(a) (r)	csin/f/l(a) (z)	
SINH(a)	sinh/f/l(a) (r)	csinh/f/l(a) (z)	opérateur en C
SQRT(a)	sqrt/f/l(a) (r)	csqrt(a) (z)	
TAN(a)	tan/f/l(a) (r)	ctan/f/l(a) (z)	opérateur en C
TANH(a)	tanh/f/l(a) (r)	ctanh/f/l(a) (z)	
NEAREST(a,s)		nextafter/f/l(a,s)	

1. En C99, il est cependant possible d'implémenter des fonctions mathématiques génériques pour les nombres réels flottants grâce au préprocesseur en incluant le fichier `tgmath.h`.

D.9 Formats

La correspondance entre les spécifications de format en fortran et en C reste très approximative. Les comportements des deux langages pour les entrées-sorties sont assez différents : en particulier en écriture, lorsque la taille du champ de sortie précisée dans la spécification de format s'avère insuffisante pour permettre l'écriture, le C prend la liberté d'étendre cette taille (quitte à ne pas respecter la taille prévue) alors que le fortran refuse de déborder et affiche des caractères * en lieu et place de la valeur à écrire.

D'autre part, le fortran donne la priorité à la liste d'entrées/sorties sur le nombre de descripteurs actifs du format, alors que le langage C cherche à satisfaire en priorité les descripteurs actifs. En particulier, si le nombre de descripteurs actifs est inférieur au nombre d'entités à coder ou décoder, le fortran ré-explore le format, alors que dans le cas contraire, il ignore les descripteurs **actifs** superflus. À l'inverse, en C, si le nombre de descripteurs actifs est inférieur au nombre d'éléments de la liste, seuls ceux qui correspondent à un descripteur sont codés ou décodés, alors que dans le cas contraire, les descripteurs actifs superflus provoquent des accès à des zones mémoires non réservées avec des conséquences imprévisibles mais souvent plus graves en entrée qu'en sortie.

	fortran 90	C
entier en décimal	In	%nd
	IO	%d
entier en octal	On	%nO
	OO	%O
entier en hexadécimal	Zn	%nX
	ZO	%X
flottant en virgule fixe	Fn.p	%n.pf
	FO.p	%f
flottant en mantisse et exponentielle général	En.p	%n.pe
	Gn.p	%n.pg
chaîne de caractères	An	%n.ps

D.10 Codage des valeurs numériques

D.10.1 Codage des entiers

fortran		langage C
Entiers sur 32 bits		
HUGE(1)	$2147483647 = 2^{31} - 1 \approx 2 \times 10^9$	INT32_MAX
Entiers sur 64 bits		
HUGE(1_8)	$9223372036854775807 = 2^{63} - 1 \approx 9 \times 10^{18}$	INT64_MAX

D.10.2 Codage des flottants

fortran		langage C
Flottants sur 32 bits		
HUGE(1.)	$3.402823 \times 10^{+38}$	FLT_MAX
TINY(1.)	1.175494×10^{-38}	FLT_MIN
EPSILON(1.)	1.192093×10^{-7}	FLT_EPSILON
Flottants sur 64 bits		
HUGE(1.DO)	$1.7976931348623157 \times 10^{+308}$	DBL_MAX
TINY(1.DO)	$2.2250738585072014 \times 10^{-308}$	DBL_MIN
EPSILON(1.DO)	$2.220446049250313 \times 10^{-16}$	DBL_EPSILON

Annexe E

Compilateurs et options de compilation

Certains compilateurs fortran sont disponibles sur plusieurs plates-formes comme `pgf95` de PORTLAND ou `nagfor` (ex-`f95`) de NAG (The Numerical Algorithms Group). D'autres sont plus spécifiques de certains systèmes propriétaires, comme `xlf` sur AIX d'IBM, ou `f90`, celui de HP issu de COMPAQ... lui-même issu de DIGITAL EQUIPMENT CORPORATION, dont la version pour LINUX sur processeur ALPHA est libre dans le cadre d'une licence particulière. Le compilateur `ifort` d'INTEL sous LINUX (<http://www.intel.com/cd/software/products/asm-na/eng/compilers/flin/index.htm>) est aussi utilisable sans frais pour un usage non commercial.

Enfin, deux compilateurs fortran libres (sur le modèle de `g77`) de la collection de compilateurs `gcc` ont été développés :

- `g95` (<http://www.g95.org>)
- et `gfortran` (<http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gfortran/>)

disponibles pour plusieurs plates-formes dont LINUX (sur processeurs 32 bits et 64 bits), WINDOWS et MACINTOSH OS X

E.1 Compilateur `xlf` sous `ibm-aix`

Le compilateur fortran des machines IBM fonctionnant sous AIX¹ ou sur processeur POWER-PC sous linux est `xlf`² (voir [Manuels Fortran IBM](#)). Pour compiler du fortran 90 au format libre, il faut utiliser `xlf90` ou `xlf -qfree=f90`. Pour respecter la convention de nommage `*.f90` des fichiers sources au format libre, on doit spécifier l'option `-qsuffix=f=f90` :

```
xlf90 -qsuffix=f=f90 <essai>.f90
```

ou

```
xlf -qfree=f90 -qsuffix=f=f90 <essai>.f90
```

E.1.1 Options du compilateur `xlf` conseillées

Il est conseillé de demander au compilateur de signaler les constructions obsolètes de fortran 90 par l'option `-qlanglvl=90pure` : ⇐ ♥

```
xlf90 -qlanglvl=90pure -qsuffix=f=f90 <essai>.f90
```

1. AIX est l'UNIX d'IBM.
2. En 2008, la version 11 de `xlf` est disponible.

De plus, au moins dans la phase de mise au point, on peut lui demander de vérifier le non-dépassement des bornes des tableaux par l'option `-qcheck` (version longue de l'option `-C`), ainsi que la compatibilité des arguments passés entre procédures par l'option `-qextchk`.

Enfin, il est prudent de provoquer une interruption de l'exécution dans les cas où les calculs en virgule flottante aboutissent à des résultats non garantis³ :

ENable : nécessaire pour permettre l'interruption à la suite d'un des problèmes suivants

INValid : opération invalide

OVerflow : dépassement de capacité vers le haut (quantité trop grande en valeur absolue pour être représentée sur la machine, car l'exposant serait trop grand et positif)

ZERODivide : division par zéro

Cette précaution est assurée en compilant avec l'option `-qfltrap`, suivie des sous-options nécessaires : `xlf90 -qfltrap=en:inv:ov:zero`

Enfin, l'option `-qinit=f90ptr` permet d'imposer aux pointeurs un état non associé à leur création, au lieu de l'état indéterminé (obtenu par défaut).

L'option `-F fichier.cfg` permet de spécifier un fichier de configuration personnel (inspiré de `/etc/xlf.cfg` qui représente la configuration par défaut) qui rassemble les options choisies.

E.2 Compilateur f95 ou nagfor de NAG

Le compilateur du NUMERICAL ALGORITHMS GROUP s'appelait `f95` mais comme le compilateur `gfortran` est aussi appelé `f95`, à partir de la version 5.2, il s'appelle maintenant `nagfor` (voir [Manuels Fortran NAG](#)).

Par défaut, `nagfor` considère les fichiers sources de suffixes `.f90`, `.f95` et leurs variantes majuscules comme écrits au format libre, alors que ceux de suffixes `.f`, `.for` `.ftn` et leurs variantes majuscules sont supposés au format fixe. Les options `-free` et `-fixed` permettent de spécifier le type de format indépendamment de ces conventions. Dans le cas du format fixe, on peut augmenter la longueur maximale des lignes de 72 à 132 caractères avec l'option `-132`.

Les fichiers sources de suffixe en majuscules sont par défaut traités par le préprocesseur `fpp`, alors que ceux dont le suffixe est en minuscules ne le sont pas par défaut.

L'option `-f2003` supprime les avertissements concernant les fonctionnalités du fortran 2003, habituellement signalées comme des extensions du fortran 95.

L'option `-kind=` permet de spécifier la numérotation des variantes de type pour les entiers, les booléens et les flottants :

`-kind=sequential` (par défaut) les numérote en séquence à partir de 1 par taille croissante ;

`-kind=byte` les numérote avec le nombre d'octets utilisés pour le stockage.

Un module `f90_kind.f90` fournit un ensemble de variantes de type (KIND) prédéfinis.

L'option `-nomod` empêche la production du fichier de module de suffixe `.mod`.

Avec la version 5.3 est apparue l'option `-encoding=` qui permet de préciser le codage du fichier source, parmi `ISO_Latin_1` (par défaut), `Shift_JIS` et `UTF_8`. Ne pas confondre avec l'option `encoding` de OPEN pour les entrées-sorties sur fichiers texte externes (*cf.* 5.3.2, p. 38) ni avec les variantes de type pour les variables de type chaîne de caractères (*cf.* 8.1.3, p. 85).

E.2.1 Options du compilateur nagfor conseillées

♡ ⇒ Les options suivantes sont conseillées pour forcer des vérifications complémentaires, au moins dans la phase de mise au point des programmes :

`-C=array` signale les dépassements de bornes des tableaux et des chaînes de caractères ;

`-C=calls` vérifie les références aux procédures ;

`-C=dangling` signale les pointeurs indéfinis ;

`-C=do` signale les boucles `do` dont le pas est nul ;

³. Les noms des sous-options peuvent être abrégés en ne conservant que la partie du mot-clef écrite ici en majuscules.

`-C=recursion` signale les récursivités invalides ;

`-nan` initialise les variables de type `REAL` ou `COMPLEX` à `NaN` (Not A Number), ce qui provoque un arrêt si elles sont utilisées avant d'être définies.

Cet ensemble d'options peut être sélectionné globalement avec la syntaxe `-C=all`, à laquelle il est prudent d'ajouter `-C=undefined`⁴ pour rechercher les variables non définies.

L'option `-float-store` interdit l'utilisation de registres de capacité supérieure à 64 bits pour les calculs flottants : en effet l'unité de calcul flottant (Floating Processor Unit) effectue bien souvent les calculs dans des registres de 80 voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Cette option ne doit être activée qu'à titre de test car elle dégrade les performances des programmes même si elle contribue à leur portabilité !

Enfin l'option `-gline` permet de retracer l'origine d'une erreur à l'exécution en indiquant les lignes de code⁵ où elle s'est produite. Comme cette option est consommatrice de temps et de place, on ne l'activera qu'en cas d'erreur à l'exécution et on relancera le programme ainsi recompilé pour analyser l'erreur.

E.3 Compilateur pgf95 de Portland

Le groupe Portland commercialise un ensemble de compilateurs C, C++ et fortran ainsi que des outils de développement et de calcul parallèle associés pour diverses plates-formes et systèmes d'exploitation. Sous LINUX, avec des processeurs AMD64 ou IA32/EM64T, on peut utiliser le compilateur pgf95 (voir [Manuels Fortran PGI](#)).

Par défaut, les fichiers sources de suffixes `.f90`, `.f95` et `.f03` sont considérés comme écrits au format libre, alors que ceux de suffixes `.f`, `.for` et `.ftn` sont supposés au format fixe. Les options `-Mfreeform` et `-Mnofreeform` permettent de spécifier le type de format indépendamment de ces conventions. Enfin, l'option `-Mnomain` spécifie lors de l'édition de lien que le programme principal n'est pas en fortran (*cf.* chapitre 12, page 115).

E.3.1 Options du compilateur pgf95 conseillées

Pour compiler des procédures récursives, il est nécessaire de préciser l'option `-Mrecursive`, car l'option par défaut est `-Mnorecursive`. Par défaut, les variables locales ne sont pas sauvegardées d'un appel à l'autre (option `-Mnosave`). Le stockage statique des variables locales peut être évité avec l'option `-Mrecursive`.

Les options suivantes sont conseillées pour signaler aider à écrire un code robuste, au moins dans la phase de mise au point des programmes : ⇐ ♥

`-Mdclchk` requiert que toutes les variables soient déclarées (l'option par défaut est `-Mnodclchk`) ;

`-Mstandard` demande au compilateur de signaler les écarts à la syntaxe du fortran standard selon la norme choisie ;

`-Minform=inform` choisit le niveau maximum d'information (erreurs, avertissements et informations)

`-Mbounds` signale les dépassements de bornes des tableaux et des chaînes de caractères.

De plus, l'option `-Mallocatable=95` ou `-Mallocatable=03` permet de choisir le comportement des affectations de variables allouables : allocation préalable nécessaire en fortran 95 ou (ré-)allocation implicite en fortran 2003.

Par ailleurs, l'option `-Mbyteswapio` permet d'invertir l'ordre des octets dans les entrées-sorties sur des fichiers binaires (conversion little endian vers big-endian et réciproquement). (*cf.* note 9, page 34)

Enfin l'option `-Ktrap=fp` force l'interruption en cas de problème d'opérations en flottant (division par zéro, opération invalide, dépassement de capacité) à l'exécution. On peut aussi activer l'option `-Kieee` pour respecter la norme IEEE 754 (*cf.* annexe C, p. 140) lors des calculs en flottant, au prix éventuel d'un moins bon temps de calcul.

4. Une procédure compilée avec l'option `-C=undefined` peut ne pas être compatible avec une procédure compilée sans cette option. De plus, cette option est incompatible avec les fonctions à résultat allouable.

5. En remontant dans la hiérarchie des appels de procédures.

E.4 Compilateur ifort d'Intel

Sous LINUX, avec des processeurs INTEL ou compatibles, on peut utiliser le compilateur `ifort` (voir [Manuels Fortran Intel](#)), disponible gratuitement sous conditions pour un usage non commercial.

Par défaut, les fichiers sources de suffixes⁶ `.f90` sont considérés comme écrits au format libre, alors que ceux de suffixes `.f`, `.for` et `.ftn` sont supposés au format fixe. Les options `-free` et `-fixed` permettent de spécifier le type de format indépendamment de ces conventions. Les options `-i8` et `-r8` permettent de promouvoir respectivement les variables entières par défaut en 64 bits et les variables réelles par défaut en double précision (64 bits). De plus l'option `-fpconstant` permet de promouvoir les constantes réelles par défaut en double précision si elles sont affectées à des variables double précision. Enfin, l'option `-nofor-main` spécifie lors de l'édition de lien que le programme principal n'est pas en fortran (*cf.* chapitre 12, page 115).

E.4.1 Options du compilateur ifort conseillées

Le stockage statique des variables locales peut être évité avec l'option `-automatic`, `-nosave` ou `-auto`. Pour compiler des procédures récursives, il est nécessaire de préciser l'option `-recursive` (qui implique `-automatic`), car l'option par défaut est `-norecursive`.

⇒ Les options suivantes sont conseillées pour signaler des erreurs à l'exécution, au moins dans la phase de mise au point des programmes :

- `-stand f90` ou `-stand f95` ou `-stand f03` demande au compilateur de signaler tous les écarts à la syntaxe du fortran standard selon la norme choisie ;
- `-warn declarations` ou `-implicitnone` signale les identifiants utilisés sans être déclarés (typage implicite) ;
- `-warn uncalled` signale les fonctions qui ne sont jamais appelées ;
- `-warn unused` signale les variables déclarées mais jamais utilisées ;
- `-warn all` demande d'émettre tous les avertissements que le compilateur peut diagnostiquer (cela implique notamment `-warn declarations`, `-warn uncalled`, `-warn unused`, ...)
- `-check bounds` ou `-CB` signale les dépassements de bornes des tableaux et des chaînes de caractères ;
- `-check format` détecte une inconsistance entre une valeur et son format de sortie ;
- `-check all` ou `-C` permet de spécifier globalement toutes les vérifications du type `-check`.

Enfin l'option `-ftrapuv` qui initialise les variables allouées sur la pile à des valeurs invalides peut permettre de détecter des défauts d'initialisation.

Des variables d'environnement permettent de spécifier le contexte d'exécution des programmes compilés avec `ifort`. Parmi celles-ci, on notera celle qui permet de désigner l'ordre des octets (*cf.* note 9, page 34) dans la représentation des données binaires : `F_UFMTENDIAN`. Elle permet d'indiquer (sous diverses formes) la liste des numéros d'unités logiques pour lesquels les entrées/sorties se feront avec conversion entre `little` et `big`, sachant que le format par défaut est `little` sur les processeurs où ce compilateur fonctionne.

E.5 Compilateur g95

Le compilateur `g95` (voir [VAUGHT, 2009](#)) est issu de `gcc`⁷, la collection de compilateurs du GNU, capable de compiler les langages C, C++, Objective-C, Fortran, Java, et Ada. Plus précisément, il s'appuie sur `gcc`, suit sa syntaxe et admet donc ses options générales, mais propose aussi des options spécifiques pour le langage fortran.

6. Contrairement à d'autres compilateurs, `ifort`, n'admet ni le suffixe `.f95` ni le suffixe `.f03` pour des sources au standard fortran 95 ou 2003.

7. En fait, le développement d'un compilateur fortran 90 au sein de `gcc` est en cours sous le projet `gfortran` (*cf.* E.6, p. 154). Mais un des anciens membres de l'équipe de `gfortran` a entrepris un développement en parallèle qui a abouti plus rapidement avec `g95`. Cependant `g95` semble maintenant figé depuis août 2010.

Par défaut, les fichiers sources de suffixes `.f90`, `.f95`, `.f03` et leurs variantes majuscules sont considérés comme écrits au format libre, alors que ceux de suffixes `.f`, `.for` et leurs variantes majuscules sont supposés au format fixe. Les options `-ffree-form` et `-ffixed-form` permettent de spécifier le type de format indépendamment de ces conventions. Dans le cas du format fixe, on peut préciser la longueur maximale des lignes avec les options `-ffixed-line-length-80` ou `-ffixed-line-length-132`.

Les fichiers sources de suffixe en majuscules sont par défaut traités par le préprocesseur du C, alors que ceux dont le suffixe est en minuscules ne le sont pas par défaut.

Avec l'option `-fmod=`, il est possible de fixer le répertoire où seront créés les fichiers `.mod` de module lors de la compilation. On pourra l'associer avec l'option `-I` qui permet d'ajouter un répertoire au chemin de recherche des fichiers de module.

L'option `-M` permet l'affichage des dépendances des fichiers objets et fichiers de module, en vue de la rédaction du fichier `makefile` :

`g95 -M <fichier.f90>` affiche les dépendances du fichier `<fichier.o>`.

Enfin, une liste de variables d'environnement (préfixées par `G95_`) permet d'adapter l'environnement d'exécution des programmes compilés par `g95`. L'avantage de tels réglages par rapport aux options de compilation est qu'ils peuvent être modifiés une fois l'exécutable créé et ajustés *a posteriori* lors de l'exploitation. On peut connaître les valeurs de ces variables d'environnement en lançant l'exécutable produit par `g95` avec l'option `--g95`. Ces variables concernent notamment les entrées/sorties, avec, entre autres (cf. 5.2.1, p. 33) :

- `G95_STDIN_UNIT` de type entier qui permet de choisir le numéro de l'unité logique pré-connectée à l'entrée standard ;
- `G95_STDOUT_UNIT` de type entier qui permet de choisir le numéro de l'unité logique pré-connectée à la sortie standard ;
- `G95_STDERR_UNIT` de type entier qui permet de choisir le numéro de l'unité logique pré-connectée à la sortie d'erreur standard ;
- `G95_COMMA` de type booléen qui permet de choisir la virgule comme séparateur décimal (cf. 5.3.2, p. 38) ;
- `G95_ENDIAN` de type chaîne de caractères et de valeurs possibles `BIG`, `LITTLE`, `SWAP` ou `NATIVE` (par défaut), qui permet de choisir l'ordre des octets (cf. note 9, p. 34) dans la représentation des données pour les lectures et écritures de fichiers binaires.

E.5.1 Options du compilateur g95 conseillées

- Le stockage statique des variables locales peut être évité avec l'option `-fno-static`.
- Il est possible de demander au compilateur d'interdire tous les écarts à la syntaxe du fortran standard avec l'option `-std=f958` ou `-std=f2003`. Cette option exclut l'emploi des procédures intrinsèques non standard, telles que la fonction `system` qui permet de lancer des commandes du système d'exploitation depuis le fortran. Mais des options⁹ sont prévues pour autoriser toutes ou partie seulement des extensions intrinsèques tout en respectant le standard par ailleurs.
- L'option `-Wimplicit-none` permet de signaler toutes les variables non déclarées. L'option `-fimplicit-none` interdit l'usage de variables non déclarées.
- Si on souhaite être informé qu'une variable a été déclarée, mais n'est pas utilisée¹⁰, il faut préciser l'option `-Wunused`.

8. On peut cependant dans ce cas autoriser certaines extensions relevant du standard 2003. En particulier l'allocation de tableaux dynamiques arguments de procédures (cf. 7.5.2, p. 82) et les tableaux allouables dans les structures (cf. 9.5, p. 95) en ajoutant l'option `-ftr15581`.

9. L'option `-fintrinsic-extensions` permet d'accéder à toutes les fonctions intrinsèques de `g95` alors que `-fintrinsic-extensions=` permet de spécifier la liste explicite des fonctions intrinsèques non-standard (par exemple les fonctions `GAMMA`, `ERF`, `ERFC`... et aussi la fonction `system`) à accepter en exigeant par ailleurs le respect d'un standard.

10. Ces avertissements peuvent paraître au premier abord superflus, en particulier dans la phase de mise au point du code. Mais ils peuvent permettre de découvrir des erreurs de codage de façon assez subtile, comme dans cet exemple, vécu par un étudiant : en signalant que, dans l'expression `x**(1/3)`, la variable `x` est en fait inutilisée, le compilateur `g95` a permis de détecter une grossière erreur de codage dans le calcul de la racine cubique. La division $1/3$ étant ici programmée en entier, elle donne un exposant nul, rendant l'expression indépendante de `x` car $x^0 = 1$.

- À l'inverse, l'option `-Wunset` avertit quand on n'attribue pas de valeur à une variable, ce qui est suspect si elle est utilisée.

Afin de bénéficier au mieux de l'analyse du code faite par le compilateur, il est utile de lui demander la plupart des avertissements disponibles : on utilise à cet effet l'option `-Wall` qui regroupe les avertissements plus importants et on peut ajouter `-Wextra` pour activer notamment `-Wmissing-intent` et `-Wobsolescent`.

♡ ⇒ Les options suivantes sont conseillées pour signaler des erreurs à l'exécution, au moins dans la phase de mise au point des programmes :

- `-fbounds-check` signale les dépassements de bornes des tableaux et des chaînes de caractères ;
- `-ftrapv` provoque une erreur lors d'un dépassement de capacité dans les opérations d'addition, de soustraction et de multiplication.
- `-freal=nan` initialise les variables scalaires réelles et complexes (non explicitement initialisées) à `not a number`, produisant des erreurs de calcul si on les utilise sans leur attribuer d'autre valeur.
- `-fpointer=null` initialise les pointeurs scalaires (non explicitement initialisés) à `null()`
- `-ftrace=full` permet retrouver le numéro de ligne d'une éventuelle erreur arithmétique, mais ralentit l'exécution.
- `-ffloat-store` interdit l'utilisation de registres de capacité supérieure pour les calculs flottants : en effet l'unité de calcul flottant (`Floating Processor Unit`) effectue bien souvent les calculs dans des registres de 80 voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Cette option ne doit être activée qu'à titre de test car elle dégrade les performances des programmes même si elle contribue à leur portabilité !
- `-fone-error` peut être activée pour arrêter la compilation dès la première erreur, évitant ainsi l'avalanche de messages concernant les erreurs provoquées par exemple par une déclaration inexacte.

`G95_MEM_INIT=NAN` cette variable d'environnement permet de choisir d'initialiser la mémoire allouée de façon dynamique, par exemple à `Not A Number`, quitte à ralentir l'allocation.

En pratique, sur les postes clients LINUX de l'UTES, des alias ou des fonctions du shell `g95-mni` et `g2003` ont été définis pour activer simplement les options conseillées.

Pour plus de détails, consulter le manuel du compilateur, maintenant disponible en ligne (VAUGHT, 2006), y compris traduit en français (<http://ftp.g95.org/G95Manual.fr.pdf>).

E.6 Compilateur gfortran

Le compilateur `gfortran` fait partie de `gcc`¹¹, la collection de compilateurs du GNU, capable de compiler les langages C, C++, Objective-C, Fortran, Java, et Ada. Plus précisément, il s'appuie sur `gcc`, suit sa syntaxe et admet donc ses options générales, mais propose aussi quelques options spécifiques pour le langage fortran.

Les deux compilateurs `gfortran` et `g95`, bien que suivant un développement séparé, proposent des fonctionnalités similaires et l'on ne détaillera donc pas ici celles de `gfortran`, laissant le lecteur se reporter à la documentation en ligne ([Site gfortran, de la collection de compilateurs gcc](#)).

E.6.1 Options du compilateur gfortran conseillées

- Par défaut, les variables locales sans l'attribut `SAVE` des procédures ne sont pas stockées en statique mais sur la pile, sauf si elles dépassent une taille spécifiée en octets par l'option `-fmax-stack-var-size` (défaut 32768). L'option `-frecursive` empêche leur stockage en mémoire statique.
- L'option `-fimplicit-none` interdit le typage implicite des variables comme le fait l'instruction `IMPLICIT NONE`.

11. En fait, c'est le compilateur fortran 90 officiel de `gcc` et il a atteint un degré de développement comparable à `g95` (cf. E.5, p. 152). Il est maintenant distribué avec les outils de développement dans la plupart des distributions linux (<http://gcc.gnu.org/wiki/GFortranBinaries>). Noter qu'il est parfois accessible sous le nom `f95`.

- L'option `-std=f95` ou `-std=f2003` permet d'exiger un code conforme à la norme fortran 95 ou fortran 2003. Les écarts à la norme requise génèrent des erreurs et les instructions obsolètes sont signalées par des avertissements. Cette option exclut l'emploi des procédures intrinsèques non standard, sauf si on ajoute l'option `-fall-intrinsics`¹² qui autorise toutes les procédures intrinsèques qui sont des extensions à la norme (comme `GAMMA`, `ERF`, `ERFC`... qui font partie du standard fortran 2008).
- L'option `-fmax-errors=n` peut être activée pour arrêter la compilation dès la n^e erreur, évitant ainsi l'avalanche de messages concernant les nombreuses erreurs provoquées par exemple par une déclaration inexacte.
- L'option `-fbacktrace` permet d'afficher la hiérarchie des appels avec les numéros de ligne en cas d'erreur à l'exécution.
- Les options suivantes initialisent respectivement les entiers `-finit-integer=<n>`, les réels et les complexes `-finit-real=snan`, par exemple à Not a Number de signalisation.

On conseille, pour la compilation, les options `-Wall` qui requièrent des avertissements classiques et `-W` pour des avertissements complémentaires. Pour la mise au point, `-fbounds-check`¹³ ou (à partir de la version 4.5) `-fcheck=bounds` active la vérification des bornes des tableaux à l'exécution.

L'option `-ffloat-store` de `gcc` a le même effet qu'avec `g95` (cf. E.5.1).

On notera que les variables d'environnement qui permettent d'ajuster le comportement de l'exécutable sont ici préfixées par `GFORTRAN_` au lieu de `G95_`. De plus, le choix de l'ordre des octets dans les fichiers binaires (cf. note 9, p. 34) peut se faire unité logique par unité logique grâce à la variable `GFORTRAN_CONVERT_UNIT`, comme avec `ifort` (cf. E.4.1, p. 152).

E.7 Options de compilation contrôlant le nombre d'octets des types numériques intrinsèques

E.7.1 Options passant sur 64 bits les entiers par défaut

processeur	compilateur	option
32 bits	<code>g95</code>	<code>-i8/-d8</code>
64 bits	<code>g95</code>	par défaut
32/64 bits	<code>gfortran</code>	<code>-fdefault-integer-8</code>
32/64 bits	<code>nagfor</code> (NAG)	<code>-double</code>
32/64 bits	<code>pgf95</code> (PORTLAND)	<code>-i8</code>
32/64 bits	<code>ifort</code> (INTEL)	<code>-integer_size 64</code> ou <code>-i8</code>
64 bits (alpha)	<code>f90</code> (HP)	<code>-integer_size 64</code>
64 bits (power PC)	<code>xlf</code> (IBM)	<code>-qintsize=8</code>

E.7.2 Options passant sur 64 bits les réels par défaut

processeur	compilateur	option
32/64 bits	<code>g95</code>	<code>-r8/-d8</code>
32/64 bits	<code>gfortran</code>	<code>-fdefault-real-8</code>
32/64 bits	<code>nagfor</code> (NAG)	<code>-r8/-double</code>
32/64 bits	<code>pgf95</code> (PORTLAND)	<code>-r8</code>
32/64 bits	<code>ifort</code> (INTEL)	<code>-real_size 64</code> ou <code>-r8</code>
64 bits (alpha)	<code>f90</code> (HP)	<code>-real_size 64</code>
64 bits (power PC)	<code>xlf</code> (IBM)	<code>-qrealsize=8</code>

12. Avec `g95`, il est possible de spécifier la liste des procédures non standard autorisées, cf. E.5.1, p. 153.

13. À partir de la version 4.5 `-fbounds-check` est un alias obsolète de `fcheck=bounds`.

Bibliographie

ADAMS, JEANNE, WALTER BRAINERD, RICHARD HENDRICKSON, RICHARD MAINE, JEANNE MARTIN et BRAIN SMITH, *The Fortran 2003 Handbook : The Complete Syntax, Features and Procedures*, 712 pages (Springer, 2009), ISBN 978-1846283789.

AKIN, ED, *Object-oriented programming via fortran 90/95*, 360 pages (Cambridge University Press, 2003), ISBN 0-521-52408-3.

Cet ouvrage présente les aspects programmation orientée objet du fortran 90/95 au travers de nombreux exemples commentés. Il comporte des tableaux synthétiques mettant en parallèle les syntaxes du fortran, du C++ et de MATLAB ainsi qu'une annexe résumant succinctement le langage fortran 90.

CHAPMAN, STEPHEN J., *Fortran 95/2003 for Scientists and Engineers*, 974 pages (Mc Graw-Hill, 2007), 3^e édition, ISBN 978-0-07-319157-7.

La troisième édition de ce livre est un des rares ouvrages qui présente délibérément le fortran du standard 2003, tout en distinguant explicitement par la présentation les éléments non disponibles dans le standard fortran 95. Toutefois, l'interopérabilité avec le C n'est pas abordée. Contrairement à METCALF *et al.* (2004), il peut être abordé par des débutants car l'approche est progressive et assortie de nombreux conseils, exemples et exercices.

CHIVERS, IAN et JANE SLEIGHTHOLME, *Introduction to Programming with Fortran With Coverage of Fortran 90, 95, 2003, 2008 and 77*, 619 pages (Springer, 2012), 2^e édition, ISBN 978-0-85729-232-2.

CLERMAN, NORMAN S. et WALTER SPECTOR, *Modern fortran : Style and Usage*, 360 pages (Cambridge University Press, 2011), ISBN 978-0521514538.

Comme son titre l'indique, cet ouvrage présente des règles de «bon usage» du fortran 2003 aussi bien dans l'écriture du code (nommage, mise en page, ...) que dans sa documentation. Il s'adresse donc à un public possédant déjà une certaine expérience du langage.

CORDE, PATRICK et HERVÉ DELOUIS, *Langage Fortran (F2003)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2012a, URL : http://www.idris.fr/data/cours/lang/fortran/choix_doc.html.

Un des rares documents en français présentant les nouveautés du fortran 2003 par rapport au fortran 95, en une série de 246 transparents. L'inter-opérabilité entre le fortran et le langage C et surtout les aspects objet du fortran 2003 y sont étudiés en détail.

CORDE, PATRICK et HERVÉ DELOUIS, *Langage Fortran (F95-2)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2012b, URL : http://www.idris.fr/data/cours/lang/fortran/choix_doc.html.

Une série de 288 transparents issus des cours de l'IDRIS présentant les aspects avancés du langage fortran 95. Cette présentation suppose une connaissance préalable des notions de base, présentées dans la partie (F95-1), FOUILLOUX et CORDE (2012). Des exercices sont disponibles en ligne et au format `.tar.gz`

E.7.2

DELANNOY, CLAUDE, *Programmer en Fortran 90 Guide complet*, 413 pages (Eyrolles, 1997), ISBN 2-212-08982-1.

Écrit par un auteur bien connu pour le succès de ses nombreux ouvrages sur les langages de programmation, ce livre présente le fortran 90 de façon progressive et concise. Il est accompagné d'exemples et d'exercices corrigés, et peut constituer un très bon manuel d'apprentissage.

DUBESSET, CLAUDE et JEAN VIGNES, *Les spécificités du Fortran 90*, 367 pages (Editions Technip, 1993), ISBN 2-7108-0652-5.

Cet ouvrage en français s'adresse avant tout à ceux qui connaissent déjà le fortran 77. Il présente avec beaucoup de précision et de clarté les apports de la norme fortran 90. Il est doté de précieuses annexes et notamment d'un lexique et d'un glossaire.

FOUILLOUX, ANNE et PATRICK CORDE, *Langage Fortran (F95-1)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2012, URL : http://www.idris.fr/data/cours/lang/fortran/choix_doc.html.

Une série de 261 transparents récemment révisés des cours de l'IDRIS présentant les notions de base du langage fortran 95. Les notions avancées sont présentées dans la partie (F95-2), [CORDE et DELOUIS \(2012b\)](#). Des exercices sont disponibles en ligne et au format `.tar.gz`.

[E.7.2](#)

LIGNELET, PATRICE, *Manuel complet du langage Fortran 90 et Fortran 95 : calcul intensif et génie logiciel*, 314 pages (Masson, 1996), ISBN 2-225-85229-4.

Au delà du langage fortran, cet ouvrage en français d'un auteur de nombreux ouvrages dans ce domaine, aborde les questions de calcul numérique, de vectorisation et de généricité.

Manuel Fortran DEC, *Digital Fortran, Langage Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, USA, 1997.

Le manuel papier du compilateur `f90` de DIGITAL EQUIPMENT CORPORATION, repris successivement par COMPAQ, puis par HP. Très bien présenté, il distingue clairement les extensions propriétaires de la norme du fortran 90.

Manuels Fortran IBM, *XL Fortran for AIX, Langage Reference and User's Guide*, IBM, 8200 Warden Avenue, Markham, Ontario, Canada, 2005, URL : <http://publib.boulder.ibm.com/infocenter/comphelp/>. [E.1](#)

Manuels Fortran Intel, *Intel Fortran Compiler Documentation*, Intel Corporation, 2010, URL : http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/compiler_f/index.htm.

Site d'INTEL qui présente un manuel en ligne du compilateur `ifort`.

[E.4](#)

Manuels Fortran NAG, *NAGWare Fortran 95 Compiler*, Numerical Algorithms Group, The Numerical Algorithms Group Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK, 2009, URL : <http://www.nag.co.uk/nagware/np.asp>.

Site du NUMERICAL ALGORITHMS GROUP permettant d'accéder aux pages de manuel en ligne du compilateur `nagfor` (ex `f95`) et des modules associés.

[E.2](#)

Manuels Fortran PGI, *PGI Fortran Compiler*, The Portland Group, The Portland Group, STMicroelectronics, Two Centerpointe Drive, Suite 320, Lake Oswego, OR 97035, 2011, URL : <http://www.pgroup.com/resources/docs.htm>.

Site de THE PORTLAND GROUP, permettant d'accéder au manuel de référence du compilateur fortran PGI au format pdf.

[E.3](#)

MARSHALL, A.C., J.S. MORGAN et J. L. SCHOFELDER, *Fortran 90 Course Notes*, The University of Liverpool, 1997, URL : <http://www.liv.ac.uk/HPC/F90page.html>.

Le site de l'Université de Liverpool propose une série de cours de différents niveaux sur le fortran 90 : les documents associés (transparentes, notes, exercices, codes ...) sont accessibles via ftp à l'url <ftp://ftp.liv.ac.uk/pub/F90Course/>.

METCALF, MICHAEL, «Fortran 90/95/HPF information file», 2004, URL : <http://www.fortran.com/metcalf.htm>.

Site dressant un inventaire des ressources sur fortran 90/95 (compilateurs, livres, cours, logiciels, ...). M. METCALF a maintenu ce site jusqu'en septembre 2004, date à laquelle il a été figé, dans la perspective de la norme fortran 2003.

METCALF, MICHAEL, JOHN REID et MALCOLM COHEN, *Fortran 95/2003 explained*, 434 pages (Oxford University Press, 2004), 3^e édition, ISBN 0-19-852693-8.

Rédigée par des promoteurs des nouveaux standards du fortran, la troisième édition de ce livre est un des rares ouvrages qui aborde les apports du fortran 2003. Il constitue une référence majeure dans ce domaine. Les apports de la norme fortran 2003 y sont présentés dans les 7 derniers chapitres, permettant ainsi de les séparer de la norme fortran 95. Il aborde notamment l'interopérabilité avec le C. Cette référence n'est cependant pas conseillée pour débiter.

E.7.2

METCALF, MICHAEL, JOHN REID et MALCOLM COHEN, *Modern Fortran explained*, Numerical Mathematics and Scientific Computation, 488 pages (Oxford University Press, 2011), 4^e édition, ISBN 978-019-960142-4.

La quatrième édition de ce classique est le premier ouvrage à aborder la norme 2008 du fortran. Succédant à l'édition METCALF *et al.* (2004) sur le fortran 2003, cette version révisée comporte notamment un chapitre sur les co-tableaux (`coarrays`) et un sur les nouveautés du fortran 2008. Cette référence n'est cependant pas conseillée pour débiter et rares sont encore les compilateurs à honorer la norme 2008.

1.1

NYHOFF, LARRY R. et SANDFIRD C. LEESTMA, *Fortran 90 for Engineers and Scientists*, 1070 pages (Prentice-Hall, 1997), ISBN 0-13-6571209-2.

Un ouvrage volumineux mais très bien illustré avec des applications concrètes dans le domaine du calcul scientifique. Sa présentation très progressive permet de l'utiliser pour découvrir le fortran 90 y compris comme premier langage de programmation.

OLAGNON, MICHEL, *Traitement des données numériques avec Fortran 90*, 244 pages (Masson, 1996), ISBN 2-225-85259-6.

Cet ouvrage ne présente pas le langage fortran en tant que tel, mais des méthodes de traitement des données (en particulier statistiques) mises en œuvre en fortran 90. Cette approche pragmatique s'appuyant sur de nombreux exemples permet d'aborder rapidement les applications tout en respectant les règles de bon usage du langage.

PRESS, WILLIAM H., SAUL A. TEUKOLSKY, WILLIAM T. VETTERLING et BRAIN P. FLANNERY, *Numerical Recipes in Fortran 90*, 551 pages (Cambridge University Press, 1996), 2^e édition, ISBN 0-521-57439-0.

Une référence très classique en analyse numérique, aussi disponible dans d'autres langages (pascal, C et C++). Bien la distinguer du manuel *Numerical Recipes in Fortran 77* qui comporte la discussion des algorithmes et leur mise en œuvre en fortran 77 et constitue le volume 1 du traité. Le volume 2 présente les codes associés en fortran 90, mais aussi dans son premier chapitre (21) une très précieuse introduction aux fonctionnalités du fortran 90. Noter enfin que les chapitres de cet ouvrage sont consultables en ligne : <http://www.library.cornell.edu/nr/>

Site gfortran, de la collection de compilateurs gcc, *Documentation de gfortran*, Free Software Foundation, 2011, URL : <http://gcc.gnu.org/onlinedocs/gfortran/>. A, E.6

STALLINGS, W., *Organisation et architecture de l'ordinateur*, 829 pages (Pearson Education, 2003), sixième édition, ISBN 9782744070075. 2.2, C.4.2

VAUGHT, ANDY, *Manuel de g95*, 2006, URL : <http://ftp.g95.org/G95Manual.pdf>. E.5.1

VAUGHT, ANDY, *Documentation de g95*, 2009, URL : <http://www.g95.org/docs.html>.

Le site du compilateur libre g95 d'où il est possible de télécharger les binaires pour les différentes architectures. Ce site comporte des liens vers la documentation. Il liste les bibliothèques qui ont été compilées avec succès avec g95.

E.5

VICKERY, CHRISTOPHER, «IEEE-754 calculators», 2009, URL : <http://babbage.cs.qc.edu/IEEE-754/>. C.4.2

WIKIPEDIA, «The IEEE-754-2008 standard», 2010, URL : http://en.wikipedia.org/wiki/IEEE_754. C.4.2

- ATAN2 125, 147
 ATANH 125
 automatique *voir* variable, tableau
- B –
- B'n' constante entière en binaire 14
 BACK=
 mot-clef de INDEX 87
 mot-clef de SCAN 87
 mot-clef de VERIFY 87
 BACKSPACE 41
 balise d'enregistrement 34
 base 8, 10, 130, 140
 bibliothèque 3, 4, 16, 33, 98, 115, 125, 147
 big endian 10, 34, 151-153, 155
 binaire
 base *voir* base
 constante entière en *voir* Z'n'
 format *voir* Bn.p
 BIND(C) 94, 116
 bit 8-16
 de signe 9, 11, 140, 143, 144
 manipulation de 131
 BIT_SIZE 10-12, 15, 130
 BLANK 37
 BN, format ignorer les blancs 43
 Bn.p, format binaire 10, 43, 46, 144
 booléen 8, 14, 21, 63
 bornes *voir* tableau
 boucle
 avec compteur 25
 implicite 45, 72
 infinie 26
 WHILE 25
 BOUND 1
 BTEST 131
 BZ, format blanc=zéro 43
- C –
- c_char 115
 c_double 115
 c_f_pointer 116, 121
 c_f_procpointer 116
 c_float 115
 c_funloc 116
 c_funptr 116
 c_int 115
 c_loc 104, 116
 c_ptr 116, 121
 CALL 51
 caractères 8, 84-91, 116, 134
 CEILING 17, 127, 147
 chaîne de caractères 8, 84-91, 116, 134
 automatique 90
 CHAR 88, 134
 CHARACTER 8, 84, 116
- cible (d'un pointeur) 104-107
 anonyme 106
 nommée 105
 CLOSE 39
 CMLX (conversion vers le type) 127
 codage
 des caractères 38, 40, 85
 des entiers 8-10
 des flottants 10-11
 norme IEEE 140-144
 comma *voir* DECIMAL
 COMMAND_ARGUMENT_COUNT 132
 commentaire 6
 COMMON 60
 comparaison 20
 compilateur
 g95 149, 152
 gfortran 154
 ifort d'INTEL 45, 149, 152
 nagfor (ex-f95) de NAG 149, 150
 pgf95 de PORTLAND 149, 151
 xlf sur IBM 149
 compilation 3
 de module 58
 options de
 avec g95 153
 avec gfortran 154
 avec ifort d'INTEL 152
 avec nagfor 150
 avec pgf95 de PORTLAND 151
 avec xlf 149
 séparée 4, 59
 COMPLEX (type) 8
 complexe *voir* COMPLEX, *voir aussi* CMLX
 composante 92, 95, 116
 concaténation 21, 85
 conformants *voir* tableaux
 CONJG 125, 147
 constante 14
 constante symbolique 15
 constructeur *voir* tableau, *voir aussi* structure
 CONTAINS 55, 92
 continuation d'instruction 7, 84
 CONTINUE 27
 conversion
 de type 17, 100, 127
 implicite 19
 COS 125, 147
 COSH 125, 147
 COUNT 78, 130
 CPU_TIME 134
 CSHIFT 77
 CYCLE 26, 146
- D –
- d8, option de g95 155

- DATE_AND_TIME 132
dc (decimal comma) 31, 44
DEALLOCATE 81, 106
débogueur 4
debugger voir débogueur
DECIMAL 38, 153
déclaration 13-14, 14-16, 52, 56-62
 d'arguments 52, 63
 d'interface 64
 d'une fonction 54
 de chaîne de caractères 84, 87
 de cible 104
 de fonction 64, 66
 de pointeur 104, 105
 de tableau 70-72, 79
 allouable 81
 de type dérivé 93, 94
DEFAULT 24
définition 61
 d'un type dérivé 92, 95, 139
'DELETE', argument de STATUS dans CLOSE 39
DELIM=, mot-clef de OPEN 37
dénormalisé 11, 141
dépassement
 de bornes 150, 154
 de capacité 12, 16, 19, 125, 141, 150, 154
descripteur
 actif 42
 de contrôle 43
DIGITS 11, 12, 129, 141
DIM= 75-78
DIMENSION 13, 70
dimension voir tableau
'DIRECT', argument de ACCESS dans OPEN 37
DO 25, 146
DO WHILE 25, 146
domaine 10, 130, 140, 141, 144, 151, 154
DOT_PRODUCT 76, 130
DOUBLE PRECISION 8
dp (decimal point) 44
- E —
- édition de liens 2, 3, 59
effet de bord 69
ELEMENTAL, attribut de procédure 69
ELSE IF 23
ELSEWHERE 75
En.p, format à virgule flottante 43
encapsulation 61, 103
encoding
 argument de INQUIRE 40
 argument de OPEN 38
 option de nagfor 150
END
 DO 25
 FORALL 79
 FUNCTION 54, 139
 INTERFACE 57
 MODULE 58, 139
 PROGRAM 3, 139
 SELECT 23
 SUBROUTINE 54, 139
 TYPE 92
 WHERE 75
END= 28, 39
ENDFILE 41
ENn.p, format ingénieur à virgule flottante 43
enregistrement 33, 34-35, 37, 39-45
entrée standard voir unités logiques
environnement 131
EOR= 28, 39
EOSHIFT 77
EPSILON 12, 15, 129, 143, 144, 148
ERF 125, 153, 155
ERFC 125, 153, 155
ERR= 28, 37
erreur 2
 d'exécution 2, 31, 34, 154, 155
 de compilation 2, 3
error_unit (erreur standard) 33, 36
ESn.p, format scientifique virgule flottante 43
étendue voir tableau
étiquette numérique 6, 27, 36, 146
EXIST= mot-clef de INQUIRE 40
EXIT 26, 146
EXP 125, 147
EXPONENT 12, 129, 140
exposant 10, 141
exposant biaisé 141
EXTENDS (type dérivé) 94, 116
EXTERNAL 13, 64
- F —
- f95 de NAG voir compilateur
-fall-intrinsics, option de gfortran .. 155
-fbacktrace, option de gfortran 155
-fbounds-check
 option de g95 154
 option de gfortran 155
-fcheck=bounds, option de gfortran 155
-fdefault-integer-8, option de gfortran ..
 155
-fdefault-real-8, option de gfortran .. 155
-ffloat-store
 option de g95 19, 154
 option de gfortran 19, 155
fichier
 de module 4, 59
 exécutable 2, 3
 externe 32
 formaté 33, 34
 interne 33, 45, 89

non-formaté 33, 34
 objet 2, 4
 FILE= 37
 FILE_STORAGE_UNIT 36
 -fimplicit-none
 option de g95 153
 option de gfortran 154
 -finit-integer=, option de gfortran ... 155
 -finit-real=, option de gfortran 155
 -fintrinsic-extensions, option de g95 . 153
 -float-store, option de nagfor 19, 151
 FLOOR 17, 127, 147
 -fmax-errors=n, option de gfortran .. 2, 155
 -fmax-stack-var-size=, option de gfortran
 154
 -fmode, option de g95 153
 FMT=
 mot-clef de OPEN 39
 mot-clef de READ 39
 mot-clef de WRITE 40
 Fn.p, format à virgule fixe 43
 fonction 53, 54
 -fone-error, option de g95 2, 154
 FORALL 69, 79
 FORM=, mot-clef de OPEN 37
 format
 binaire voir Bn.p
 chaîne voir An
 décimal voir In.p
 descripteur de 42
 fixe 1, 6, 6
 hexadécimal voir Zn.p
 libre 6, 6-7
 octal voir On.p
 variable 45, 89
 FORMAT= 42
 'FORMATTED', argument de FORM dans OPEN 37
 -fpointer, option de g95 105, 154
 FRACTION 12, 129, 140
 -freal, option de g95 154
 -frecursive, option de gfortran 154
 -ftrace=, option de g95 154
 -ftrapv, option de g95 12, 154
 fuite de mémoire 106
 FUNCTION 53, 54, 67, 90, 108, 139

– G –

g95 voir compilateur
 G95_COMMA 44, 153
 G95_ENDIAN 153
 G95_MEM_INIT 154
 G95_STDERR_UNIT 153
 G95_STDIN_UNIT 153
 G95_STDOUT_UNIT 153
 GAMMA 125, 153, 155
 GET_COMMAND 131, 132

GET_COMMAND_VARIABLE 132
 GET_ENVIRONMENT_VARIABLE 131, 132
 gfortran voir compilateur
 GFORTRAN_CONVERT_UNIT 155
 Gn.p, format général 43
 GO TO 27, 146

– H –

hexadécimal
 constante entière en voir Z'n'
 format voir Zn.p
 HUGE ... 10, 11, 12, 15, 129, 141, 143, 144, 148
 HYPOT 125

– I –

-i8, option de g95 155
 IACHAR 88, 134
 IAND 131, 146
 IBCLR 131
 IBITS 131
 IBSET 131
 ICHAR 88, 134
 IEEE (norme) 13, 140-144, 151
 IEOR 131, 146
 IF 22, 146
 ifort voir compilateur
 IF ... THEN 22, 146
 imaginaire voir AIMAG
 IMPLICIT NONE 13
 IMPORT 62, 117
 IN voir INTENT
 In.p, format décimal 43
 INCLUDE 58
 INDEX 87, 135
 indexation voir tableau
 Inf 141
 initialisation 14, 52, 69, 72, 105
 INOUT voir INTENT
 input_unit (entrée standard) 33, 36
 INQUIRE 40, 49
 INT 17, 127
 INTEGER 8-10, 12, 16
 INTENT 13, 52, 69, 82, 117
 INTERFACE 57, 66
 ASSIGNMENT 100
 OPERATOR 98
 intrinsèque voir procédure, module, type
 intrinsèques
 INTRINSIC
 module 63
 procédure 13
 IOR 131, 146
 IOSTAT 29, 37, 39
 IOSTAT_END (fn de fichier) 29, 36, 39
 IOSTAT_EOR (fn d'enregistrement) .. 29, 36, 39
 IS_IOSTAT_END 29, 36, 39

IS_IOSTAT_EOR 29, 36, 39
 ISHFT 131, 146
 ISHFTC 131
 ISO_104646 134
 ISO_C_BINDING (module) ... 63, 104, 115-116
 ISO_FORTRAN_ENV (module) 29, 33, 36, 39, 63

– J –

justification
 à droite voir ADJUSTR
 à gauche voir ADJUSTL

– K –

'KEEP', argument de STATUS dans CLOSE .. 39
 KIND . 8, 11, 15, 15, 17, 115, 116, 129, 140, 150

– L –

LBOUND 15, 75, 130
 ld éditeur de liens 2, 3
 LEN 15
 fonction 87, 90, 134
 paramètre de type 14, 84
 LEN_TRIM 87, 90, 134
 LGE 88, 134
 LGT 88, 134
 library voir bibliothèque
 ligne
 d'entrée-sortie ... 30, 31, 33, 35, 41, 43, 88
 d'instruction 6, 7, 84
 de commentaire 6
 link voir édition de liens
 liste chaînée 112
 liste d'entrée-sortie 35
 little endian 10, 34, 151-153, 155
 LLE 88, 134
 LLT 88, 134
 Ln, format booléen 43
 LOG 125, 147
 LOG10 125, 147
 LOG_GAMMA 125
 LOGICAL
 fonction 127
 type 8, 145

– M –

-M, option de g95 5, 153
 majuscule 3, 5, 150, 153
 make 4
 makefile 5, 153
 mantisse 10, 43, 44, 129, 141
 MASK= 78, 130-131
 masquage de variable 56
 masque voir tableau
 MATMUL 76, 130
 MAX 125
 MAXEXPONENT 129
 MAXLOC 76, 130

MAXVAL 76, 130
 memory leak voir fuite de mémoire
 MERGE 77, 130
 MIN 125
 MINEXPONENT 129
 MINLOC 76, 130
 minuscule 3, 5, 150, 153
 MINVAL 76, 130
 MOD 126, 147
 MODULE 58-63, 139
 module 16, 58, 92, 95
 fichier de 4, 59
 intrinsèque 63
 MODULO 126, 147
 mot-clef 64, 93

– N –

nagfor de NAG voir compilateur
 NAME=, mot-clef de BIND 116
 NaN, Not a Number 19, 141
 NEAREST 12, 129, 143, 144, 147
 'NEW', argument de STATUS dans OPEN 37
 NEW_LINE 35, 88, 134
 NINT 17, 127
 normalisé 11, 141
 NOT 131, 146
 NULL 105, 131, 154
 NULLIFY 105

– O –

O'n' constante entière en octal 14
 octal
 constante entière en voir O'n'
 format voir On.p
 od (octal dump) 34
 'OLD', argument de STATUS dans OPEN 37
 On.p, format octal 43, 46
 ONLY: voir USE
 OPEN 37
 OPENED=, mot-clef de INQUIRE 40
 opérateur
 création d' 98
 standard du fortran 18
 surcharge d' 98
 OPERATOR voir INTERFACE
 OPTIONAL 13, 63
 options de compilation voir compilation
 ORDER, mot-clef de RESHAPE 77, 130
 OUT voir INTENT
 output_unit (sortie standard) 33, 36
 overflow voir dépassement de capacité

– P –

PACK 77, 130
 PARAMETER 13, 15, 79, 84
 partie entière voir CEILING, FLOOR, INT,
 NINT

- partie réelle *voir* REAL
 pgf95 *voir* compilateur
 point *voir* DECIMAL
 POINTER 13, 69, 104-114
 pointeur 104-114, 116, 131, 147
 portée 52, 60
 POS=, mot-clef de INQUIRE, READ, WRITE 39-41
 POSITION, mot-clef de OPEN 37, 41
 PRECISION 12, 13, 130
 précision numérique 10-12, 15, 16, 18, 140-144,
 151
 prédécesseur 12, 129
 PRESENT 63
 PRINT 30, 40
 priorité des opérateurs 18
 PRIVATE 13, 61, 92, 103, 116
 procédure
 élémentaire 15, 69, 86-88, 90, 125
 externe 56
 générique 97
 interface de 57
 interne 55
 intrinsèque 13, 15, 125, 153
 pure 69
 récursive 67, 112
 spécifique 97
 standard 125, 153
 PRODUCT 76, 130
 produit matriciel 76
 produit scalaire 76
 profil *voir* tableau
 PROGRAM 3, 139
 PROTECTED 13, 62
 PUBLIC 13, 61, 92, 116
 PURE, attribut de procédure 69
- Q –
- quiet NaN 141
- R –
- r8, option de g95 155
 RADIX 12, 130
 RANDOM_NUMBER 127
 RANDOM_SEED 127
 rang *voir* tableau
 RANGE 9, 11, 130
 range *voir* domaine
 READ 30, 39
 REAL
 fonction 17, 127
 type 8, 16, 145
 REC=, mot-clef de READ ou WRITE 39, 40
 recherche
 de caractères *voir* SCAN, VERIFY
 de motif *voir* INDEX
 RECL=, mot-clef OPEN 36, 37
 record *voir* enregistrement
 record marker *voir* balise d'enregistrement
 RECURSIVE 67, 69
 registre 151, 154
 REPEAT 15, 88, 134
 'REPLACE', argument de STATUS dans OPEN 37
 RESHAPE 15, 72, 77, 130
 reste *voir* MODULO, *voir* MOD
 RESULT 67
 RETURN 28, 28, 146
 REWIND 41
 RRSPACING 130
 run-time error *voir* erreur d'exécution
- S –
- S, format signe plus selon processeur 43
 SAVE 13, 14, 52, 67, 69
 SCALE 130
 SCAN 87, 135
 scope *voir* portée
 'SCRATCH', argument de STATUS dans OPEN 37
 section *voir* tableau
 SELECT CASE 23, 146
 SELECTED_CHAR_KIND 85, 134
 SELECTED_INT_KIND 15, 16, 130
 SELECTED_REAL_KIND 15, 16, 130
 SEQUENCE 92, 94, 95, 116
 'SEQUENTIAL', argument de ACCESS dans OPEN
 37
 SET_EXPONENT 130
 SHAPE 15, 75, 130
 SIGN 126
 signaling NaN 141, 155
 signe
 bit de *voir* bit de signe
 format d'entrée-sortie *voir* SP, SS
 SIN 126, 147
 SINH 126, 147
 SIZE 15, 75, 130
 sortie d'erreur standard .. *voir* unités logiques
 sortie standard *voir* unités logiques
 sous-chaînes 85
 sous-programme 53, 54
 SP, format signe plus 43
 SPACING 12, 130
 SPREAD 77, 130
 SQRT 126, 147
 SS, format sans signe plus 43
 statique *voir* variable statique
 STATUS=
 mot-clef de CLOSE 39
 mot-clef de OPEN 37
 -std=f2003, option de g95 153
 -std=f2003, option de gfortran 155
 -std=f95, option de g95 153
 -std=f95, option de gfortran 155

- STOP **27, 28, 69, 146**
- 'STREAM', argument de ACCESS dans OPEN **37, 44**
- string *voir* chaîne de caractères
- structure **92, 116**
 constructeur de **93**
- SUBROUTINE **51, 53, 54, 67, 90, 139**
- successeur **12, 129**
- SUM **76, 131**
- system **153**
- SYSTEM_CLOCK **134**
- T –
- tableau
- à profil
- explicite **79**
- implicite **80**
- automatique **80**
- bornes d'un **70, 75, 107**
- constructeur de **72**
- de chaînes **89**
- dimensions d'un **70**
- dynamique **79**
- et pointeurs **109**
- étendue d'un **70**
- indexation de **70-73, 76, 119**
- masque **78**
- pointeur sur un **107**
- profil d'un **70, 75, 107**
- rang d'un **70, 107**
- section de **72**
- non-régulière **73**
- régulière **72, 107**
- taille d'un **70**
- tableaux
- conformants **70**
- taille *voir* tableau
- TAN **126, 147**
- TANH **126, 147**
- TARGET **13, 105**
- temps **132**
- THEN *voir* IF ... THEN
- TINY **11, 15, 130, 141, 143, 144, 148**
- Tln, format tabulation vers la gauche **43**
- Tn, format tabulation **43**
- tr15581, option de g95 **82, 95, 108, 153**
- TRANSFER **15, 131**
- TRANSPOSE **77, 131**
- TRIM **15, 86, 134**
- TRn, format tabulation vers la droite **43**
- TYPE **92**
- type **70, 97, 145**
- dérivé **16, 92, 99, 103, 104, 109, 112, 116, 121**
- implicite **13**
- interopérable **115**
- intrinsèque **8**
- variante de .. **12, 15-16, 16, 19, 70, 85, 97**
- U –
- UBOUND **15, 75, 131**
- underflow *voir* dépassement de capacité
- 'UNFORMATTED', argument de FORM dans OPEN **37**
- Unicode **38, 40, 85, 150**
- UNIT= **41**
- mot-clef de CLOSE **39**
- mot-clef de INQUIRE **40**
- mot-clef de OPEN **37**
- mot-clef de READ **39**
- mot-clef de WRITE **39**
- unités logiques **33, 36, 37, 39, 40**
- entrée standard **33, 153**
- sortie d'erreur standard **33, 153**
- sortie standard **33, 153**
- 'UNKNOWN', argument de STATUS dans OPEN **37**
- UNPACK **77, 131**
- USE **58, 69**
- => **16, 62**
- ONLY: **61**
- UTF-16 **85**
- UTF-32 **85**
- UTF-8 **38, 40, 85, 150**
- V –
- VALUE, attribut d'argument **13, 117**
- variable
- automatique **52, 151-154**
- locale **52**
- statique ... **13, 14, 52, 52, 60, 67, 151-154**
- VERIFY **87, 135**
- visibilité **56-62**
- W –
- Wall, option de g95 **154**
- Wextra, option de g95 **154**
- WHERE **75, 79**
- Wimplicit-none, option de g95 **153**
- WRITE **39-40**
- X –
- X, format espace **43**
- xlf *voir* compilateur
- Z –
- Z'n' constante entière en hexadécimal . **14, 85**
- zéro
- division par **141**
- en entrée-sortie **37, 43**
- mise à zéro de bit **131**
- norme IEEE **141, 142**
- remplissage par **77, 131**
- troncature vers **17**
- Zn.p, format hexadécimal **43, 46**