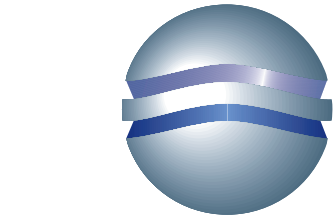


ECOLE DES MINES D'ALBI
C A R M A U X

Initiation au Fortran

Olivier LOUISNARD
Jean-Jacques LETOURNEAU
Paul GABORIT

1997 – 2000

Table des matières

Avant-propos

Ce document a été initialement rédigé par Olivier Louisnard[1] puis relu, corrigé et remis en page par Jean-Jacques Letourneau[2] et Paul Gaborit[3].

Toute erreur, omission, imprécision ou incohérence pourra être rapportée aux auteurs qui s'efforceront d'y remédier. Sachez tout de même que ce document n'est en rien un manuel de référence du langage Fortran. Certains aspects du langage ont été volontairement passés sous silence ou présentés de manière simplifiée dans un souci pédagogique.

La référence est le Fortran 77 qui reste (malheureusement, diraient certains) un langage très utilisés dans le domaine du calcul numérique par l'industrie et le monde de la recherche. La raison principale est les milliers de bibliothèques contenant des millions de lignes de code Fortran qui ont été développées, testées et validées depuis plus de 30 ans. C'est un investissement lourd qu'il faut continuer d'exploiter.

Ce document ne peut être diffusé à l'extérieur de l'École des Mines d'Albi sans l'accord explicite de ses auteurs.

Paul Gaborit

[1] mailto:louisnar@enstimac.fr
[2] mailto:letourne@enstimac.fr
[3] mailto:gaborit@enstimac.fr

Chapitre 1

Introduction

1.1 Composition d'un ordinateur

- 1 microprocesseur pour calculer
- de la mémoire pour ranger des données

Le microprocesseur ne sait effectuer que des opérations simples sur des nombres codés en binaire (1 et 0) :

- additionner
- soustraire
- multiplier
- diviser
- lire dans la mémoire
- écrire dans la mémoire
- ...

C'est le *code machine*, très éloigné de la logique humaine.

1.2 Exemples de problème

- Multiplier deux matrices
  Le microprocesseur ne sait pas de lui-même manipuler de tels objets. Il faut une structure capable d'organiser une matrice en mémoire comme l'illustre la figure 1.1.
- Calculer $sin(x)$ pour 100 valeurs de x régulièrement espacées sur $[0, 2\pi]$.
  Le microprocesseur n'a pas d'instruction capable de calculer le sinus d'un nombre réel.
- Selon le résultat de la comparaison entre deux réels, effectuer une tâche ou une autre.

1.3 Un langage

On voit donc qu'il manque un chaînon entre l'homme et la machine, un langage commun. C'est le langage informatique.

Un langage est constitué par :
– un ensemble de mots-clés
– un ensemble d'objets manipulables, éventuellement extensible
– des règles de syntaxe
– de structures logiques
Programmer, c'est écrire un texte respectant les règles du langage, susceptible de résoudre un problème donné.

1.4 Compilation et debugger

Ce texte est ensuite vérifié et traduit en une suite de codes machines par l'intermédiaire d'un *compilateur*. Si le texte est incorrect, le compilateur indique les *erreurs de compilation*, qu'on pourrait comparer à des fautes d'orthographe et de grammaire dans un langage courant.

Exécuter le programme, c'est faire dérouler par la machine cette séquence de codes machines ainsi créée.

Il est malheureusement rare qu'un programme fonctionne du premier coup, et qu'il fournisse exactement le résultat escompté : il présente des dysfonctionnements qu'on appelle des « bugs ».

On dispose en général d'un outil appelé *debugger*, qui permet de faire tourner le programme par petits bouts, afin de repérer les erreurs (cf. figure 1.2).

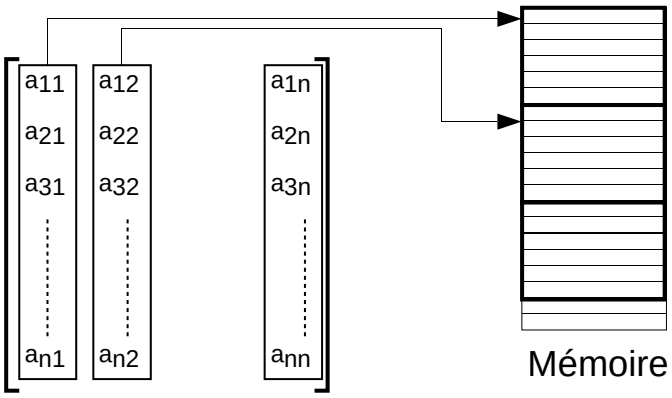


FIG. 1.1 – Organisation d'une matrice en mémoire

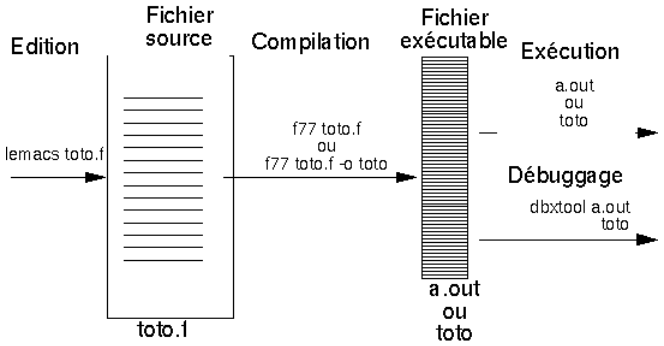


FIG. 1.2 – chaîne de compilation

Chapitre 2

Généralités sur le langage Fortran

Les contenu des lignes en Fortran doit respecter les quelques règles suivantes :

- Une ligne de **commentaire** doit commencer par un `c` ou une `*` en première colonne.
- Tout ce qui suit un `!` dans une ligne est aussi considéré comme un **commentaire**.
- Les **instructions** :
  - doivent commencer à partir de la colonne 7 (ou plus) ;
  - se placent automagiquement avec[X]Emacs en appuyant sur la touche `TAB` ;
  - ne doivent pas dépasser la colonne 72.
- Les **Labels** (ou étiquettes) sont entre les colonnes 2 et 5 et gérés automagiquement avec [X]Emacs.
- Pour **couper** une instruction sur plusieurs lignes, on met un caractère quelconque en colonne 6 des lignes de suite.

La figure 2.1 illustre ces règles.

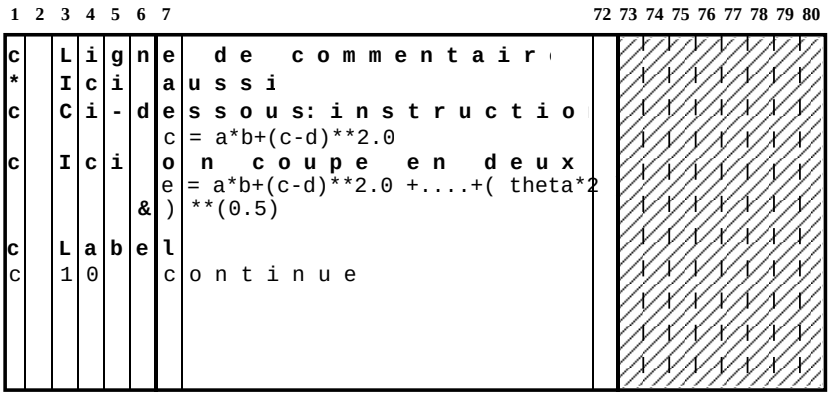


FIG. 2.1 – Exemples de contenu de lignes en Fortran

2.1 Organisation d'un programme FORTRAN

Succession de « *pavés* » élémentaires qu'on appellera *blocs fonctionnels*. Il en existe 3 sortes :

1. *Le programme principal* inclus entre `program` (facultatif) et `end`

```
program nom
...
end
```

2. *Les subroutines* inclus entre `subroutine` et `end`

```
subroutine nom (arguments)
...
end
```

3. *Les fonctions* inclus entre `function` et `end`

```
type function nom (arguments)
...
end
```

2.2 Programme principal

Le programme principal est obligatoirement présent. Il n'existe qu'un seul programme principal. Ce programme principal se découpe en deux parties distinctes successives détaillées ci-dessous.

2.2.1 Partie déclaration

C'est dans cette partie qu'on définit les objets (type + nom) qui seront manipuler par le programme.

2.2.2 Partie instructions

L'exécution d'un programme FORTRAN consiste à dérouler dans l'ordre toutes les instructions de la partie exécutable du programme principal.

Certaines instructions déroutent le pointeur de programme vers d'autres blocs fonctionnels (subroutines ou fonctions)

Chapitre 3

Les données

3.1 Les différents types de données

Le tableau 3.1 résume l'ensemble des types de données manipulables en Fortran. De plus, il est possible d'assembler plusieurs grandeurs dans des tableaux. Ce qui permet de créer des vecteurs, des matrices...

Nous allons maintenant détaillé ces types.

3.1.1 Type `integer`

Un `integer` contient un entier et est représenté par son écriture en base 2 signée sur 4 octets (31 bits pour la valeur plus un bit pour le signe). Ses valeurs possibles sont dans l'intervalle $[-2^{31}, 2^{31} - 1]$.

3.1.2 Type `real`

un `real` contient un nombre réel et est codé en virgule flottante (IEEE) sur 4 octets [1].

Chaque nombre est représenté sous la forme $x = \pm 0.m \times 2^e$ où m est la *mantisse* codée sur 23 bits et e est *l'exposant*, codé sur 8 bits ($-127 < e < 128$).

[1] cf. documentation FORTRAN SUN §4.2

TAB. 3.1 – Types de données manipulables en Fortran.

Grandeurs numériques	Entiers	`integer`
	Réels	`real`
		`double precision`
	Complexes	`complex`
Caractères		`character`
Grandeurs logiques (vraies ou fausses)		`logical`

Les valeurs (en valeur absolue) sont comprises dans l'intervalle $[1.401 \times 10^{-45}, 3.403 \times 10^{38}]$ et il stocke environ 7 chiffres significatifs.

3.1.3 Type `double precision`

Le `double precision` est un `real` plus précis, codé en virgule flottante sur 8 octets dont une mantisse codée sur 52 bits et un exposant codé sur 11 bits ($-1023 < e < 1024$).

Les valeurs (en valeur absolue) sont comprises entre $[4.941 \times 10^{-324}, 1.798 \times 10^{308}]$ avec 15 chiffres significatifs.

3.1.4 Type `complex`

Assemblage de 2 `real` dans un même objet.

3.2 Constantes numériques

Question : je veux utiliser dans un programme les nombres 1, 3.14, $2 + 3i$. Comment les écrire ?

3.2.1 Constantes `integer`

Une constante de type `integer` est écrite sans point décimal.

Exemples :

```
1
123
-28
0
```

3.2.2 Constantes `real`

Une constante de type `real` doit obligatoirement comporter : – soit le point décimal, même s'il n'y a pas de chiffres après la virgule ; – soit le caractère `e` pour la notation en virgule flottante. Pour les nombres écrits `0.xxxxx`, on peut omettre le `0` avant le point décimal.

Exemples :

```
0.
1.0
1.
3.1415
31415e-4
1.6e-19
1e12
.001
-36.
```

3.2.3 Constantes `double precision`

Une constante `double precision` doit obligatoirement être écrite en virgule flottante, le `e` étant remplacé par un `d`.

Exemples :

```
0d0
0.d0
1.d0
1d0
3.1415d0
31415d-4
1.6d-19
1d12
-36.d0
```

3.2.4 Constantes `complex`

Une constante de type `complex` est obtenue en combinant deux constantes réelles entre parenthèses séparées par une virgule. $2.5 + i$ s'écrira (2.5,1.)

Exemples :

```
(0.,0.)
(1.,-1.)
(1.34e-7, 4.89e-8)
```

3.3 Définition de constantes symboliques

Elle permettent de référencer une constante à l'aide d'un symbole.

Elles ne peuvent être modifiées au milieu du programme et sont affectées une fois pour toutes avec le mot-clé `parameter` dans la section déclarations.

3.3.1 Syntaxe

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

Le type de chaque constante doit être déclaré explicitement ou en suivant les mêmes règles de typage automatique que les variables (cf 4.3). Une constante est toujours locale à un bloc fonctionnel.

3.3.2 Exemple

```
      double precision q
      parameter(max=1000, q=1.6d-19)
```

Chapitre 4

Les variables

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

Les variables permettent (entre autres) de :
– manipuler des symboles ;
– programmer des formules.
Avant d'utiliser une variable, il faut :
– définir son type ;
– lui donner un nom.
C'est la *déclaration*. Elle doit être écrite dans la première partie (la partie déclaration) d'un bloc fonctionnel (programme principal, subroutine ou fonction) dans lequel intervient la variable.

Dans les langages modernes, *toute variable doit être déclarée*. En FORTRAN, il y a des exceptions obéissant à des règles bien précises.

Une variable est : – *locale* si seul le bloc fonctionnel où elle est déclarée peut y accéder. C'est le défaut ; – *globale* si tous les blocs fonctionnels peuvent y accéder.

Pour savoir comment rendre une variable globale, voir le paragraphe concernant l'instruction `common` (10.6).

4.1 Déclaration de variables

4.1.1 A quel endroit ?

Entre le mot-clé | `program`
`subroutine`
`function` | et la première instruction exécutable.

4.1.2 Syntaxe

```
type var1, var2, var3, .....
```

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

4.1.3 Exemple

```
      integer i,j,k
      real alpha, beta
      double precision x,y
      complex z
```

4.2 Noms des variables

Pour nommer une variable, il faut respecter les règles suivantes :
- Caractères autorisés :
  - toutes les lettres,
  - tous les chiffres,
  - le caractère « blanc » (déconseillé),
  - le caractère « _ » (« underscore » ≠ « moins ») ;
- Le premier caractère doit être une lettre ;
- *Seuls les 6 premiers caractères sont significatifs* : `epsilon1` et `epsilon2` représentent la même variable ;
- (Rappel) pas de différence minuscules majuscules.

4.3 Règles de typage implicite

On peut ne pas déclarer une variable (fortement déconseillé), en utilisant les règles suivantes :

Une variable dont le nom commence par `i,j,k,l,m,n` est automatiquement de type `integer`.
Une variable commençant par toute autre lettre (de `a` à `h` et de `o` à `z`) est automatiquement de type `real`.

4.3.1 Directive `implicit`

Elle permet modifier ces règles par défaut *de manière locale* à un bloc fonctionnel.

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

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

Cette directive doit être écrite juste après | `program`
`subroutine` .
`function`

4.3.2 Exemples

```
      implicit real (a-c,e,w-z)
```

Tout ce qui commence par `a,b,c,e,w,x,y,z` sera `real`

```
      implicit double precision (a-h,o-z)
```

Similaire à la règle par défaut : tout ce qui commence par `i,j,k,l,m,n` sera `integer`, tout le reste `double precision` (*Très Utilisé !!*).

```
      implicit complex (z)
```

Tout ce qui commence par z est `complex` par défaut.

```
      implicit none
```

Aucune variable n'est utilisable si elle n'est pas déclarée.

Important : il s'agit de règles applicables aux variables *non déclarées*. La déclaration d'une variable l'emporte sur les règles implicites.

4.4 Affectation d'une variable

4.4.1 Syntaxe

nomvar = *constante*	Ex : `x=1.23`
nomvar = *autre variable*	Ex : `x=y`
nomvar = *opération*	Ex : `x=y+3.2*z`

4.4.2 Fonctionnement

– Lit la valeur de toutes les variables à droite du signe `=` ;
– Effectue les opérations demandées ;
– Affecte le résultat à la variable à gauche de `=`.

4.4.3 Exemple

```
i = i + 1
```

Augmente le contenu de la variable `i` de 1 (on dit aussi « *incrémenter* » une variable).

Chapitre 5

Opérateurs et fonctions mathématiques

5.1 Opérateurs arithmétiques

Le tableau 5.1 donne la liste des opérateurs arithmétiques de Fortran. Ils sont listés par ordre de priorité croissante.

Dans une expression, on évalue donc d'abord `**` puis `/`, puis `*`, et enfin `+` et `-`.

On peut aussi grouper des sous-expressions entre parenthèses.

5.1.1 Exemples

```
x=a+b/c-d
```

évalue $a + \frac{b}{c} - d$.

```
x=(a+b)/(c+d)
```

évalue $\frac{a+b}{c+d}$.

TAB. 5.1 – Les opérateurs arithmétiques.

Addition	`+`
Soustraction	`-`
Multiplication	`*`
Division	`/`
Puissance	`**`

5.2 Conversion de type dans une opération

Lorsque deux opérandes de types différents interviennent de chaque coté d'un opérateur :

1. l'opérande de type le plus faible est converti dans le type de l'autre opérande
2. l'opération est effectuée, et le type du résultat est le type le plus fort

Les types sont classés dans l'ordre suivant (du plus fort au plus faible) :
- `complex`
- `double precision`
- `real`
- `integer`
- `logical` [1]

5.2.1 Exemples

`b**2` sera du type de `b`.

`4*a*c` sera du type le plus fort de `a` et de `c`.

Si `a` et `c` sont `real`, et `b double precision`, le résultat de `b**2-4*a*c` sera `double precision`, mais attention, le produit `a*c` sera effectué en simple précision, d'où une perte de précision possible.

5.2.2 Pièges classiques

`2/3` sera du type `integer`. Autrement dit la division effectuée sera une division entière, et le résultat sera 0. Pour calculer effectivement deux tiers en réel, écrire : `2./3`, `2/3.` ou `2./3.`

Idem pour `i/j` où `i` et `j` sont deux variables `integer`. Écrire : `real(i)/j`

5.3 Conversion de type dans une affectation

Lorsqu'une variable est affectée avec une expression de type différent, le résultat de l'expression est converti dans le type de la variable.

5.3.1 Exemples

`i=1.3456` affectera la variable `integer i` avec 1.

Attention :

`x=2/3` affectera la variable `real x` avec 0.0 !!!

[1] `.true.` est converti en 1, et `.false.` en 0

TAB. 5.2 – Fonctions arithmétiques prédéfinies.

`real`	`double precision`	`complex`	Fonction
SIN	DSIN	CSIN	$sin(x)$
COS	DCOS	CCOS	$cos(x)$
TAN	DTAN		$tg(x)$
ASIN	DASIN		$arcsin(x)$
ACOS	DACOS		$arccos(x)$
ATAN	DATAN		$arctg(x)$
SINH	DSINH		$sh(x)$
...	...		...
LOG10	DLOG10		$log_{10}(x)$
LOG	DLOG	CLOG	$ln(x)$
EXP	DEXP	CEXP	$exp(x)$
SQRT	DSQRT		$\sqrt{x}$

5.4 Fonctions mathématiques

Une fonction FORTRAN est une boite dans laquelle rentre un ensemble de grandeurs d'un type donné (les *arguments*) et de laquelle sort une grandeur d'un type donné (cf 10.4).

Certaines fonctions mathématiques sont prédéfinies dans le langage (tables 5.2 et 5.3). On n'utilisera pas la même fonction selon le type de l'argument. Par exemple la fonction sinus sera `SIN` pour un argument réel, `DSIN` pour un argument double precision, `CSIN` pour un argument complexe.

Important :
- Le ou les arguments d'une fonction sont *toujours entre parenthèses*.
- Chaque fonction a son domaine de définition. Son non-respect entraîne une erreur d'exécution, c'est-à-dire que le programme s'arrête.

5.4.1 Exemples

```
      z=(a*sin(x)+b*cos(y)) / (a*sinh(x)+b*cosh(y))
```

Pour définir π en double precision :

```
      pi=4d0*datan(1d0)
```

5.4.2 Fonctions de conversion

Ces fonctions permettent de convertir explicitement des données d'un type en un autre type. La figure 5.1 donne les noms des des différentes fonctions permettant à partir d'un type initial d'obtenir un autre type de valeur.

TAB. 5.3 – Fonctions diverses prédéfinies.

`integer`	`real`	`double precision`	`complex`	Fonction
MAX0	AMAX1	DMAX1		$max(x, y)$
MIN0	AMIN1	DMIN1		$min(x, y)$
IABS	ABS	DABS	CABS	$\|x\|$ ou $\|z\|$
INT	AINT	DINT		Partie entière
MOD	AMOD	DMOD		Reste dans la division entière

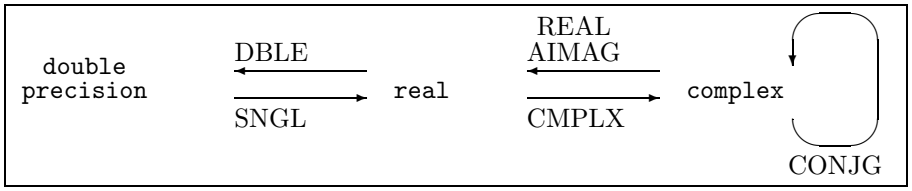


FIG. 5.1 – Fonctions de conversion de types.

5.5 Calcul en virgule flottante

La représentation des réels en virgule flottante entraîne fatalement des erreurs[2] :
- de représentation lors d'une affectation
  Elles viennent du fait qu'un nombre réel quelconque n'admet pas de représentation exacte en virgule flottante.
- d'arrondi dans les calculs.
  À la suite d'une opération arithmétique, le résultat est tronqué (c'est-à-dire que des chiffres significatifs sont effacés) pour pouvoir être codés en virgule flottante.

Ces erreurs peuvent être dramatiques, dans des algorithmes sensibles aux erreurs. Lors d'une opération simple, l'erreur effectuée est petite, mais si l'algorithme amplifie ces erreurs, le résultat peut être complètement faux.

5.5.1 Exemples

`1.+1e-8` sera codé comme 1.

`1d0+1d-16` sera codé comme 1d0

De nombreuses suites récurrentes exhibent les conséquences dramatiques des erreurs d'arrondi (voir TD).

Par exemple, la suite :

$$\begin{aligned} u_0 &= e - 1 \\ u_n &= n.u_{n-1} - 1 \end{aligned}$$

[2] Voir *La recherche* Juillet-Août 1995 p.772-777

converge mathématiquement vers 0 mais selon la machine utilisée et la précision choisie, elle pourra converger vers $-\infty$, ∞ ou 0 ou même ne pas converger du tout !

Chapitre 6

Manipulation de textes

Les textes sont stockés dans des chaînes de caractères. Dans ce chapitre, nous donnons quelques moyens de les manipuler.

6.1 Constantes chaînes

Elles sont constituées par une série de caractères encadrés par des apostrophes (ou « simple quotes » en anglais). Exemples :

```
'Ceci est une chaîne'
'/home/louisnar'
'L'apostrophe doit être doublé'
```

6.2 Variables chaînes

Syntaxe de déclaration :

```
character*n var
```

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

Exemples :

```
      character*15 nom
      character*100 nomfichier
```

On peut ensuite affecter ces variables avec l'opérateur = comme pour toute autre variable :

```
nomfichier='/usr/local/public/louisnar/th.mai'
nom='Louisnard'
```

Comme la chaîne `'Louisnard'` ne contient que 9 caractères, les 6 derniers caractères de `nom` sont affectés avec le caractère blanc. Si on affecte une variable chaîne de 15 caractères avec une chaîne de 16 caractères, il y aura une erreur d'exécution.

6.3 Fonctions sur les chaînes

6.3.1 Longueur d'une chaîne

`LEN(`*`chaine`*`)` renvoie la longueur en mémoire de `chaîne`.

Attention : avec l'exemple précédent, `len(nom)` renvoie 15 (et pas 9 !) puisque la variable `nom` a été définie comme une chaîne de 15 caractères.

6.3.2 Recherche dans une chaîne

`INDEX(`*`chaine1, chaine2`*`)` renvoie la position de la chaine `chaine2` dans la chaine `chaine1`.

Par exemple (toujours à partir de l'exemple précédent), `index(nom,'nar')` renvoie 6.

6.3.3 Sous-chaînes

`chaine(m :n)` est la sous-chaîne allant du $m^{ième}$ ou $n^{ième}$ caractère de `chaine`. `m` et `n` peuvent être des constantes ou des variables entières.

6.3.4 Concaténation

La concaténation permet de coller deux chaînes bout à bout. En FORTRAN, cet opérateur se note `//`.

Par exemple :

```
'Bon'//'jour'
```

représente la chaîne suivante :

```
'Bonjour'
```

Attention (toujours avec l'exemple précédent) :

```
nom //'est mon nom'
```

représente la chaîne suivante :

```
'Louisnard ␣␣␣␣␣␣ est ␣ mon ␣ nom'
```

Chapitre 7

Entrées / Sorties

On appelle « *Entrées / Sorties* », tout ce qui permet à un programme de dialoguer avec l'extérieur :
– l'utilisateur via le clavier, l'écran, une imprimante, etc. ;
– les disques via des fichiers ;
– d'autres machines via le réseau ;
– ...
Le langage FORTRAN permet d'écrire ou de lire des données sur différentes choses.

Pour écrire, on utilise l'instruction `write` :
– à l'écran ,
– sur un fichier,
– dans une chaîne de caractères.
Pour lire, on utilise l'instruction `read` :
– sur le clavier,
– dans un fichier.
– dans une chaîne de caractères.
On distingue les lectures/écritures :

formatées c'est à dire organisée en lignes de caractères.

C'est la cas des lectures clavier et des écritures écran, des lectures/écritures de fichiers texte et de chaînes.

non-formatées qui signifie transfert octet par octet.

C'est le cas des lectures/écritures sur fichiers binaires.

7.1 Écriture formatée

7.1.1 Syntaxe

`write` (*unité d'écriture*, *formatage*) *liste de données*

– L'*unité d'écriture* est un entier (voir « fichiers » 11).
  Pour l'écran, on utilise `*`.
– Le formatage indique *sous quelle forme* on va écrire les données.
  Il existe une formatage par défaut qui laisse FORTRAN écrire comme il veut : le format `*`.

7.1.2 Exemples

```
      write(*,*) i,j,x,y
```

écrit les valeurs des variables `i,j,x,y` sur une ligne, séparées par des blancs.

```
      write(*,*) 'z vaut',z
```

écrit la chaine « `z vaut` », suivie de la valeur de la variable `z`.

Important : se souvenir qu'une instruction `write` écrit une ligne puis revient à la ligne. Donc un `write` est égal à une ligne d'affichage.

7.2 Formats d'écriture

Un format est une série de codes, chaque code définissant le format d'écriture d'un élément d'une donnée.

7.2.1 Définition du format

Deux solutions :
– Directement dans l'instruction `write` avec une chaîne de caractères :

```
write (*,'format') liste
```

– Dans une ligne labellée contenant l'instruction `format` :

```
nn   format (définition du format)
     write (*,nn)
```

Cette solution permet d'utiliser le même format dans plusieurs instructions `write`.

7.2.2 Format entier

Dans un format, la notation `in` permet d'afficher un entier. L'entier est écrit sur n caractères en tout :

– S'il y a besoin de plus, l'écriture échoue.
– Si cela suffit, on ajoute éventuellement des blancs à gauche.

Exemples :

```
      i=11
      j=20312
      write(*,'(i6,i6)') i,j
```

donne

Variante avec l'instruction `format` :

```
10    format(i6,i6)
      i=11
      j=20312
      write(*,10) i,j
```

7.2.3 Format réel virgule flottante

Dans un format, la notation `en.m` permet d'afficher un réel en virgule flottante sur n caractères en tout avec m chiffres significatifs, c'est à dire :

$$\underbrace{\pm 0.\underbrace{\sqcup\sqcup\sqcup\ldots\sqcup}_{m\,caract.}E\pm\sqcup\sqcup}_{n\,caract.}$$

– S'il y a besoin de plus, l'écriture échoue.
– Si cela suffit, on ajoute éventuellement des blancs à gauche.

Pour que le format soit cohérent, il faut $n \geq m + 7$.

Exemples :

```
      x=-1456.2
      y=1.6e-19
      write(*,'(e14.5,e14.5)') x,y
```

affiche

$$\underbrace{\sqcup\sqcup -0.14562E+04}_{14\,caract.}$$
$$\underbrace{\sqcup\sqcup\sqcup 0.16000E-18}_{14\,caract.}$$

7.2.4 Format chaîne

Dans un format, la notation `an` permet d'afficher une chaîne de caractères.
- Si n est spécifié, la chaine est écrite sur n caractères en tout, en ajoutant des blancs à gauche pour compléter.
- Si n n'est pas spécifié, la chaine est écrite avec son nombre de caractères total (tel que déclaré !).

On peut ajouter un `$` après le `a` pour éviter de revenir à la ligne.

7.2.5 Exemples de formats mixtes

```
      i=36
      px=-1456.2
      write(*,'(a,i4,a,e12.5)')
     &  'i vaut', i, ' et x vaut', x
```

affichera

```
i vaut 36 et x vaut -0.14562E+04
```

7.2.6 En conclusion

Les formatages doivent être utilisés si c'est *absolument nécessaire*. Dans la plupart des cas le format par défaut (`*`)1 suffit largement, et il est inutile de perdre du temps a formater les sorties écran.

7.3 Lecture formatée

La lecture formatée s'applique | au clavier
aux fichiers texte

7.3.1 Principe

On lit une ligne de caractères d'un seul coup, la lecture étant validée par :
- la frappe de la touche RETURN pour une lecture clavier,
- une fin de ligne pour une lecture de fichier texte.

Les données sur une même ligne doivent être séparées par des blancs.

7.3.2 Syntaxe

```
read (unité de lecture, formatage) liste de variables
```

– L'unité de lecture est un entier (voir « fichiers » 11).
  Pour le clavier, on utilise `*`.
– Le formatage indique *sous quelle forme* on va lire les données (voir `write`).

7.3.3 Conseil

Le plus simple est d'utiliser tout le temps le format libre `*`.
Exception : pour lire des variables caractères ou chaînes de caractères, le format libre ne fonctionne pas. Utiliser le format chaîne `a`.

7.3.4 Exemple

```
      real a,b,c
      ...
      read(*,*) a,b,c
```

attend de l'utilisateur qu'il frappe trois réels au clavier séparés par des espaces puis la touche RETURN (↩). On peut entrer les nombres en format virgule flottante. Pour entrer $(1, 2)$ dans `a`, $(1, 6.10^{-19})$ dans `b` et (32) dans `c`, l'utilisateur pourra taper :

`1.2␣1.6e-19␣32`

Un exemple classique d'écriture suivie d'une lecture sur la même ligne :

```
      write(*,'(a,$)') 'Entrez x :'
      read(*,*) x
```

Le message sera affiché, mais le curseur ne reviendra à la ligne que lorsque l'utilisateur aura entré x suivi de RETURN.

Chapitre 8

Contrôle de l'exécution

Un programme enchaîne les instructions qu'on lui donne une à une dans l'ordre. Pour réaliser un *vrai* programme, il faut tout de même disposer de moyens pour faire des tests et des boucles : on appelle cela le contrôle d'exécution.

8.1 Instructions conditionnelles

8.1.1 Objectif

Exécuter une séquence d'instructions si une condition logique est vérifiée, sinon en exécuter une autre.

8.1.2 Syntaxes

Pour exécuter une série d'instructions uniquement si une *condition logique* est vraie :

```
if (condition logique) then
    ...
    ...
endif
```

On peut aussi spécifier une autre série d'instructions à exécuter si la *condition logique* est fausse :

```
if (condition logique) then
    ...
    ...
else
    ...
    ...
endif
```

On peut même enchaîner plusieurs conditions logiques :

```
if (condition logique 1) then
    ...
    ...
else if (condition logique 2) then
    ...
    ...
else if (condition logique 3) then
    ...
    ...
else
    ...
    ...
endif
```

Le programme exécute le bloc d'instructions suivant la première condition logique vraie puis reprend après `endif`. Si aucune condition logique n'est vraie, le bloc `else` est exécuté.

8.2 Expressions logiques

Ce sont des objets de type `logical`.

Ils ne peuvent prendre que deux valeurs | `.true.` `.false.`

Une expression logique est en général le résultat d'une comparaison entre deux objets :

En maths	En FORTRAN	En Anglais
$x = y$	`x.eq.y`	« equal »
$x \neq y$	`x.ne.y`	« not equal »
$x > y$	`x.gt.y`	« greater than »
$x < y$	`x.lt.y`	« less than »
$x \geq y$	`x.ge.y`	« greater or equal »
$x \leq y$	`x.le.y`	« less or equal »

On peut combiner ces expressions entre elles avec les opérateurs logiques usuels :

Ou	`.or.`
Et	`.and.`
Ou exclusif	`.xor.`
Négation	`.not.`

8.2.1 Exemples

Pour lire un caractère au clavier et agir en conséquence en tenant compte du fait que l'utilisateur peut répondre « Oui » en tapant `O` majuscule ou `o` minuscule, et idem pour non (avec `N` ou `n`) :

```
      character rep
      ...
      write(*,'(a,$)')
     &  'Répondez par (o)ui ou (n)on :'
      read(*,a) rep

      if (rep.eq.'O'.or.rep.eq.'o') then
        write(*,*) 'Vous répondîtes oui'
        ...
      else if (rep.eq.'N'.or.rep.eq.'n') then
        write(*,*) 'Vous répondîtes non'
        ...

      else
        write(*,*)
     &    'Vous répondîtes n■importe quoi'
        ...
      endif
```

Pour tester le signe du discriminant d'une équation du second degré :

```
      double precision a,b,c,delta,x1,x2
      ...
      delta=b**2-4*a*c
      x1=(-b-dsqrt(delta))/2/a
      x2=(-b+dsqrt(delta))/2/a

      if (delta.gt.0d0) then
        write(*,*) x1,x2

      else if (delta.eq.0d0) then
        write(*,*) x1

      else
        write(*,*) 'Pas de racines'
      endif
```

8.3 Variables `logical`

On peut déclarer des variables de ce type et leur affecter comme valeur :

– soit `.true.` ou `.false.`,

– soit le résultat d'une expression logique.

Cela sert parfois à améliorer la lisibilité des programmes.

Par exemple, pour l'équation du second degré :

```
      double precision a,b,c,delta,x1,x2
      logical une_racine
      logical deux_racines
      ...
      delta=b**2-4*a*c
      une_racine=(delta.eq.0d0)
      deux_racines=(delta.gt.0d0)

      if (deux_racines) then
        x1=(-b-dsqrt(delta))/2/a
        x2=(-b+dsqrt(delta))/2/a
        write(*,*) x1,x2
      else if (une_racine) then
        x1=-b/2/a
        write(*,*) x1
      else
        write(*,*) 'Pas de racines'
      endif
```

8.4 Boucles

8.4.1 Objectif

Une boucle permet d'exécuter une séquence d'instructions plusieurs fois d'affilée.

Le nombre de boucles peut être déterminé :
– à l'avance ,
– par le basculement d'une condition logique.

8.5 Boucles `do ... enddo`

On effectue la boucle un nombre de fois prédéterminé.

8.5.1 Syntaxes

```
do var = deb, fin
    ...
    ...
enddo
```

```
do var = deb, fin, pas
    ...
    ...
enddo
```

8.5.2 Fonctionnement

var est une variable de type `integer` et *deb*, *fin* et *pas* sont des objets de type `integer` (constantes ou variables).

La variable *var* prend d'abord la valeur de *deb* et est augmenté de *pas* à chaque boucle. Dès que $var > fin$, la boucle s'arrête et l'exécution continue après le `enddo`.

L'entier *pas* peut être omis et vaut 1 par défaut.

Si $fin < deb$ et $pas > 0$, la boucle n'est jamais exécutée.

Si $fin > deb$ et $pas < 0$, la boucle n'est jamais exécutée.

8.5.3 Exemples

Somme des premiers nombres entiers jusqu'à n :

```
      ...! affectation de n
      somme=0

      do i=1,n
        somme=somme+i
      enddo
```

ou bien encore :

```
      ...! affectation de n
      somme=0

      do i=n,1,-1
        somme=somme+i
      enddo
```

Somme des nombres impairs inférieurs à n :

```
      ...! affectation de n
      somme=0

      do i=1,n,2
        somme=somme+i
      enddo
```

8.6 Boucles `do ... while`

On effectue la boucle tant qu'une condition logique est vérifiée.

8.6.1 Syntaxe

```
do while (condition logique)
    ...
    ...
    ...
enddo
```

8.6.2 Fonctionnement

On rentre dans la boucle seulement si la condition logique vaut `.true.` et on exécute la boucle tant qu'elle reste à `.true.`.

Dès qu'à la fin d'une boucle, la condition est `.false.`, l'exécution reprend après `enddo`.

8.6.3 Remarque importante

Pour pouvoir sortir de la boucle, il faut que la condition logique puisse devenir `.false.` à l'intérieur. Si ce n'est pas le cas, le programme ne s'arrêtera jamais (Pensez-y !).

8.6.4 Exemples

Sommation de la série $\sum_{n\geq 1} 1/n^2$ jusqu'à ce que le terme général soit inférieur à ϵ fois la somme partielle courante :

```
      integer n
      double precision somme, epsilon
      ... ! affectation de epsilon
      n=1
      somme=0
      do while (1d0/n**2 .ge. epsilon*somme)
        somme=somme + 1d0/n**2
        n=n+1
      enddo
```

Plus élégant, en utilisant une variable `logical` :

```
integer n
double precision somme, epsilon
logical fini

...! affectation de epsilon
n=1
somme=0
fini=.false
do while (.not. fini)
  somme=somme + 1d0/n**2
  n=n+1
  fini=(1d0/n**2 .lt. epsilon*somme)
enddo
```

8.7 Instructions `goto` et `continue`

Il s'agit d'un archaïsme. À proscrire absolument, sauf si on ne peut pas faire autrement. Tout abus sera puni !

Les instructions de branchement conduisent à des programmes illisibles et difficiles à corriger. Certaines instructions du FORTRAN (voir « fichiers » 11) utilisent des branchements de manière implicite pour gérer des erreurs. Dans ce cas, et seulement dans celui-ci, le branchement est obligatoire.

8.7.1 Syntaxe

```
goto N° de label
```

En arrivant sur une telle ligne, le programme est branché directement sur la ligne comportant le label mentionné. En général, pour faire beau (il faut le dire vite...), cette ligne contient seulement l'instruction qui ne fait rien `continue`.

8.7.2 Exemple

Pour vous donner des mauvaises idées :

```
      character rep
      ...
10    continue
      write(*,*) 'Repondez oui ou non'
      read(*,*) rep

      if (rep.ne.'o'.and.rep.ne.'n') then
        goto 10
      endif
      ...
```

Chapitre 9

Les tableaux

A partir des types simples du FORTRAN, on peut former des vecteurs, des matrices, et même des tableaux à plusieurs indices.

9.1 Déclaration

```
type var(m1, m2, ...)
```

`m1`, *m2*,... déterminent la taille du tableau. Elles doivent être des *des constantes entières*. Il est interdit de mettre des variables entières. Autrement dit : la taille d'un tableau FORTRAN est fixée une fois pour toutes.

9.1.1 Exemples

```
      real v(100)
      double precision a(100,100)
      integer i(20)
```

9.1.2 Premier conseil

Pour créer des tableaux, il est conseillé de déclarer les tailles dans des constantes symboliques :

```
      parameter (max=100)
      double precision a(max,max)
      real v(max)
```

9.1.3 Second conseil

Avant de déclarer un tableau, pensez à la taille mémoire qu'il va occuper. Le tableau `a` ci-dessus occupe par exemple $100 \times 100 \times 8 = 80000$ octets.

9.2 Utilisation des tableaux

On y accède élément par élément en indiquant le ou les indices entre parenthèses séparés par des virgules. Les indices peuvent être des constantes ou des variables.

9.2.1 Exemples

Somme de deux matrices :

```
      double precision a(max,max)
      double precision b(max,max)
      double precision c(max,max)
      ...
      do i=1,max
        do j=1,max
          c(i,j)=a(i,j)+b(i,j)
        enddo
      enddo
```

Produit scalaire de deux vecteurs :

```
      double precision u(max), v(max)
      double precision prodsca
      ...
      prodsca=0
      do i=1,max
        prodsca=prodsca + u(i)*v(i)
      enddo
```

9.3 Instructions `read` et `write` avec boucles implicites

C'est une extension très pratique pour lire des matrices au clavier ou les écrire à l'écran. Il s'agit en quelque sorte d'une boucle `do` ... `enddo` combinée a un `read` ou un `write`.

9.3.1 Syntaxe

```
read(*,*) (var(i), i = i1, i2, i3)
write(*,*) (var(i), i = i1, i2, i3)
```

i représente une variable entière, *var(i)* une expression ou un tableau dépendant de cette variable *i*. *i1*, *i2* et *i3* ont le même sens que pour les boucles `do`.

9.3.2 Exemples

Lire les n premières composantes d'un vecteur sur une même ligne au clavier :

```
      read(*,*) (v(i), i=1,n)
```

Écrire les n premiers termes de la $j^{\text{ème}}$ ligne d'une matrice à l'écran, séparés par le caractère « ! » :

```
      write(*,*) (a(i,j), ' ! ', j=1,n)
```

9.4 Utilisation optimale des tableaux

9.4.1 Problématique

La taille de déclaration des tableaux définit la taille mémoire réservée pour stocker le tableau. En général, on choisit cette taille comme étant la taille maximale du problème que l'on veut traiter, mais il se peut que pour un problème particulier, on utilise seulement une partie du tableau.

Dans l'exemple de l'addition, supposons que nous utilisions les tableaux FORTRAN `a(100,100)` et `b(100,100)` pour stocker des matrices 3×3. La structure de `a` sera la suivante :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

et de même pour `b`. Pour faire la somme des deux matrices, il est inutile d'additionner les `0`, et les boucles doivent être effectuées de 1 à 3 plutôt que de 1 à 100.

Lorsqu'on utilise un tableau FORTRAN, il faut stocker ses dimensions réelles dans des variables en plus de sa taille de déclaration.

Dans l'exemple de l'addition précédent, on utilisera une variable `n` si on somme des matrices carrées, ou deux variables `nligne` et `ncolon` si on somme des matrices rectangulaires.

9.4.2 Exemple

Écrire un programme complet qui lit au clavier deux matrices de même taille $m \times n$ et effectue leur somme. On part du principe que ces matrices ont au plus 50 lignes et 80 colonnes.

On déclare donc 3 tableaux 50×80, et on utilise deux variables `nligne` et `ncolon` pour stocker les tailles réelles des matrices que l'on traite. Ces tailles réelles sont bien sûr demandées à l'utilisateur.

```
      parameter (mligne=50, mcolon=80)

      double precision a(mligne, mcolon)
      double precision b(mligne, mcolon)
      double precision c(mligne, mcolon)

      integer nligne, ncolon

      write(*,*)
     &  'Nombre de lignes des matrices'
      read(*,*) nligne
      write(*,*)
     &  'Nombre de colonnes des matrices'
      read(*,*) ncolon

      write(*,*) 'Matrice a '
      do i=1,nligne
        read(*,*) (a(i,j), j=1,ncolon)
      enddo

      write(*,*) 'Matrice b '
      do i=1,nligne
        read(*,*) (b(i,j), j=1,ncolon)
      enddo

      do i=1,nligne
        do j=1,ncolon
          (i,j)=a(i,j)+b(i,j)
        enddo
      enddo

      end
```

Chapitre 10

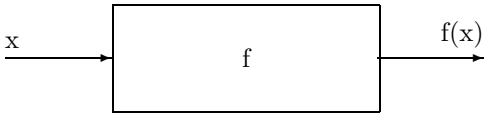
Fonctions et subroutines

10.1 Premier objectif

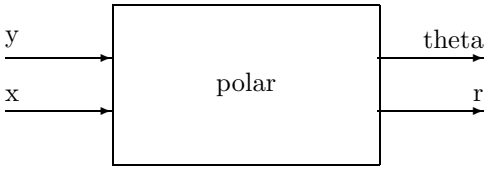
Il arrive fréquemment que l'on doive faire plusieurs fois la même chose au sein d'un même programme, mais dans un contexte différent. Par exemple :

– Saisir des matrices au clavier :

– Calculer la fonction $f(x) = \arcsin(\log x/x)$ pour plusieurs valeurs de x :



– Calculer les coordonnées polaires (r, θ) d'un point défini par un couple de réels (x, y) :

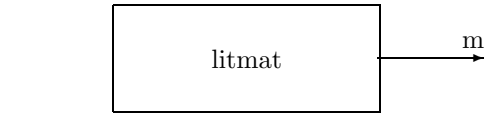


10.2 Second objectif

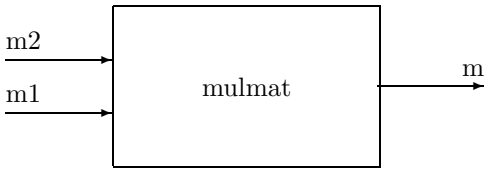
On a un problème décomposable en plusieurs sous problèmes. On cherche à « *enfermer* » chaque sous problème dans un bloc et à faire communiquer ces blocs.

Exemple : écrire un programme qui lit trois matrices au clavier, en fait le produit, puis affiche le résultat. On écrira trois blocs de base :

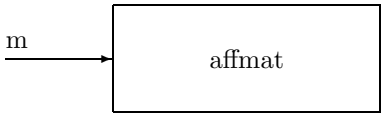
– un qui permet de lire une matrice au clavier :



– un qui effectue la somme de deux matrices :



– un qui affiche une matrice à l'écran :



Les objets FORTRAN correspondants à ces blocs sont les *subroutines* ou les *fonctions*.

On voit que chacun de ces blocs peut être écrit séparément, et qu'il est relié à l'extérieur par des « portes » d'entrée/sortie repérées par un nom. Persuadons-nous que ce nom n'a de sens que pour le bloc, et qu'il est là pour identifier une entrée ou une sortie.

Les connexions des boites avec l'extérieur sont appelés en FORTRAN « *paramètres formels* », et seront traités comme des variables dans les instructions exécutables.

Avant d'aller plus loin, montrons comment utiliser les trois blocs définis ci-dessus pour résoudre le problème proposé.

– On va utiliser le bloc LITMAT 3 fois, et lui faire cracher 3 matrices a, b, c.
– On va utiliser MULMAT pour faire le produit de a et b, et mettre le résultat dans une variable $p1$
– on va réutiliser MULMAT pour faire le produit de $p1$ par c et mettre le résultat dans $p2$.
– on va utiliser AFFMAT pour afficher $p2$.

Symboliquement cela revient à connecter les blocs comme le montre la figure 10.1.

10.3 Les subroutines

Une subroutine est une séquence d'instructions appelable d'un point quelconque du programme. Elle peut être appelée depuis le programme principal ou depuis une autre subroutine ou fonction.

Une subroutine est définie par :
– un nom

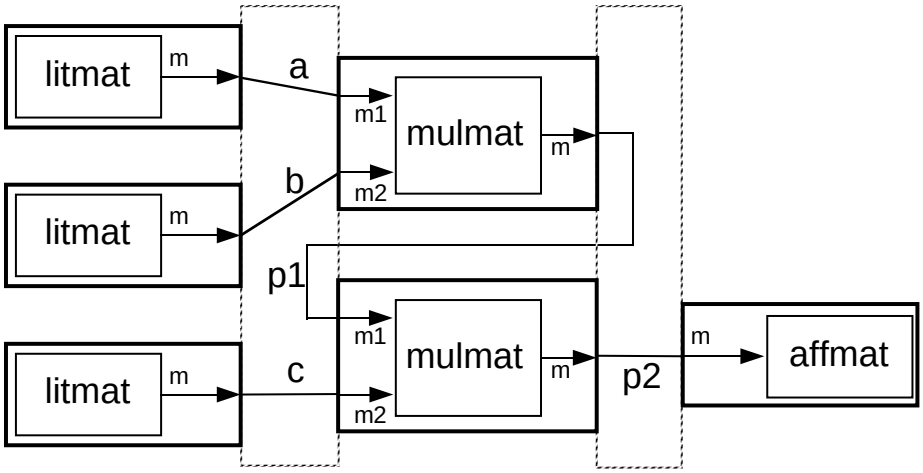


FIG. 10.1 – Connexion de blocs fonctionnels pour réaliser un programme.

– des paramètres formels, qui ont comme les variables :
  – un nom
  – un type

10.3.1 Écriture d'une subroutine

```
subroutine nomsub(pf1, pf2, pf3, ...)
type pf1
type pf2
type pf2
...
Déclaration des variables locales
...
Instructions exécutables
...
return
end
```

Les instructions de la subroutine peuvent manipuler :
– les paramètres formels (comme des variables normales),
– les variables locales,
– les variables d'un common.
Les instructions de la subroutine ne peuvent pas manipuler les variables locales du programme principal ou d'une autre subroutine ou fonction.

10.3.2 Appel d'une subroutine

L'appel d'une subroutine se fait depuis un bloc fonctionnel quelconque (programme principal, subroutine, fonction) avec l'instruction `call`.

```
call nomsub (v1, v2, v3, ...)
```

Les arguments *v1*, *v2*, *v3* peuvent être :
– des variables du bloc fonctionnel,
– des constantes (déconseillé !).

10.3.3 Très important

Les types des arguments *v1*, *v2*, *v3*,... doivent correspondre exactement à ceux des paramètres formels *pf1*, *pf2*, *pf3*,...

10.3.4 Exemple

Écrire une subroutine qui calcule les coordonnées polaires associées à des coordonnées cartésiennes (x, y).

$$\begin{array}{ll} r = \sqrt{x^2 + y^2} & \\ \theta = \arctan |y/x| & x > 0 \\ \theta = -\arctan |y/x| & x < 0 \\ \theta = \pi/2 & x = 0 \text{ et } y > 0 \\ \theta = -\pi/2 & x = 0 \text{ et } y < 0 \end{array}$$

Le figure 10.2 donne un code source possible de cette subroutine.

Un exemple d'appel de cette subroutine dans le programme principal :

```
      programm
      ...
      double precision a, b, rho, phi
      ...
      call polar (a, b, rho, phi)
      ...
      end
```

10.3.5 Remarques très importantes

– Les variables `a`, `b`, `rho` et `phi` sont des variables locales du programme principal : elles n'ont aucun sens pour la subroutine `polar`.
– Les variables `pi` et `temp` sont des variables locales de la subroutine `polar` : elles n'ont aucun sens pour le programme principal.
– `x`, `y`, `r` et `theta` sont les paramètres formels de la subroutine `polar`. Ce sont les portes de communication de la subroutine et leurs noms n'ont aucun sens à l'extérieur de la subroutine.

```
      subroutine polar(x, y, r, theta)

      double precision x, y, r, theta

      double precision pi
      double precision temp

      pi=4*datan(1d0)

      r=dsqrt(x**2+y**2)

      temp=datan(y/x)
      if (x.gt.0d0) then
        theta=temp
      else if (x.lt.0d0) then
        theta=-temp
      else if (x.eq.0d0) then
        if (y.ge.0d0) then
          theta=pi/2
        else
          theta=-pi/2
        endif
      endif

      return
      end
```

FIG. 10.2 – Code source d'une subroutine de calcul de coordonnées polaires à partir de coordonnées cartésiennes.

En particulier, s'il y a dans le programme principal des variables locales appelées `x`, `y`, `r` ou `theta`, *elles n'ont rien à voir avec les paramètres formels de la subroutine.*

En revanche, il est possible de passer ces variables locales en tant qu'arguments d'appel à `polar` :

```
      call polar (x, y, rho, phi)
```

La variable locale `x` du programme principal est alors passée à la subroutine via son premier paramètre formel, qui incidemment s'appelle aussi `x`, mais les deux objets sont bien distincts.

10.3.6 Exercice 1

Réécrire le programme de résolution des équations du second degré avec des subroutines.

10.3.7 Exercice 2

Dans le programme de la figure 10.3, il y a trois erreurs ô combien classiques. Corrigez-les, puis répondez ensuite aux questions suivantes :

- Quelles sont les variables locales
  - du programme principal ?
  - de la subroutine ?
- Pourquoi n'a-t-on pas utilisé une variable `integer` pour coder $n!$?
- Pourquoi ne pas utiliser la subroutine factor pour calculer les C_n^p ?

Complétez ensuite les subroutines pour qu'elles calculent effectivement $n!$ et C_n^p.

10.4 Les fonctions

Une fonction est en tout point identique à une subroutine, mais son nom contient en plus une valeur. C'est-à-dire qu'au lieu de l'appeler par `call`, on l'utilise à droite du signe `=` dans une instruction d'affectation.

On avait déjà vu les fonctions prédéfinies du FORTRAN, par exemple `datan`, qui n'a qu'un paramètre formel :

```
      pi=4*datan(1d0)
```

```
      program factcnp

      do i=1,100
        call factor(i, facti, ifaux)
        write(*,*) i, facti
      enddo

      write(*,*)
     &  'Entrez deux entiers i et j'
      read(*,*) i, j

      call calcnp (i, j, cij)
      end

      subroutine factor(n, fact, ierr)

      integer n, ierr
      double precision fact
      ...
      return
      end

      subroutine calcnp (n, p, cnp, ierr)
      implicit double precision (a-h,o-z)
      integer n, p, cnp
      ...
      return
      end
```

FIG. 10.3 – Programme de calcul de $n!$ et de C_n^p (à compléter).

10.4.1 Valeur de retour

Puisque le nom de la fonction contient une valeur, cette valeur doit être typée. Par exemple `datan` renvoie une valeur `double precision`. En plus du type de ses paramètres formels, la définition d'une fonction doit donc décrire le type de la valeur qu'elle retourne.

La valeur de retour de la fonction sera affectée dans le corps de la fonction, comme si le nom de la fonction était une variable ordinaire.

10.4.2 Écriture d'une fonction

```
type function nomfonc (pf1, pf2, ...)
type pdf1
type pdf2
...
déclarations des variables locale
...
instructions exécutables
! devrait contenir quelque chose comme : nomfonc = ...
...
return
end
```

Le type peut être omis auquel cas le type de la fonction est fixé par les règles par défaut.

10.4.3 Utilisation d'une fonction

Une fonction est utilisable dans tout autre bloc fonctionnel, comme une variable qui aurait des arguments. Comme pour les subroutines, *il est indispensable de respecter la correspondance entre les arguments passés à la fonction et ses paramètres formels.*

Attention : il faut non seulement déclarer le type de la fonction lors de sa définition, mais aussi dans tous les blocs fonctionnels où on l'utilise. Si cette déclaration est absente, les règles de typage automatiques du bloc fonctionnel courant s'appliquent.

10.4.4 Exemple 1

Fonction qui calcule le rayon-vecteur $r = \sqrt{x^2 + y^2}$ associé à un couple (x,y) : figure 10.4.

10.4.5 Exemple 2

Fonction qui saisit un caractère au clavier : figure 10.5.

```
      program test

      double precision abscis, ordonn, r
      double precision rayon

      read(*,*) abscis, ordonn
      r=rayon(abscis, ordonn)
      write(*,*) r
      end

      double precision function rayon (x, y)

      double precision x, y

      rayon=dsqrt(x**2+y**2)

      return
      end
```

FIG. 10.4 – Calcul de $r = \sqrt{x^2 + y^2}$ par une fonction.

```
      program test
      character c, litcar

      do while (litcar() .ne. 'q')
        write(*,*) 'On continue'
      enddo
      end

      character function litcar()
      read(*,*) litcar
      return
      end
```

FIG. 10.5 – Fonction de lecture d'un caractère au clavier.

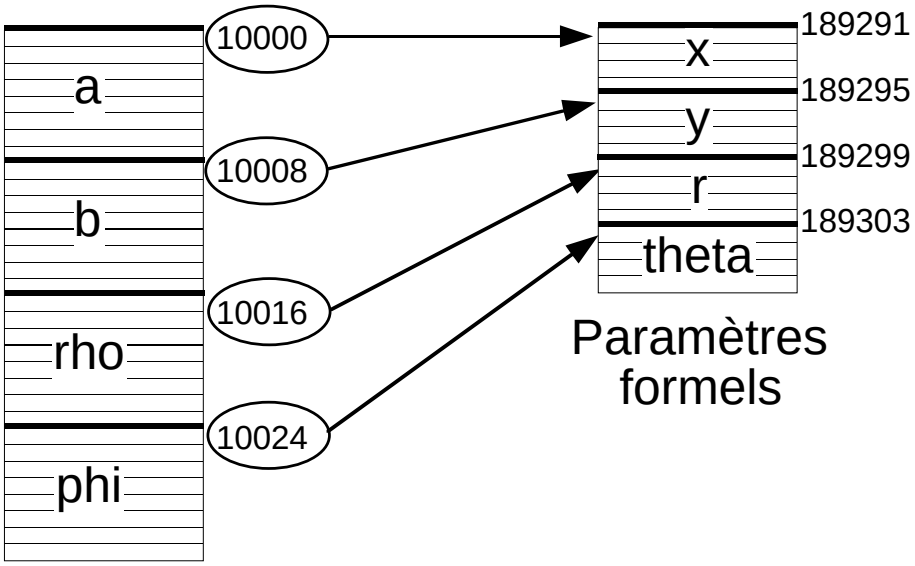


FIG. 10.6 – Passage des paramètres dans l'exemple de la fonction `polar`.

10.5 Mécanisme de passage des arguments à une subroutine ou une fonction

Les variables locales des divers blocs fonctionnels sont stockées à une adresse mémoire fixée par le compilateur. Cette adresse mémoire est un grand entier, qui est « le numéro de bureau » de la variable.

Certains bureaux sont plus grands que d'autres. Une variable `integer` ou `real` occupera un bureau à 4 cases, une variable `double precision` un bureau à 8 cases, un vecteur de 100 `real`, un bureau de 4×100 cases, une matrice de 50×50 `double precision` un bureau de $50 \times 50 \times 8$ cases.

Mais dans tous les cas, l'adresse de la variable est l'adresse de la première de ces cases.

A un paramètre formel de subroutine ou de fonction est associé en mémoire un nombre de cases suffisant pour stocker une adresse (4 octets sous UNIX). Lors d'un appel à une subroutine ou fonction, *l'adresse du premier argument* est écrite à l'emplacement réservé au premier paramètre formel, idem pour le second, le troisième, etc.

La figure 10.6 illustre ceci dans l'exemple de la subroutine `polar`.

Lorsque dans le corps de la subroutine, le programme rencontrera le paramètre formel `x` de type double precision à droite de `=`, il ira lire 8 octets à partir de l'adresse contenue dans `x` (de 10000 à 10007), c'est-à-dire la valeur de `a`.

Lorsqu'il rencontrera une affectation du paramètre formel `theta`, il ira écrire les 8 octets à partir de l'adresse contenue dans `theta` (de 10024 à 10031), c'est-à-dire `phi`.

D'où l'importance de respecter le nombre et le type d'arguments.

10.6 L'instruction `common`

On a vu que par défaut les variables d'un bloc fonctionnel lui étaient locales, donc inconnues des autres blocs. Il existe un moyen d'étendre la portée d'une variable à plusieurs blocs fonctionnels : le `common`.

Un `common` comporte un nom et une liste de variables. Les variables de cette liste seront connues dans tous les blocs fonctionnels où l'on écrit le `common`.

10.6.1 Syntaxe

```
common/nomcom/v1, v2, v3...
```

10.6.2 Remarques

- Le `common` ne dispense pas des déclarations.
- On ne peut mettre des constantes déclarées par `parameter` dans les `commons` (en particulier les tailles de tableaux)
- On ne peut mettre la même variable dans deux `commons` différents.
- On ne peut mettre un paramètre formel dans un `common`.
- On peut passer des tableaux en `common`, à condition de les déclarer de la même longueur partout.
- Tout ce que vérifie le compilateur, c'est que la longueur totale en octets de chaque `common` est la même dans tous les blocs fonctionnels où il apparaît. On a donc le droit de changer le nom des variables entre deux utilisations d'un même `common`, mais cela est déconseillé.

10.6.3 Exemples

Dans l'exemple de la figure 10.7, on voit que les noms des variables du `common bidon` sont différentes dans le programme principal et dans truc.

Mais cela est correct puisqu'il y a 2 `reals` de chaque coté. `u` et `a` représentent exactement le même objet car ils correspondent à la même zone mémoire. Cela dit, il vaut mieux garder les mêmes noms partout.

10.7 Paramètres formels de type tableau

On peut passer des tableaux à des subroutines ou fonctions si celles-ci sont conçues pour les recevoir. Comment déclare-t-on des paramètres formels de type tableaux ? Cela dépend du nombre d'indices.

```
      program test

      double precision pi
      real a, b

      common /trig/ pi
      common /bidon/ a, b

      pi=4*datan(1d0)
      ...
      end

      subroutine truc (x,y)

      common /trig/ pi
      common /bidon/ u, v

      double precision pi
      real u, v
      ...
      y=x*tan(pi*u/v)
      ...
      return
      end
```

FIG. 10.7 – Exemple d'utilisation de l'instruction `common`.

vecteur `v` (1 indice) : on n'a pas besoin de la taille de déclaration du vecteur qui arrivera par le paramètre `v` :

```
subroutine sub (v, ...)
type v(*)
```

matrice `m` (2 indices) : il faut la première taille de déclaration de la matrice et donc il faut prévoir un paramètre formel pour cette constante :

```
subroutine sub (a, mligna, ...)
type a (mligna, *)
```

Ces déclarations s'appliquent uniquement aux paramètres formels, qui rappelons-le ne sont que des portes de communication pour les subroutines et les fonctions. En aucun cas une variable ne pourra être déclarée avec une `*`. Une variable de type tableau (rappel) doit toujours être déclarée avec des constantes entières

10.7.1 Exercice

Au vu du mécanisme de passage des arguments aux subroutines, expliquez pourquoi les paramètres formels de type vecteurs et matrices sont déclarés de cette manière.

10.7.2 Exemple

Subroutine qui lit une matrice au clavier. La subroutine devra sortir la matrice, son nombre de lignes réelles, son nombre de colonnes réelles (figure 10.8).

Il faut bien comprendre que la seule matrice ayant une existence réelle est la matrice `mat` du programme principal, et que pour lui réserver de la mémoire, il faut la déclarer avec un nombre de lignes et un nombres de colonnes explicites.

Le paramètre formel `a` de la subroutine va recevoir l'adresse de `mat` au moment de l'appel, et connaissant sa première taille de déclaration (10) via le paramètre formel `ma`, elle sera à même de lire et écrire un élément quelconque de `mat`.

10.7.3 Exercice 1

Que se passe-t-il si l'utilisateur tape un nombre de lignes supérieur à 10 et/ou un nombre de colonnes supérieur à 20 ?

10.7.4 Exercice 2

Écrire une subroutine qui affiche une matrice à l'écran, et combinez-la avec `litmat` pour vérifier votre réponse à la question précédente.

```
      program test
      parameter(mligne=10, mcolon=20)
      double precision mat (mligne, mcolon)

      call litmat (mat, mligne, nligne, ncolon)

      end

      subroutine litmat (a, ma, nl, nc)

      double precision a (ma, *)

      write(*,*)
     &  'Entrez nombre de lignes-colonnes'
      read(*,*) nl, nc

      do i=1, nl
        write(*,*) 'Ligne ', i
        read(*,*) (a(i,j), j=1,nc)
      enddo

      return
      end
```

FIG. 10.8 – Lecture d'une matrice au clavier.

10.8 Déclaration `external`

10.8.1 Objectif

Utiliser le nom d'une fonction ou d'une subroutine comme argument d'une autre fonction ou subroutine.

Quelle drôle d'idée ?

Examinons le problème suivant : Écrire une subroutine qui calcule l'intégrale $\int_a^b f(x)\,dx$ pour une fonction f *quelconque*. Que faudra-t-il faire entrer et sortir de la subroutine ?

- **la fonction** f ;
- les bornes d'intégration a et b ;
- la valeur de l'intégrale.

Or quel est le moyen en FORTRAN de programmer une fonction $f(x)$? C'est d'utiliser une fonction FORTRAN, qui renverra par exemple une valeur `real`.

On peut donc prédire la forme de la subroutine d'intégration :

```
      subroutine integ (a, b, f, valint)
```

Les paramètres formels `a`, `b` et `valint` seront `double precision`, mais de quel type est `f` ?

C'est le nom d'une fonction FORTRAN, et pour déclarer un paramètre formel aussi bizarre, on écrira :

```
      external f
```

et aussi

```
      real f
```

car `f` renvoie une valeur `real`. On peut aussi déclarer des paramètres formels de type subroutine, en utilisant simplement `external`. Dans le corps de la subroutine, on fera bien sûr appel à cette fonction `f` pour calculer l'intégrale. Pas de difficulté ! On fait comme si elle existait et on écrira des choses du style :

```
      valint = valint + h/2 *(f(x1+h) + f(x1))
```

Maintenant ma subroutine d'intégration est écrite. Je souhaite l'appliquer à une fonction que j'ai écrite en FORTRAN, du style :

```
      real function truc(x)
      real x
      ...
      truc=...
      return
      end
```

Je veux appeler la subroutine `integ` pour calculer l'intégrale de cette fonction entre disons 1 et 2. J'écrirai :

```
      call integ (1.0, 2.0, truc, somtruc)
```

`1.0` et `2.0` sont des constantes `real`, `somtruc` une variable `real` qui me renverra la valeur de l'intégrale...

Et `truc` ? C'est le nom d'une fonction FORTRAN. Ai-je le droit de passer ce genre de chose en argument à une subroutine ? La réponse est oui, si je la déclare :

```
      external truc
```

dans le bloc fonctionnel d'où je fais le `call`. De plus, comme cette fonction renvoie une valeur `real`, j'ajouterai :

```
      real truc
```

10.8.2 Récapitulation

La figure 10.9 récapitule tout ce que l'on vient d'expliquer.

Remarquons que la structure de `truc` est imposée par la subroutine qui demande que la fonction représentée par son paramètre formel `f` ait un argument réel et renvoie une valeur réelle. Nous ne pourrions donc pas par exemple déclarer :

```
      real function truc(x,y)
```

En général, les subroutines du type `integ` sont des boites noires toutes faites (par des spécialistes), et on vous indique juste le type de ses paramètres formels. Lorsque l'un d'entre eux est une fonction ou une subroutine, on vous indique en plus la liste des paramètres formels que doit avoir cette fonction ou subroutine.

10.8.3 Exercice

Écrire la structure d'un programme (programme principal / subroutine / fonctions) pour trouver les zéros d'une fonction $f(x)$ par la méthode de Newton. On rappelle que cette méthode nécessite la connaissance de la fonction $f(x)$ et de sa dérivée $f'(x)$.

```
      program test

      real somtruc

      external truc
      real truc

      call integ (1.0, 2.0, truc, somtruc)

      end


      subroutine integ (a, b, f, valint)

      real a, b, valint
      external f
      real f
      ...
      valint=valint + ( f(x1+h) + f(x1) )*h/2
      ...
      return
      end


      real function truc(x)
      real x
      ...
      truc=...
      return
      end
```

FIG. 10.9 – Structure générale d'un programme d'intégration simple.

Chapitre 11

Les fichiers

Avertissement : nous parlons ici implicitement de fichiers dits *séquentiels*, c'est-à-dire qu'à partir de l'ouverture du fichier, on lit les données dans l'ordre où elles sont stockées, sans pouvoir accéder directement à une donnée particulière. Au fur et à mesure des lectures ou écritures, un *pointeur de fichier* avance automatiquement d'une donnée à la suivante.

Il existe en FORTRAN des fichiers dits *à accès direct*, mais nous n'en parlerons pas ici.

Les fichiers séquentiels sont divisés en deux types : le fichiers formatés et les fichiers binaires.

11.1 Les fichiers formatés

Plus simplement, ce sont des fichiers texte, c'est-à-dire organisés en lignes de caractères, que l'on pourrait lire ou écrire avec un éditeur.

Lorsque l'on écrit un objet FORTRAN sur ce type de fichier, celui-ci est converti en chaîne de caractère selon un formatage défini par l'utilisateur ou par défaut (voir « Écritures formatées » 7.1)

Ce type de fichier est pratique car on peut les visualiser par un éditeur de texte, et de plus presque tous les logiciels savent lire des fichiers texte.

L'écran et le clavier sont des cas particuliers de fichiers formatés.

11.2 Les fichiers binaires

Les objets FORTRAN sont écrits sur ce type de fichier tels qu'ils sont stockés en mémoire. Par exemple, pour écrire un réel, on écrit directement les 4 octets constituant son codage en virgule flottante.

De ce fait, ce type de lecture/écriture est plus rapide et engendre des fichiers plus petits. L'inconvénient est que ces fichiers ne sont pas consultables par un éditeur de texte.

11.3 Ouverture d'un fichier

Il s'agit d'ouvrir un fichier déjà existant ou d'en créer un nouveau. Le FORTRAN associe au nom du fichier un entier appelé « *unité* », auquel toutes les instructions FORTRAN de manipulation des fichiers se réfèreront. Autrement dit le fichier n'est appelé par son nom qu'au moment de l'ouverture.

11.3.1 Syntaxe

```
open(  numéro d'unité,
       file=chaîne de caractère,
       form=chaîne de caractères,
       status=chaîne de caractères,
       err=numéro de label
```

Lors de l'ouverture d'un fichier, le pointeur de fichier est automatiquement placé avant la première donnée, sauf s'il est ouvert avec `status='append'`.

11.3.2 Description

Numéro d'unité : tout entier compris entre 10 et 99. On peut également mettre une variable `integer` contenant cet entier.

`file=` suivi d'une constante chaîne de caractère indiquant le nom du fichier en clair (par exemple `'/home/louisnar/truc'`) ou bien une variable de type chaîne contenant le nom du fichier.

`form=` chaîne de caractère pouvant être :
- `'formatted'` ouvre ou crée un fichier formaté. C'est le défaut.
- `'unformatted'` ouvre ou crée un fichier binaire.

`status=` chaîne de caractère pouvant être :
- `'new'` crée un nouveau fichier, ou génère une erreur s'il existe.
- `'old'` ouvre un ancien fichier, ou génère une erreur s'il n'existe pas.
- `'unknown'` ouvre le fichier quoi qu'il arrive. C'est le défaut.
- `'append'` ouvre le fichier et se place automatiquement à la fin de celui-ci.

`err=` numéro de label vers lequel le programme sera dérouté en cas d'erreur à l'ouverture.

11.3.3 Exemples

```
      open(10,file='ethanol')
```

ouvre le fichier formaté `ethanol` du répertoire courant, qu'il existe ou non, et l'attache à l'unité 10.

```
      integer unite
      character*80 nomfich
      ...
      unite=10
      nomfich='ethanol'
      open(unite,file=nomfich)
```

fait la même chose que l'exemple précédent.

```
      character*80 nomfich

      nomfich='bidon.dat'

      open(10,file=nomfich,form='unformatted',
     &  status='new',err=99)
      ...! (l'ouverture a réussi)
99    write(*,*) 'Fichier ', nomfich,
     &  'deja existant'
      ...
```

essaye de créer un nouveau fichier binaire `bidon.dat` et affiche un message d'erreur au cas où ce fichier existe déjà.

11.4 Fermeture d'un fichier

C'est simple :

```
close(numéro d'unité)
```

Cette instruction détache le numéro d'unité du nom du fichier. Pour tout nouvel accès au fichier, il faut l'ouvrir à nouveau avec l'instruction `open`.

Notons que si le programme se termine normalement, tous les fichiers sont automatiquement fermés.

11.5 Lecture / Écriture sur un fichier

11.5.1 Syntaxe

```
read(  numéro d'unité,
       format,
       err=numéro de label,
       end=numéro de label) liste de données

write(  numéro d'unité,
        format,
        err=numéro de label) liste de données
```

– Pour un fichier non formaté, on n'indique pas de format.
– Pour un fichier formaté, on indique un format.
  Pour les lectures/écritures formatées, le format `*` est le format par défaut, les autres formats s'utilisent comme pour les écritures écran, lectures clavier (voir « formats d'écriture » 7.2).

Les autres paramètres optionnels sont :

`err=numéro de label` En cas d'erreur, le programme est dérouté vers ce label.

`end=numéro de label` Utilisable en lecture : lorsqu'on arrive à la fin du fichier, autrement dit quand il n'y a plus rien à lire, le programme est dérouté vers le label mentionné.

Remarque importante : Lorsque l'on effectue une instruction `write` dans un fichier séquentiel, *toutes les données suivant la position courante du pointeur de fichier sont effacées.*

Ainsi si l'on ouvre un fichier existant avec comme `status 'old'` ou `'unknown'`, et que l'on fait tout de suite un `write`, toutes les données seront effacées. Aussi, si l'on veut ajouter des données dans un fichier déjà existant, il faut l'ouvrir avec `status='append'`.

11.6 Exemples d'utilisation de fichiers

11.6.1 Affichage d'un fichier texte

Ouvrir le fichier texte `truc.txt` dont on ne connaît pas à priori le nombre de lignes et afficher à l'écran toutes les lignes de ce fichier texte (comme la commande `cat` UNIX).

C'est un cas exceptionnel ou on ne peut se passer de `goto` :

```
      character*80 ligne

      open(20,file='truc.txt')

10    continue
      read(20,'(a)', end=99) ligne
      write(*,*) ligne
      goto 10

99    continue
      end
```

11.6.2 Stockage de données (x, y)

Écrire une subroutine recevant en entrée deux vecteurs x et y `double precision` contenant n données chacun, et une chaîne de 80 caractères contenant un nom de fichier, qui écrit sur un fichier texte une valeur de x et une valeur de y par ligne.

```
      subroutine ecritxy (x, y, nomfich)
      character*80 nomfich
      double precision x(*), y(*)

      open(20,file=nomfich)

      do i=1,n
        write(20,*) sngl(x(i)), sngl(y(i))
      enddo

      close(20)
      return
      end
```

Pourquoi convertit on `x(i)` et `y(i)` en `real` avec la fonction `sngl` ? Tout simplement parce que lorsque le FORTRAN écrit une grandeur double precision avec le format `*`, celui-ci écrit un `D` à la place du `E`, par exemple : `1.345367222D-5` au lieu de `1.345367222E-5`

Malheureusement la plupart des logiciels susceptibles de relire ce fichier (par exemple MATLAB) comprennent cette notation avec `E` mais pas avec `D`.

11.7 Exercice

On suppose que tout corps pur est déterminé par trois grandeurs que l'on appellera P_c, T_c et ω. Pour les besoins d'un programme, on veut se concocter une petite base de données, et pour chaque corps pur, à l'aide d'un éditeur on écrit en clair ces 3 valeurs sur 3 lignes d'un fichier texte. Par exemple, le fichier `eau` contiendra :

```
2
373
5e7
```

Écrire une subroutine recevant en entrée une variable chaîne contenant un nom de fichier, et fournissant en sortie :
– les trois valeurs de P_c, T_c et ω lues sur le fichier,
– un code d'erreur valant 0 si la lecture a réussi, -1 sinon.

Chapitre 12

La compilation et la correction des erreurs

12.1 Le compilateur FORTRAN SUN

Le compilateur de Format s'appelle `f77` (c'est général sous UNIX). Compilation d'un programme contenu dans `truc.f` :

```
f77 truc.f
```

Le fichier exécutable se nomme alors `a.out`. Si vous préférez appeler votre exécutable `chose`, utilisez l'option `-o` :

```
f77 truc.f -o chose
```

Il existe de nombreuses options de `f77`. En voici deux très utiles :

```
f77 -g truc.f -o chose
```

est indispensable si vous voulez par la suite débugger votre programme. L'omission de l'option `-g` rendra impossible l'utilisation du debugger. Utilisez-la systématiquement lorsque vous mettez au point votre programme.

```
f77 -O truc.f -o chose
```

optimise votre programme, qui tournera plus vite. L'option `-O` est incompatible avec l'option `-g`.

Il est possible de couper un programme FORTRAN en plusieurs fichiers sources : il suffit que l'un des fichiers sources contienne le programme principal et on peut ensuite répartir l'ensemble des subroutines et fonctions dans des fichiers différents.

Pour compiler tous ces fichiers en même temps, on indique leurs noms à la queue leu leu après `f77`. *Indiquez le fichier contenant le programme principal en tête de liste !*

Exemple :

```
f77 truc.f subs.f machin.f -o chose
```

Si vous examinez votre répertoire après avoir compilé un ou plusieurs fichiers FORTRAN, vous y trouverez des fichiers finissant par `.o`. Il s'agit de fichiers dits *objets*.

Le fichier `truc.o` contiendra le code machine correspondant au fichier source `truc.f`, mais les appels aux subroutines extérieures à `truc.o` sont laissées en suspens.

Un *éditeur de liens* appelé `ld` fait alors le lien entre tous les fichiers `.o` pour engendrer l'exécutable correspondant. Dans la ligne de compilation précédente, les deux phases sont effectuées successivement de manière transparente pour l'utilisateur :
– `truc.f`, `subs.f`, et `machin.f` sont compilés séparément par `f77` en `truc.o`, `subs.o` et `machin.o`
– `truc.o`, `subs.o` et `machin.o` sont liés par `ld` pour créer le programme `chose`.
On peut imposer cette décomposition soi-même en utilisant l'option `-c` pour chaque fichier source :

```
f77 -c truc.f
f77 -c subs.f
f77 -c machin.f
f77 truc.o subs.o machin.o -o chose
```

Ceci peut être intéressant lorsque l'on compile de gros programmes (plusieurs dizaines de fichiers sources) pour ne pas avoir à recompiler les fichiers sources qui n'ont pas été modifiés. Un utilitaire très puissant appelé « `make` » permet de gérer simplement ce procédé.

12.2 Utilisation de librairies

Une librairie est une collection de fichiers objets `.o` archivés dans un même fichier dont le nom commence généralement par `lib` et finit toujours par `.a`. La librairie NAG est par exemple archivée dans `libnag.a`.

Les librairies sont placées à des endroits de l'arborescence de fichiers que vous n'êtes pas obligés de connaître et auxquels vous avez accès si votre compte est configuré pour cela. Pour lier la librairie `libxxx.a` à votre programme :

```
NAME
      f77 - Sun FORTRAN compiler

SYNOPSIS
      f77  [-C] [-c] [-g] [-lx] [-o output] [-u] sourcefile ...

OPTIONS
 -c   Compilation sans édition de lien; Permet de créer les
      différents fichiers objets (.o) des fichiers .f qui
      figurent en argument.

      ex: f77 -c test.f     ou       f77 -c *.f

 -C   Permet lors de la compilation de tester si l'on dépasse de
      manière explicite la dimension d'un vecteur ou d'un tableau.

 -g   Génère les fichiers objets et exécutables associés à une
      table de symboles  permettant d'utiliser le débugger dbx.

      ex: f77 -g *.f
          dbx a.out

 -lx  Permet l'édition de liens avec les fichiers objets d'une
      bibliothèque de sous-programmes (libx.a).

      ex: f77 -c *.f
          f77 *.o -lnag  (pour la bibliothèque NAG)

 -o output

      Permet de donner un autre nom (ici output) au fichier
      exécutable.

      ex: f77 *.f -o exec

 -u   Génère une erreur avec un message pour toute les variables
      non déclarées (équivalent à placer l'instruction
      "IMPLICIT NONE" en tête de tous les programmes)
```

FIG. 12.1 – Extrait traduit de la « manual page » de f77.

```
f77 truc.f subs.f machin.f -o chose -lxxx
```

La figure 12.1 présente un extrait (traduit en français) des « manual pages » UNIX de la commande `f77`. Il existe beaucoup plus d'options que celles présentées ici. Le lecteur intéressé pourra se reporter à la documentation FORTRAN SUN §1.3 ou bien consulter les pages manuelles complètes par la commande :

```
man f77
```

12.3 Quelques erreurs de compilation

```
"try.f", line 276: Error: bad dimension list for array "dfdx"
Compilation failed
```

Survient :
- lorsqu'on utilise un vecteur ou un tableau (ici `dfdx`) qui n'est pas dimensionné
- lorsqu'on dimensionne autre chose qu'un paramètre formel avec `*` ou avec une variable.

```
"try.f", line 277: Error: unbalanced parentheses, statement skipped
"try.f", line 277: Error: unclassifiable statement
```

Problème de parenthèses. Attention le problème peut provenir d'une ligne en amont de la ligne indiquée.

```
"try.f", line 160: Warning: incompatible lengths for common block zone01
```

Un common n'a pas la même taille dans les différents sous-programme où il est utilisé. Bien que ce problème n'empêche pas la compilation (Warning), il entraînera généralement une erreur d'exécution.

```
"try.f", line 160: Error: declaration among executables
```

Une instruction de déclaration est placée au milieu de la partie exécutable du programme.

```
"try.f", line 183: Error: do not closed
"try.f", line 187: Error: nested loops with variable "i"
"try.f", line 188: Error: do loops not properly nested
```

Il manque une instruction ENDDO pour fermer une boucle.

```
"try.f", line 5: Error: unbalanced quotes; closing quote supplied
"try.f", line 5: Error: unclassifiable statement
```

Il manque une quote (') pour fermer une chaine de caractères. Cela arrive souvent lorsqu'une instruction est écrite à partir de la $6^{\text{ème}}$ colonne ou bien lorsque vous dépassez la $72^{\text{ème}}$ colonne. Cette erreur peut aussi arriver lorsqu'on oublie le caractère de déclaration de commentaire en première colonne.

En général, ce dernier oubli a des conséquences catastrophiques. La preuve :

```
"try.f", line 1: Error: illegal continuation card ignored
"try.f", line 65: Error: "vect" is a non-argument adjustable array
"try.f", line 65: Error: "nvar":  Adjustable array's bounds must be a dummy
argument or in a common block
"try.f", line 79: Error: bad dimension list for array "vect"
"try.f", line 86: Warning: RETURN statement in main or block data
"try.f", line 104: Warning: RETURN statement in main or block data
"try.f", line 120: Warning: RETURN statement in main or block data
"try.f", line 231: Error: external name "MAIN"
```

```
ld: Undefined symbol
   _sub1_
   _sub2_
```

Erreur lors de l'édition de liens :
- Ou bien vous appelez quelque part dans votre programme des subroutines ou fonctions `sub1` et `sub2` qui n'existent dans aucun des fichiers source mentionnés. Cela arrive souvent lorsque l'on oublie le lien avec une librairie.
- Ou bien vous utilisez des tableaux `sub1` et `sub2` non dimensionnés.

Lorsque l'un des symboles « `undefined` » est `_MAIN_`, c'est qu'il n'y a pas de programme principal. Cela se produit si vous compilez un fichier ne contenant que des subroutines sans l'option `-c`.

12.4 Quelques erreurs d'exécution

```
*** Segmentation Violation = signal 11 code 3
Traceback has been recorded in file:
         /home1/dr/pcf/letourne/ALGO/TEST/./a.out.trace
Note: Line numbers for system and library calls may be incorrect
IOT trap
```

Survient lorsqu'on dépasse la dimension d'un vecteur ou un tableau.

```
*** Bus Error = signal 10 code 2
Traceback has been recorded in file:
         /home1/dr/pcf/letourne/ALGO/TEST/./a.out.trace
Note: Line numbers for system and library calls may be incorrect
IOT trap
```

Très fréquent. Par exemple :

- Un vecteur passé à une subroutine ou fonction n'est pas dimensionné dans le bloc fonctionnel appelant.
- Une instruction WRITE contient trop ou pas assez de données à écrire par rapport au format utilisé.

```
F(X)=NaN
Note: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared; see ieee_flags(3M):
Invalid Operand;
Note: IEEE NaNs were written to ASCII strings or output files;
see econvert(3).
Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

`NaN` signifie « Not a Number ». Se produit lorsque les arguments d'une fonction ne sont pas dans son domaine de définition (exemple racine d'un nombre négatif) Remarque : Il est possible que l'exécution continue en affichant la valeur `NaN` là où l'on attendait voir s'afficher des résultats par l'intermédiaire d'une instruction WRITE.

```
Inf
Note: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared; see ieee_flags(3M):
Inexact;  Overflow;
Note: IEEE Infinities were written to ASCII strings or output files;
see econvert(3).
```

Même principe que ci-dessus. `Inf` signifie « Infinity ». Se produit lorsqu'une opération donne un nombre trop grand pour être codé dans un type donné.

```
*** Illegal  = signal 4 code 2
Traceback has been recorded in file:
         /home1/dr/pcf/letourne/ALGO/TEST/./a.out.trace
Note: Line numbers for system and library calls may be incorrect
IOT trap
```

Survient lorsqu'on oublie de déclarer EXTERNAL une fonction ou une subroutine passée en argument à une fonction ou subroutine.

```
list io: [112] incomprehensible list input
logical unit 5, named 'stdin'
lately: reading sequential list external IO
part of last format: ext list io
IOT trap
```

Le type de la donnée lue au clavier ou sur un fichier est incompatible avec le format de lecture spécifié dans l'instruction READ.

```
list read: $[-1]$ end of file
logical unit 5, named 'stdin'
lately: reading sequential list external IO
part of last format: ext list io
part of last data: .0001J$|$
IOT trap
```

On essaye de lire après la fin d'un fichier.

```
F(X)=*************
```

Erreur de format ; la valeur à écrire ne peut l'être avec le formattage spécifié dans l'instruction `write`.

Ces deux listes d'erreurs ne sont pas exhaustives. Il est vraisemblable que vous rencontriez beaucoup d'autres erreurs. Dans ce cas, et afin d'affiner ces listes, prévenez-nous ou envoyez-nous le programme pour que nous les complétions.

Bonne chance !!