Fortran 95

École normale supérieure
L3 sciences de la planète Terre
2011/2012

Lionel GUEZ
Version du 22 novembre 2012

MCours.com

Cours inspiré de :
Lignelet, 1996, Fortran 90 et Fortran 95

Table des matières

Table des matières (suite)

Introduction

Introduction

- Fortran : *Formula translator*
- Un des plus anciens langages de programmation encore utilisés : créé par IBM en 1954
- Objectif : calcul scientifique

Historique

- 1954 Fortran, première spécification du langage, premier compilateur de Fortran
- 1957 Fortran II
- 1958 Fortran III non publié
- 1962 Fortran IV
  Normalisation en 1966 par organisme américain : ANSI (*American National Standards Institute*), version normalisée devient connue sous le nom « Fortran 66 »

Historique (suite)

- 1978 Fortran 77
  Passage à une normalisation internationale : ISO (*International Standards Organization*)
- 1991 Fortran 90 (révision majeure)
- 1997 Fortran 95 (révision mineure)
  Compilateurs gratuits : g95 (depuis 2006 mais développement arrêté en 2008), gfortran depuis version 4.2.4 (2008), ifort (Intel, pour utilisation non commerciale)

Historique (suite)

- 2004 Fortran 2003 (révision majeure)
  Compilateur IBM XL Fortran depuis version 13.1, sortie en 2010
  Pas encore toutes les fonctionnalités dans autres compilateurs
- 2010 Fortran 2008 (révision mineure)

Historique (suite)

- Normalisation → stabilité, évolution lente (Tous les langages de programmation ne sont pas normalisés. Exemples : Python, Perl…)
- Alternance de révisions majeures et mineures depuis Fortran 90

Les conséquences de l'historique

- Importante accumulation de programmes et de bibliothèques de sous-programmes depuis naissance de Fortran.
- Doivent rester utilisables. → Les archaïsmes du langage doivent rester licites. Les normes successives suppriment très peu et ajoutent beaucoup.

Les conséquences de l'historique (suite)

- Fortran devient de plus en plus gros. Plusieurs façons licites de programmer pour arriver à un même résultat.
- Façons de programmer à juger du point de vue de : la concision, la sécurité, la clarté, la performance. (Pas de conflit avec l'évolution des normes : les meilleures façons sont bien les plus modernes.)

Ce cours

- Fortran 95
- Une sélection et non une vue exhaustive
  - Sélection des fonctionnalités les plus utiles
  - Sélection d'une façon de programmer

Bibliographie

(Du plus sélectif au plus exhaustif, du guide au dictionnaire)

- Lignelet, Patrice
  Manuel complet du langage FORTRAN 90 et FORTRAN 95 - Calcul intensif et génie logiciel
  Publisher: Masson
  Date: 1996

Bibliographie (suite)

- Metcalf, Michael; Reid, John
  Fortran 90/95 Explained
  Publisher: Oxford University Press
  Date: 1999
  Edition: 2nd
- Adams, Jeanne C. et al.
  Fortran 95 Handbook - Complete ISO/ANSI Reference
  Publisher: MIT Press
  Date: 1997

Bibliographie (suite)

- Document de référence gratuit, norme Fortran 95 : http://j3-fortran.org/doc/standing/archive/007/97-007r2/pdf/97-007r2.pdf

B, A, BA

B, A, BA

- Caractères du langage : `ABC...Z` et `abc...z` équivalents, `0 1 … 9`, blanc souligné `_`, caractères spéciaux blanc `= + - * /`
  `( ) , . ' : ! “ % & ; < >`
- Identificateurs choisis par le programmeur : 1 à 31 caractères, lettres, chiffres, `_`, commençant par une lettre

B, A, BA (suite)

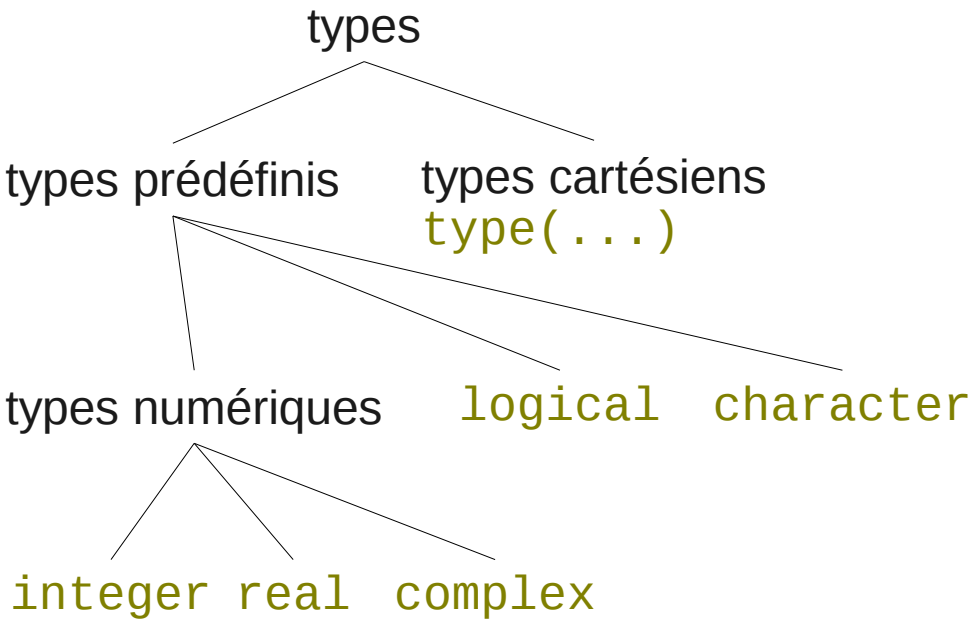
- Mots-clefs et identificateurs prédéfinis du langage : ne pas les utiliser pour autre chose. Cf. liste de mots clefs et liste des procédures prédéfinies (> 100).
- Instruction : 132 caractères maximum par ligne, & à la fin de la ligne pour continuer à la ligne suivante
- Commentaires introduits par !
  Seuls sur ligne ou à la suite d'une instruction

Structure globale d'un programme

- 1 « programme principal » et éventuellement des modules
- ```
  program nom
     [déclarations]
     instructions exécutables
  end program nom
  ```
- Arrêt possible avec :
  ```
  stop 1
  ```
  ```

Types, objets, déclarations

Types



- Typage automatique possible selon première lettre, le désactiver : `implicit none`

  à placer avant toute autre déclaration

Déclarations

- spécification de type[, liste d'attributs::] liste d'objets
- Exemples :

```
integer i, j
real, dimension(10, 3):: a, b, c
```

Opérations numériques

- Pour entiers (`integer`) et réels (`real`), opérateurs relationnels :
  `<`, `<=`, `==`, `/=`, `>=`, `>`
  Résultat logique
  Exemple :
  `x = i == 3`
- Opérations arithmétiques : `+ - * / **`

Sous-types numériques

- Ensembles de valeurs plus ou moins étendus (valeurs extrêmes, nombres de chiffres significatifs)
  Cf. programme `num_inquiry`.
- Existence d'un sous-type par défaut

```
real a ! sous-type par défaut
a = 3.
```

Choix d'un sous-type à la déclaration

```
integer(kind=...) a
real(kind=...) x
complex(kind=...) z
```

Valeurs possibles de `kind` : entières ≥ 1, dépendent du compilateur

```
x = 3._2 ! real(kind=2)
```

Exemple de sous-types numériques

```
Properties of integer kinds:
Default integer kind:           4
                        minimal      half-word           word    double word
         kind =               1              2              4              8
         huge =             127          32767     2.15E+09       9.22E+18
        range =               2              4              9             18
     bit_size =               8             16             32             64
        radix =               2              2              2              2
       digits =               7             15             31             63

Properties of real kinds:
                           default    double precision
      epsilon =          1.19E-07            2.22E-16
    precision =                 6                  15
        range =                37                 307
         huge =          3.40E+38           1.80+308
         tiny =          1.18E-38           2.23-308
        radix =                 2                   2
       digits =                24                  53
  minexponent =              -125               -1021
  maxexponent =               128                1024
```

Type entier

- Plus grande valeur du sous-type :
  `huge(i), huge(0)`
- Constantes littérales : sans point décimal
- `/` : division euclidienne
  Attention : `1 / 2` vaut 0
- Fonctions prédéfinies :
  `abs(a)` valeur absolue
  `mod(a, p)` identique à : `a – (a / p) * p`

Choix d'un sous-type entier

`selected_int_kind(r)`

pour un sous-type qui contienne au moins les entiers à `r` chiffres

Exemple :

```
integer, parameter:: wp &
  = selected_int_kind(9)
integer(wp) i
```

Type réel

- Plus petite valeur positive non nulle : `tiny(x)`
  Plus grande valeur : `huge(x)`
  Plus petit ε tel que 1 + ε > 1 : `epsilon(x)`
- Constantes littérales :
  Sans exposant : `-8.4`
  Avec exposant : `1e4`, `64.2e-5`
- Choix d'un sous-type réel :
  `selected_real_kind(p, r)`
  `p` : nombre de chiffres décimaux
  `r` : intervalle de valeurs de l'exposant (10^{-r} à 10^{r})

Réel double précision

- C'est un sous-type réel prédéfini
- Déclaration :
  `double precision x`
  équivalent à :
  `real(kind=`une certaine valeur qui dépend du compilateur`) x`
- Constantes littérales :
  `0d0 4.3d0 5d-3`

Type `character`

- Constantes littérales entre apostrophes ou guillemets :
  `'*' 'A' '9'  "'" '"' "x = ?"`
- Longueur d'une chaîne : nombre de caractères
  Déclaration :
  `character(len=5) a`
  `character b`
- Opérateur de concaténation : `//`

Type logique

- Constantes littérales :
  `.false. .true.`
- Opérateurs :
  `.not. .or. .and. .eqv. .neqv.`

Initialisation à la déclaration

- Obligatoire pour une constante symbolique :

```
real, parameter:: pi = 3.14159265,&
    e = 2.7182818
```

- Facultative pour une variable :

```
real:: x = 0, y
```

Affectations, expressions

Affectation

- variable réceptrice = expression
  Exemple : `n = n + 1`
- La variable et l'expression doivent être toutes deux de types numériques, ou toutes deux logiques, ou toutes deux chaînes de caractères

Affectation (suite)

- Des règles de priorité entre opérateurs (arithmétiques avant comparaison avant logique, `/` et `*` avant `+` et `-`, etc.)
  Parenthèses pour imposer un ordre
  Exemples :
  `a / b / c` identique à : `a / (b * c)`
  `x > y .or. i + j == k * l`

Conversions entre types numériques

- Conversion implicite lors de l'affectation
  Exemple :

```
integer i
real r
i = -1.9 ! reçoit -1 (pas d'arrondi)
r = 2 ! reçoit 2.
r = (2.3, 1.) ! reçoit 2.3
```

Conversions entre types numériques (suite)

- Conversion explicite :
  `int(a[, kind]) real(a[, kind])`
  `cmplx(x, [y, ][, kind])`
- Conversion vers le type « le plus riche » dans une expression
  Exemple : `1. / 2`

Sous-chaînes et affectation entre chaînes de caractères

- Sous-chaîne :
  `v(d:f) v(d:) v(:f)`
  Exemples :
  `character(len=8):: mot = "aversion"`
  `mot(2:5)` vaut `"vers"`
  `mot(6:6)` vaut `"i"`
  `mot(2:1)` vaut `""`
- Affectation : chaîne expression tronquée ou complétée avec des blancs si besoin

Affichage à l'écran et saisie au clavier

- `print *,` liste de valeurs
  `print * ! ligne vide`
- `read *,` liste de variables
  Chaîne de caractères : guillemets inutiles en général
  Valeurs peuvent être entrées sur plusieurs lignes
- Question et réponse sur la même ligne :

```
write(unit=*, fmt="(a)", &
    advance="no") "x = ?"
read *, x
```

L'alternative

Instruction simple conditionnée

if (expression logique) instruction simple
Exemple :

```
if (unew > umax) umax = unew
```

Alternative : construction `if`

```
if (expression logique 1) then
   séquence 1
[else if (expression logique 2) then
   séquence 2
…
else if (expression logique n) then
   séquence n]
[else
   séquence n + 1]
end if
```

Une séquence au plus est exécutée.

select case

- ```
  select case (c)
  case (s1)
     séquence 1
  case (s2)
     séquence 2
  …
  case (sn)
     séquence n
  [case default
     séquence n + 1]
  end select
  ```
  ```

`select case` (suite)

- `c` : expression entière ou chaîne de caractères
- `s1`, …, `sn` : valeur unique, ou intervalle de valeurs, ou liste de valeurs
  Valeurs du type de `c`
  Exemple : `case(1, 3, 7:10, 15:)`
  Les cas `s1`, …, `sn` doivent être mutuellement exclusifs.

L'itération

Boucle « pour »

- `do indice = a, b[, r]`
  `    séquence`
  `end do`
  `indice` : variable entière, ne pas la modifier à l'intérieur de la boucle
  `a`, `b`, `r` : expression entières, `r` ≠ 0, évaluées une seule fois à l'entrée de la boucle
- Par défaut : `r` vaut 1
- Le nombre d'itérations peut être nul

Boucle « pour » (suite)

- Valeur de l'indice après la boucle :
  `a` si aucune itération
  `b + 1` si itérations avec `r` = 1
  `b – 1` si itérations avec `r` = -1
- Pseudo-boucle dans les sorties :

```
print *, (liste d'expressions, &
    indice = a, b, r)
```

Boucles « tant que », « jusqu'à »

- Tant que :

```
do while (condition)
   séquence
end do
```

- Jusqu'à :

```
do
   séquence
   if (condition) exit
end do
```

Généralisation : boucle « n + ½ de Dijkstra »

```
do
   séquence 1
   if (condition) exit
   séquence 2
end do
```

Exemple : intérêt de la boucle n + ½

```
do
   print *, "x=?"
   read *, x
   if (correct) exit
   print *, "bad"
end do
```

```
print *, "x=?"
read *, x
do while (incorrect)
   print *, "bad"
   print *, "x=?"
   read *, x
end do
```

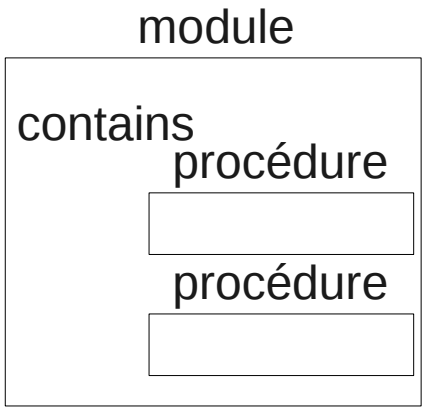
Procédures : sous-programmes et fonctions

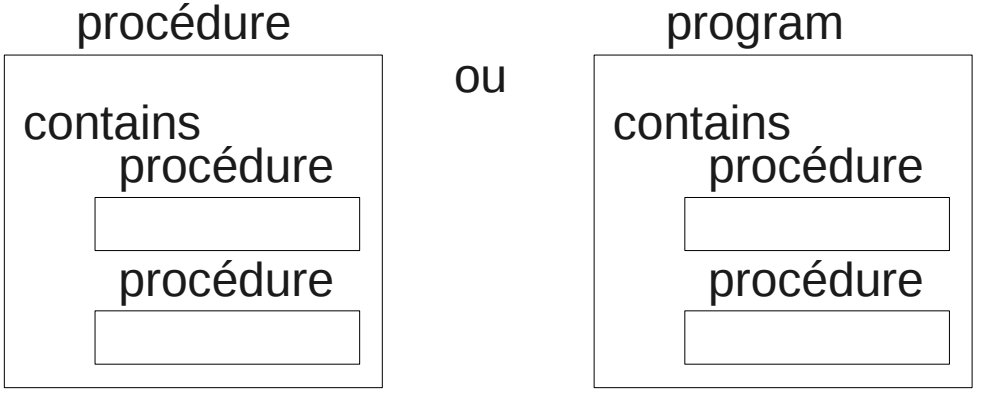
Sous-programmes et fonctions

- Procédure : `subroutine` ou `function`
- [préfixe] `subroutine`|`function &`
  identificateur[`(`liste d'arguments`)`]
  déclarations arguments et variables locales
  instructions exécutables
  `end subroutine`|`function` identificateur
- Préfixe : type de la fonction (ou `pure`, `elemental`, `recursive`)

Procédures de module et procédures internes

- Procédure de module :
  - Utilisable partout
  - Peut contenir une procédure interne
- Procédure interne :
  - Utilisable seulement par l'hôte (intérêt local)
  - Pas de sous-procédure interne
- Nombre quelconque de procédures par contenant





Utilisation d'une procédure de module : déclaration use

```
program my_program
use foo_m, only: foo
…
call foo(...)
```

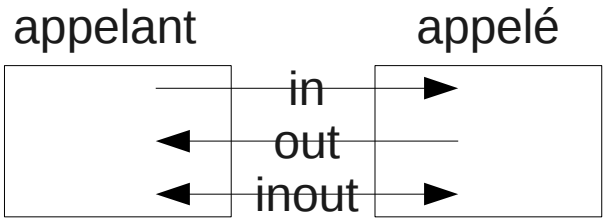
```
module foo_m
...
contains
subroutine foo(...)
use bar_m, only: bar
…
… = bar(...) + ...
```

- `use` nom de module`,` `only:` nom de procédure

```
module bar_m
...
contains
… function bar(...)
...
```

Les arguments et leurs trois modes de communication

- Données de la procédure :
  - attribut `intent(in)`
  - non modifiables
- Résultats de la procédure :
  - attribut `intent(out)`
  - valeur initiale indéterminée
- Données-résultats :
  - attribut `intent(inout)`
  - valeur initiale vient de l'argument effectif
  - modifiable



Exemple

```
real function foeew(t)
  real, intent(in):: t
  real, parameter:: r3les = 17.269
  real, parameter:: r4les = 35.86
  real, parameter:: rtt = 273.16
  !------------------
  foeew = exp(r3les * (t - rtt) / (t - r4les))
end function foeew
```

Correspondance arguments effectif et muet

- Même type, même sous-type (et même rang)
- Possibilité de mettre plusieurs fois le même argument effectif seulement si les arguments muets correspondants sont `intent(in)`

Cas particulier des chaînes de caractères

- Argument muet :
  `character(len=*), intent(...):: a`

  Hérite la longueur de l'argument effectif correspondant.
- Fonction prédéfinie len pour connaître cette longueur à l'intérieur de la procédure

```
if (len(a) == 1) then
    ...
```

Objets locaux et objets globaux

- Objets locaux inaccessibles à l'extérieur
- Objets de l'hôte accessibles
  Conseil : ne pas modifier les variables d'une procédure hôte depuis une procédure interne

Objets locaux et objets globaux (suite)

- Une déclaration locale masque un objet de même nom dans l'hôte

```
program
   integer i
   …
contains
   suboutine …
      integer i
   …
```

- Penser à re-déclarer les variables localement (`implicit none` ne vous protège pas)

Variables locales rémanentes

- Variable locale non rémanente : non définie au début de chaque exécution de la procédure
- Variable locale rémanente : conserve sa valeur d'une exécution de la procédure à l'exécution suivante
- Attribut `save`
  ```
  integer, save:: a
  ```
  ou initialisation :
  ```
  logical:: first_call = .true.
  ```

Variables chaînes automatiques

- Possibilité pour une variable locale chaîne de caractères d'avoir une longueur définie automatiquement à chaque appel, en fonction des arguments
- Exemple :

```
subroutine plouf(n)
   integer, intent(in):: n
   character(len=n) my_string
```

Appel d'une procédure

- `subroutine` :
  `call` identificateur`(` liste d'arguments effectifs`)`
- Fonction :
  identificateur`(` liste d'arguments effectifs`)`
  à l'intérieur d'une expression
  Exemple :
  ```
  print *, foeew(273.) * 100.
  ```

Appel par position ou mot-clef

- Correspondance argument effectif – argument muet par position :

```
function rtsafe(funcd, x1, x2, &
    xacc)
  real, intent(in):: x1, x2, xacc
  …
end function rtsafe
print *, rtsafe(my_func, 0., 1., &
  1e-4)
```

Appel par position ou mot-clef (suite)

- Ou par mots-clefs :

```
print *, rtsafe(funcd=my_func, &
  x1=0., x2=1., xacc=1e-4)
```

- Ou panachage :

```
print *, rtsafe(my_func, &
  xacc=1e-4, x1=0., x2=1.)
```

- Intérêt du mot-clef pour un argument effectif qui est une expression

Arguments optionnels

- Attribut `optional`
- Nécessité d'utiliser un appel à mot-clef pour sauter un argument effectif optionnel

Arguments optionnels : exemple

```
function spline(x, y, yp1, ypn)
  real, intent(in):: x(:), y(:)
  real, intent(in), optional:: yp1, ypn
  …
  if (present(yp1)) then
    …
  if (present(ypn)) then
    …
end function spline
print *, spline(x, y, ypn=ypn)
```

Arguments optionnels (suite)

- Tester la présence avec `present` avant toute référence sauf :
  - Argument de `present` lui-même
  - Argument effectif optionnel : absence ou présence transmise

```
 subroutine plouf(…, a, …)
real, optional, intent(…):: a
…
call foo(a)
…
```

```
subroutine foo(a, …)
real, optional, intent(…):: a
```

Tableaux

Définitions

- Scalaire (*scalar*) ≠ tableau (*array*)
- Rang (*rank*) d'un tableau : nombre de dimensions, entre 1 et 7
  Rang d'un scalaire : 0
- Vecteur : tableau de rang 1

Définitions (suite)

- Taille (*size*) d'un tableau dans une dimension donnée : nombre d'éléments
  Taille totale : produit des tailles dans toutes les dimensions
  Taille 0 autorisée
- Profil (*shape*) d'un tableau a :
  tableau de rang 1, type entier
  valeurs : taille a dim 1, taille a dim 2, …

Définitions (suite)

- Exemple :
  tableau réel `a(5, 7)`
  profil de `a` : `(/5, 7/)`
- Profil d'un scalaire :
  tableau de rang 1 de taille 0
- Tableaux compatibles (anglais : *conformable*) :
  même profil
- Scalaire et tableau toujours compatibles

Déclaration d'un tableau

- Obligatoire (scalaire par défaut)
- Attribut `dimension`
  Définit au moins le rang
  Éventuellement le profil

Déclaration du profil

- Variable locale (pas un argument)
  Taille connue au début de l'exécution
- Obligatoire pour une constante symbolique
- Exemple :
  ```
  real, dimension(10):: a, b, c
  ```
  ou :
  ```
  real a(10), b(10), c(10)
  ```
- Borne inférieure 1 par défaut
  Possibilité de borne inférieure quelconque
  ```
  real x(-2:3, 5)
  ```

Tableaux dynamiques

- Profil dépend de la suite de l'exécution
- La déclaration ne donne que le rang
- Attribut `allocatable`
  ```
  real, allocatable:: a(:), b(:, :)
  ```
- Détermination du profil par une instruction exécutable :
  ```
  allocate(a(-2: 3), b(5, 8))
  ```
- Pas d'utilisation possible avant `allocate`

Tableaux dynamiques (suite)

- Possibilité de dés-allouer (pour libérer de la mémoire ou avant de ré-allouer) : `deallocate(a, b)`
- Inutile de dés-allouer en fin de procédure pour un tableau dynamique non rémanent : dés-allocation automatique
- Possibilité de test : `allocated(a)` : résultat logique

Ordre des éléments de tableaux

- Certaines instructions font référence à un ordre
  Exemple :
  ```
  real a(2, 2)
  print *, a
  ```
- Ordre : premier indice varie le plus vite
  `a(1, 1)`, `a(2, 1)`, `a(1, 2)`, `a(2, 2)`
- Ordre correspond normalement à disposition en mémoire

Ordre des éléments de tableaux (suite)

- Pour un tableau de rang 2 : interprétation habituelle premier indice est l'indice de ligne → ordre colonne par colonne (interprétation pour `matmul`)
- Cas du parcours des éléments d'un tableau : Faire varier le premier indice dans la boucle la plus imbriquée →meilleure performance

Ordre des éléments de tableaux (suite)

- Exemple de parcours des éléments d'un tableau :

```
real a(long, lat, lev)
do k = 1, lev
  do j = 1, lat
    do i = 1, long
      ! traitement de a(i, j, k)
      …
    end do
  end do
end do
```

Constructeur de vecteur anonyme

- (/liste de composants/)
  Composant :
  - expression scalaire
  - Pseudo-boucle
  - tableau (linéarisé dans l'ordre de ses éléments)
- Tous les composants de mêmes type et sous-type

Constructeur de vecteur anonyme : exemples

```
character(len=3):: nom(2) = (/"to ", &
  "tou"/)
real:: v(3) = (/3.2, 4., 6.51/)
(/(3.1, (3.1 / i, i = 1, n)/)
(/2., v/)
character(len=3), parameter:: &
  currency(3) = (/"EUR", "FRA", "USD"/)
```

Tableaux de rang ≥ 2

- Construction possible avec la fonction intrinsèque `reshape` :
  `reshape(source, shape)`
  `source` : vecteur des éléments du tableau à construire, dans l'ordre
  `shape` : profil du tableau à construire
- Exemple :

```
reshape((/(i, i = 1, 6)/), &
   (/2, 3/))
```

donne $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

Accès à un élément de tableau

- Liste d'indices
  Nombre d'indices : rang du tableau
  Indice :expression entière
- Exemple :
  `t3(i * j, 1, k / i + 2)`

Sous-tableau

- Choix d'une progression arithmétique dans une ou plusieurs dimensions
  `a: b: r`
  `a` : premier indice, défaut borne inférieure
  `b` : dernier indice, défaut borne supérieure
  `r` : raison, défaut 1, éventuellement < 0
- Exemples :
  `t1(m: n)`, `t1(m + n: n: -1)`
  `mat(i, :)`, `mat(:i, j:)`

Sous-tableau (suite)

- Pour progression non arithmétique : indice vectoriel
  `integer:: index(4) = (/2, 1, 5/)`
  `t(index)` est `(/t(2), t(1), t(5)/)`
- Si l'indice vectoriel contient deux fois la même valeur : pas d'affectation au sous-tableau

Sous-objet d'un tableau de chaînes

- Possibilité de combiner sous-chaîne et sous-tableau
  Spécification du sous-tableau avant la sous-chaîne
- Exemples :
  `character(len=5) ch(2, 2), a(3)`
  `ch(1, 1)(i: j)` : chaîne scalaire
  `a(i: j)` : sous-tableau, chaînes complètes
  `a(:)(i: j)` : tableau entier, sous-chaînes
  (`(:)` nécessaire pour indiquer tableau entier)

Expression tableau

- Les opérateurs prédéfinis (arithmétiques, logiques, de comparaison, de concaténation) s'appliquent aux tableaux
- Les tableaux doivent être compatibles
- Résultat : tableau de même profil
- Rappel : compatibilité scalaire – tableau
- Exemples :
  `mat / 2`, `mat(:, 0) * t3(:, 1, 1)`
  `real a(5), b(5)`
  `a == b` : tableau logique

Affectation de tableaux

- Possibilité d'affecter à un tableau une expression tableau compatible
- Exemples :

```
a = a(10: 1: -1)
mat = 0
a(::2) = 0.
t1 = (/(i, i = 1, n)/)
t1(index) = t1(:4)
```

si `index` a pour taille 4 et ne contient pas deux fois la même valeur

Affectation conditionnelle : `where`

- Affectation à certains éléments d'un tableau
- Choix des éléments : masque, tableau logique
- `where (masque) tableau_cible = &`
  `    expression`
  masque, tableau cible et expression doivent être compatibles
- Exemples :
  `where (a < 0) a = 0` (tableau et scalaire)
  `where (mat /= 0) mat = 1 / mat`

Généralisation : construction `where`

```
where (masque)
    t_1 = e_1
    …
    t_n = e_n
[elsewhere
    u_1 = f_1
    …
    u_k = f_k]
end where
```

- Masque, tableaux, expressions compatibles
- Masque évalué une fois pour toutes
- Affectations en séquence, where puis elsewhere

Construction `where` : exemple

- ```
  where (pressure <= 1.)
      pressure = pressure+inc_pressure
      temp = temp + 5.
  elsewhere
      raining = .true.
  end where
  ```
- `pressure` modifié mais masque évalué une seule fois
  ```

Généralisation : `where` avec plusieurs masques

```
where (masque_1)
   …
elsewhere (masque_2)
   … ! masque_2 .and..not. masque_1
elsewhere (masque_3)
   … ! masque_3 .and..not. masque_2
   … ! .and. .not. masque_1
[elsewhere
   …]
end where
```

Généralisation : `where` avec plusieurs masques (suite)

- Tous masques, tableaux, expressions compatibles
- Exécution en séquence : évaluation de masque 1, séquence 1, évaluation de masque 2, séquence 2, …
- Chaque masque évalué une seule fois, non affecté par les séquences suivantes

`where` avec plusieurs masques : exemple

```
integer:: ia(3) = (/-3, 4, 1/)
where (ia > 0)
   ia = ia – 2 ! sur indices 2, 3
   ! {ia = (/-3, 2, -1/)}
elsewhere (ia(3: 1: -1) < 0)
   ia = 5 ! sur indice 1
   ! {ia = (/5, 2, -1/)}
elsewhere
   ia = 0 ! sur aucun indice
end where
```

`forall`

- C'est une forme d'affectation à une variable indicée : tableau ou chaîne de caractère
- Utile quand l'affectation ne peut pas s'écrire simplement avec des tableaux entiers, des sections de tableaux ou des tableaux masqués (avec `where`) : quand l'affection s'écrit plus clairement en faisant référence à des indices

`forall` (suite)

- Instruction `forall` :
  `forall (`indice `=` borne inférieure`:`borne supérieure[`:`pas]`)` variable`(`référence à indice`) =` ...
- Exemple, diagonale d'une matrice :

```
integer a(5, 5), i
forall (i = 1: 5) a(i, i) = 0
```

`forall` (suite)

- Le membre droit de l'affectation est évalué pour toutes les valeurs de l'indice avant toute affectation.
  Exemple :

```
integer b(10), i
forall (i=2:9) b(i)= &
    sum(b(i-1:i+1)) / 3
```

  Résultat différent de :

```
do i = 2, 9
    b(i) = sum(b(i-1:i+1)) / 3
end do
```

`forall` avec plusieurs indices

- `forall (i1=l1:s1[:p1], i2=l2:s2[:p2]... [,filtre]) affectation`
- Le domaine couvert par les indices doit être « rectangulaire » : pas de référence à un autre indice dans les bornes ou le pas
- La variable réceptrice dans l'affectation doit faire référence à chacun des indices
- Le filtre est une expression scalaire logique, qui peut faire référence aux indices : il restreint le domaine des affectations

`forall` avec plusieurs indices : exemples

```
forall (i=1:n, j=1:m) a(i,j) = i+j
forall (i=1:n, j=1:m, y(i,j)/=0.) &
   a(i,j) = i+j+1./y(i, j)
```

Triangle supérieur = transposé du triangle inférieur :

```
forall (i=1:n, j=1:n, j>i) &
   a(i,j)=a(j,i)
```

Généralisation : construction `forall`

- `forall (i1=l1:s1[:p1], &`
  `i2=l2:s2[:p2]... [,`filtre`])`

  corps

  `end forall`

  où le corps peut comprendre des affectations, d'autres `forall` emboîtés, ou des `where`
- Exécution en séquence de chaque instruction du corps

Exemples de construction `forall`

- ```
  forall (i = 1:n)
     where (a(i, :) == 0) a(i, :) = i
     b(i, :) = i / a(i, :)
  end forall
  ```
- Triangle supérieur = transposé du triangle inférieur :

  ```
  forall (i=1:n)
     forall (j=i+1:n) a(i,j)=a(j,i)
  end forall
  ```

  (la variable réceptrice doit faire référence aux deux indices)
  ```

`forall` : résumé et conseils

- `forall` n'est pas `do`. Ce n'est pas une structure de contrôle, c'est une forme d'affectation. On peut mettre n'importe quoi dans le corps de `do` (entrées ou sorties, appels de `subroutine`, ...), on peut mettre seulement des affectations dans le corps de `forall`.
- `forall` : membres droits évalués avant affectations

`forall` : résumé et conseils (suite)

- Tout `forall` pourrait être remplacé par un `do` avec éventuellement des variables intermédiaires et des `if` (rappel cours algorithmique : tout algorithme peut s'écrire avec séquence, itération et alternative) mais…
  - Clarté du programme : pour les affectations, réserver le `do` aux cas où la séquence est importante (calcul récursif d'une variable, etc.), utiliser `forall` dans les autres cas
  - Bonus : meilleure performance possible avec `forall` (selon compilateur), possibilité de parallélisation automatique

Lecture ou écriture d'un tableau

- Possibilité de faire référence à un tableau, et pas seulement à des éléments de tableaux, dans une lecture ou écriture

```
integer a(10,2)
```

Non seulement :

```
read *, a(3, 2)
print *, a(4, 1)
```

mais aussi possible :

```
read *, a
print *, a(5:, :)
```

Lecture ou écriture d'un tableau (suite)

- Lecture ou écriture dans l'ordre des éléments de tableau (cf. page 78)

```
integer a(10,2)
print *, a(5:, :)
```

affiche à la suite :

`a(5, 1)`, `a(6, 1)`, …, `a(10, 1)`, `a(5, 2)`, …

sur autant de lignes que nécessaire (selon compilateur)

Lecture d'un tableau

- `read *,` tableau
  Éléments entrés séparés par virgules ou espaces ou retours à la ligne à votre convenance

Lecture d'un tableau (suite)

- Attention, contre-intuitif, lecture des éléments d'une matrice : colonne par colonne

```
integer a(2, 2)
read *, a
1 2
3 4
```

→ `a` vaut $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

- Éventuellement :

```
do i = 1, 2
   read *, a(i, :)
end do
```

Tableaux et procédures : argument tableau

- Dans procédure, argument muet tableau déclare seulement son rang :

```
subroutine foo(a)
real, intent(...):: a(:, :, :)
```

  Un caractère `:` par dimension

  Profil imposé par l'argument effectif

- À l'appel : correspondance obligatoire du rang entre arguments effectif et muet

Tableaux et procédures : argument tableau (suite)

- Rappel (cf. page 58) : correspondance obligatoire du type et sous-type entre arguments effectif et muet (pour les arguments tableaux comme pour les arguments scalaires)
- Possibilité de choisir librement les bornes inférieures de l'argument muet

```
integer a(10)
call foo(a)
```

```
subroutine foo(a)
integer, intent(…):: &
   a(0:)
```

Tableaux et procédures : argument effectif tableau dynamique

- Si l'argument effectif est un tableau `allocatable`, il doit être alloué avant l'appel
- (Pas d'attribut `allocatable` pour un argument muet)

Tableaux et procédures : argument effectif tableau avec indice vectoriel

- Argument effectif tableau avec indice vectoriel (cf. page 86) seulement si l'argument muet a l'attribut `intent(in)`.

  Exemple :

  ```
  call foo(b((/3, 5, 1/))
  ```

  seulement si :

  ```
  subroutine foo(a)
  integer,intent(in):: a(:)
  ```

Profil d'un argument muet tableau

- Dans la procédure appelée, le profil pour l'appel courant est donné par `shape` ou `size`.
  `shape(`tableau`)` donne le profil.
  `size(`tableau`)` donne taille totale.
  `size(`tableau`, dim=`...`)` donne taille dans une dimension.

Profil d'un argument muet tableau : exemple 1

- Appelant :
  ```
  integer a(3, 4)
  call foo(a)
  ```
  Appelé :
  ```
  subroutine foo(a)
  integer, intent(...): a(:, :)
  ```
  `shape(a)` vaut `(/3, 4/)`
  `size(a)` vaut 12
  `size(a, 1)` vaut 3
  `size(a, 2)` vaut 4

Exemple 2 : évaluation d'un polynôme

$p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$

Économie de calculs :

c_n

$c_n x + c_{n-1}$

$c_n x^2 + c_{n-1} x + c_{n-2}$

...

```
tableau c(0:n)
p = c(n)
pour i = n-1 à 0
par -1
    p = p*x + c(i)
fin pour
```

Exemple 2 : évaluation d'un polynôme (suite)

```
real function polynomial(c, x)
   real, intent(in):: c(0:), x
   integer n, i ! local variables
   !---------------
   n = size(c) - 1
   polynomial = c(n)
   do i = n – 1, 0, - 1
      polynomial = polynomial*x+c(i)
   end do
end function polynomial
```

Tableaux automatiques

Variable locale d'une procédure (pas un argument) dont le profil n'est pas constant : il dépend d'un argument, ou d'une variable de l'hôte, ou d'une variable associée avec `use`.

Tableaux automatiques : exemple 1

```
subroutine swap(a, b)
  real, intent(inout):: a(:), b(:)
  real work(size(a))
  work = a
  a = b
  b = work
end subroutine swap
```

Tableaux automatiques : exemple 2

- ```
  subroutine plouf(n)
    integer, intent(in):: n
    real x(n)
    …
  ```
- Le tableau `x` doit être créé à l'appel de `plouf` donc `n` doit être connu : attribut `intent(in)` ou `intent(inout)` obligatoire
  ```

Tableaux automatiques : conseil

- Déclaration de tableau automatique : forme d'allocation dynamique
- Lorsque le profil est connu au moment de l'appel, pour la clarté et la performance, préférer un tableau automatique à un tableau `allocatable`

Fonction à valeur tableau

- Le résultat d'une fonction peut être un tableau
- Le profil du tableau doit être déclaré explicitement dans la fonction
- Le profil du tableau peut ne pas être constant et faire intervenir les arguments de la fonction (ou une variable de l'hôte, ou une variable associée avec `use`) (comme un tableau automatique).

Fonction à valeur tableau : exemple avec un profil constant

```
function o3_prod(o3_fraction, …)
integer, parameter:: n_long=16, &
   n_lat=12, n_vert=11
real o3_prod(n_long, n_lat, n_vert)
…
program …
…
dq_o3 = o3_prod(…) * time_step
```

Fonction à valeur tableau : exemples avec un profil non constant

- ```
  FUNCTION zroots_unity(n)
  integer, intent(in):: n
  complex zroots_unity(n)
  ```
  …
- ```
  FUNCTION cumprod(arr)
  real, intent(in):: arr
  real cumprod(size(arr))
  ```
  …

Procédures intrinsèques

Procédures intrinsèques

- Nombreuses
- Utiles : concision, clarté, performance
- Difficulté : soupçonner qu'il existe une procédure intrinsèque adaptée, penser à l'utiliser
- Conseil : de temps en temps, lire en diagonale la liste des procédures intrinsèques

Fonctions intrinsèques distributives

- Terme officiel en anglais : *elemental*
- Une fonction distributive peut s'appliquer à un scalaire et retourner un scalaire ou s'appliquer à un tableau et retourner un tableau de même profil
- Dans le cas d'un tableau, l'opération scalaire est distribuée sur tous les éléments du tableau en entrée
- Exemple : `sin(x)`, `sin((/x, 2*x, 3*x/))`

Fonctions intrinsèques distributives (avec conversion éventuelle)

- Fonctions de conversion :
  `int`, `floor`, `ceiling`, `nint`, `real`, `cmplx`, etc.
- Autres fonctions avec conversion éventuelle :
  `abs` (sur entier, réel ou complexe)
  `aimag`

Fonctions intrinsèques distributives (sans conversion)

- `mod`, `modulo`, `conjg`
- `max`, `min`
  Exemple :
  `a` vaut `(/1, 2, 0, 0/)`
  `b` vaut `(/2, 1, 2, 3/)`
  `c` vaut `(/3, 1, 0, 4/)`
  `max(a, b, c)` vaut `(/3, 2, 2, 4/)`
- `sign`

Fonctions intrinsèques distributives (mathématiques)

- `acos`, `asin`, `atan`, `cos`, `sin`, `tan`
- `atan2`
  `atan2(x, y)` : argument dans]-π, π] de x+iy
- `sqrt`, `log`, `log10`, `exp`, `cosh`, `sinh`, `tanh`

Fonction intrinsèque distributive `merge`

- `merge(tsource, fsource, mask)` vaut `tsource` si `mask` est vrai et `fsource` sinon
- `tsource` et `fsource` : même type, même sous-type, profils compatibles
- `mask` : logique, profil compatible avec `tsource` et `fsource`
- Exemple : `merge('a', 'b', i > 0)`

Fonction intrinsèque distributive `merge` (suite)

- Attention : `tsource` et `fsource` doivent être calculables quel que soit `mask`

```
if (b /= 0.) then
   x = a / b
else
   x = 0.
end if
```

n'est pas remplaçable par `merge`

Fonction intrinsèque distributive `merge` (suite)

- Exemple avec arguments tableaux :
  Si :
  `tsource` vaut `(/1, 3/)`
  `fsource` vaut `(/7, 0/)`
  `mask` vaut `(/.true., .false./)`
  alors :
  `merge(tsource, fsource, mask)`
  vaut `(/1, 0/)`

Fonctions intrinsèques pour attributs de tableaux

- `lbound(array[, dim])`
  `ubound(array[, dim])`
  `shape(source)`, `size(array[, dim])`
- Exemples :
  `integer a(0:3, 7)`
  `ubound(a)` donne `(/3, 7/)`
  `shape(a)` donne `(/4, 7/)`
  `size(a)` donne 28
- Ces fonctions ne dépendent pas des valeurs du tableau. Le tableau peut ne pas être défini.

Fonctions intrinsèques de réduction de tableau

- Cas simple : ces fonctions appliquent une opération à l'ensemble des éléments d'un tableau pour produire un scalaire
  (« réduction » : passage d'un tableau à un scalaire)
- Sur tableau numérique :
  `sum`, `product`, `minval`, `maxval`
- Sur tableau logique :
  `any` (ou logique), `all` (et logique), `count` (nombre de valeurs vraies)

Fonctions intrinsèques de réduction de tableau : exemple

$$m = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

$$m2 = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$$

`any(m==m2)` vaut `.true.`

`all(m==m2)` vaut `.false.`

`count(m==m2)` vaut 3

`sum(m)` vaut 21

Fonctions intrinsèques de réduction de tableau (suite)

- Appliquées à tableau vide, elles renvoient l'élément neutre de l'opération :

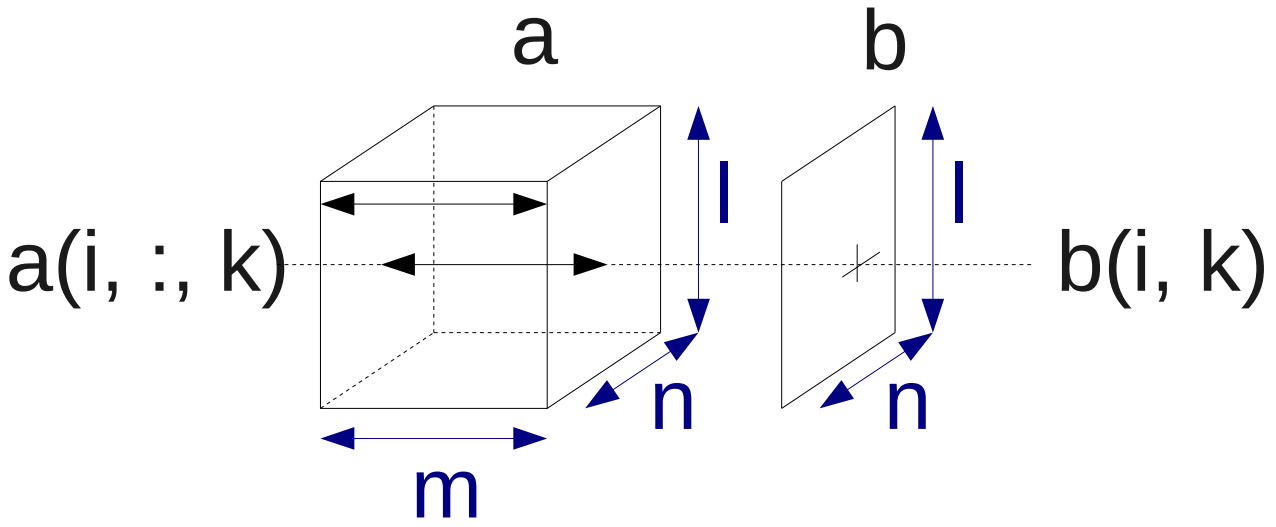
```
sum → 0
product → 1
minval → huge(…)
maxval →- huge(…)
any →.false.
all →.true.
count → 0
```

Fonctions intrinsèques de réduction de tableau : argument `dim`

- Argument `dim` facultatif :
  `sum(array, dim)`, `product(array, dim)`,
  ...
  `any(mask, dim)`, `all(mask, dim)`,
  `count(mask, dim)`
- Application de l'opération sur des vecteurs dans la dimension spécifiée
  `sum(m, dim=1)`
  vaut `(/3, 7, 11/)`

$$m = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Fonctions intrinsèques de réduction de tableau : argument `dim` (suite)




```
b = product(a, dim=2)
```

`b(i, k)` vaut `product(a(i, :, k))`

Toujours une réduction : diminution de 1 du rang

Fonctions intrinsèques de réduction de tableau numériques: argument `mask`

- Argument `mask` facultatif pour les fonctions de réduction numériques :
  `sum(array, dim, mask)`
  `product(array, dim, mask)`
  …
- `mask` : tableau logique compatible avec `array`
- Exemple :
  `sum(a, mask=a>0)`

Produit scalaire

- `dot_product(vector_a, vector_b)`
  Vecteurs de même taille, type numérique ou logique
- Si type entier ou réel :
  vaut `sum(vector_a * vector_b)`
- Si type complexe :
  vaut `sum(conjg(vector_a)*vector_b)`
- Si type logique :
  vaut `any(vector_a .and. vector_b)`

Produit matriciel et transposition

- `matmul(matrix_a, matrix_b)`
  2 matrices ou bien 1 matrice et 1 vecteur
  Type numérique ou logique
  Contraintes sur les tailles pour que le produit matriciel soit bien défini
- `transpose(matrix)`
  Sur un tableau de rang 2 de type quelconque

Localisation d'un extremum

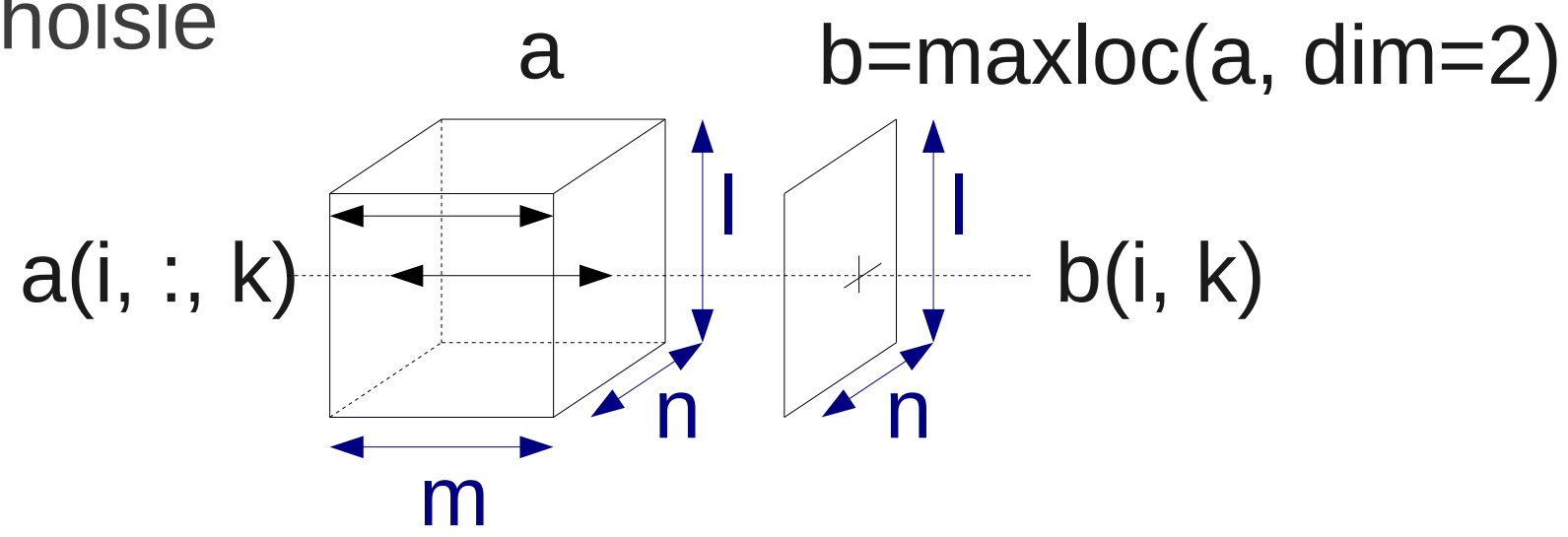
- `minloc(array[, mask])`
  `maxloc(array[, mask])`
  Résultat : vecteur des indices du premier extremum trouvé
- Exemples :
  V: `(/7, 6, 9, 6/)`

$$A : \begin{bmatrix} 8 & -3 & 0 & -5 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & 4 \end{bmatrix}$$

  `minloc(V)` vaut `(/2/)`
  `minloc(A, mask=A>-4)` vaut `(/1, 2/)`

Localisation d'un extremum (suite)

- Autre forme, avec argument `dim` :
  `minloc(array, dim, [, mask])`
  `maxloc(array, dim, [, mask])`
  Résultat : tableau de rang diminué de 1
  Chaque élément du résultat est l'indice de l'extremum du vecteur dans la dimension choisie



Localisation d'un extremum (suite)

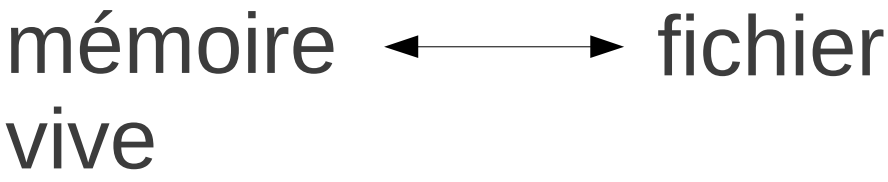
- Exemples :
  V: `(/7, 6, 9, 6/)`

$$A : \begin{bmatrix} 8 & -3 & 0 & -5 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & 4 \end{bmatrix}$$

  `minloc(V, dim=1)` vaut `2`
  `minloc(A, dim=2)` vaut `(/4, 3, 1/)`

Entrées et sorties

Entrées et sorties



- Mémoire vive : ne persiste que pendant le temps d'exécution
- Fichier : extérieur au programme, stocké sur disque, ou écran (écriture seulement) ou clavier (lecture seulement)

Fichiers en Fortran

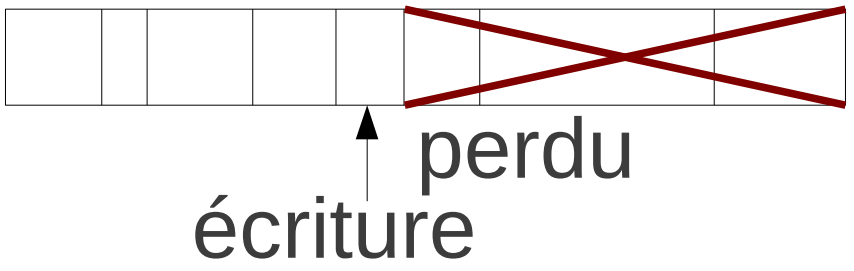
- Notion d'enregistrement (*record*)
  Un fichier est une suite d'enregistrements. Les opérations simples de lecture ou écriture traitent un enregistrement à la fois.
- Deux distinctions :
  - accès séquentiel ≠ accès direct
  - format texte (*formatted*)
    ≠ format binaire (*unformatted*)

Accès séquentiel

- Notion de « position courante » dans le fichier. On ne peut lire ou écrire qu'à la position courante. Pour aller d'une position à une autre, plus loin, obligation de lire tout les enregistrements entre les deux positions.
- Les enregistrements n'ont pas forcément la même taille. Marque de fin d'enregistrement.

Accès séquentiel (suite)

- Possibilité d'écrire seulement à la fin du fichier. (→ possibilité de récrire seulement le dernier enregistrement)



Accès direct

- Possibilité de se placer directement à un enregistrement quelconque du fichier, pour le lire ou l'écrire
- Possibilité de récrire n'importe quel enregistrement (sans perdre la suite du fichier)
- Restriction : tous les enregistrements doivent avoir la même longueur
  (→ il peut ne pas exister de marque de fin d'enregistrement)

Format texte ou bien binaire

Format binaire : représentation interne au programme d'une valeur
Format texte : suite de caractères
Par exemple :
$2^{14} = 16384 = (100000000000000)_2$

Écriture de la valeur formattée : 5 codes de caractères (typiquement 5 fois huit bits)
Écriture de la valeur non formattée : typiquement une suite de 32 bits

Format texte ou binaire : avantages et inconvénients

- Portabilité :
  binaire : -
  (le codage dépend du compilateur et de la machine)
- Performance du programme Fortran :
  texte : -
  (temps de conversion en caractères)
- Taille de fichier :
  binaire : économie de taille
  Exemple : 8.328622e-17
  4 octets en binaire, 12 octets en texte

Format texte ou binaire : avantages et inconvénients (suite)

- Exactitude :
  texte : -
  (pas forcément de représentation exacte binaire d'un décimal ; pas forcément de représentation décimale exacte d'un développement binaire fini ; pas les mêmes troncatures)
  Exemple :
  $1 / 10 = 0,1 = (0,000110011001\ldots)_2$
  $= 1/16 + 1/32 + 0/64 + 0/128 + 1/256 + 1/512 + 0/1024 + \ldots$

Format texte ou binaire : avantages et inconvénients (suite)

	binaire	texte
portabilité	-	+
lisibilité immédiate	-	+
performance programme Fortran	+	-
exactitude	+	-
économie de taille de fichier	+	-

Fichier temporaire

- Possibilité de créer un fichier temporaire : qui n'existe que pendant l'exécution du programme
- Intérêts : stocker un ensemble d'informations qui grossit au cours de l'exécution du programme et dont la taille finale n'est pas connue à l'avance ; éviter la complexité d'allouer, de désallouer et de réallouer des tableaux ; économiser de la mémoire vive.
- Inconvénient : lenteur des accès au disque par rapport aux accès à la mémoire vive.

Utilisation des accès et des formats

	accès séquentiel	accès direct
format binaire	un peu (fichier temporaire)	- par instructions Fortran + par bibliothèques (NetCDF…)
format texte	+	-

Instructions d'entrées-sorties

- `open`
  `read, write, print`
  `close`
  `rewind`
  `inquire`
- Pour écrire sur la sortie standard (normalement l'écran) ou lire sur l'entrée standard (normalement le clavier) : pas de `open` ni de `close`.
  Dans les autres cas : `open` et `close` obligatoires.

Arguments de `open`

- À l'ouverture d'un fichier, on choisit :
  - `access="sequential"` ou `"direct"`
  - `form="formatted"` (texte) ou `"unformatted"` (binaire)
  - `status="old"` (fichier pré-existant) ou `"new"` (fichier ne doit pas pré-exister) ou `"replace"` (détruit si pré-existant) ou `"scratch"` (fichier temporaire)

Arguments de `open` (suite)

- `action="read"`, `"write"` ou `"readwrite"`
  Remarque : logiquement `"readwrite"` pour fichier temporaire
- `position="rewind"` ou `"append"`

- Syntaxe :

```
open(..., access="...", form="...", &
  status="...", action="...", &
  position="...")
```

Entrée et sortie standard

- Cf. page 26
  `read *, ...`
  `print *, ...`
- Accès séquentiel, format texte
- `print *, ...`
  est un raccourci, exactement équivalent à :
  `write(unit=*, fmt=*) ...`
- `read *, ...`
  est un raccourci, exactement équivalent à :
  `read(unit=*, fmt=*) ...`

Numéro d'unité

- Pour un fichier autre que l'entrée ou la sortie standard, les instructions
  `read, write`
  `close`
  `rewind`
  `inquire`
  font référence au fichier via un « numéro d'unité ».

Numéro d'unité (suite)

- Pour un fichier temporaire, l'instruction `open` crée le fichier et lui assigne un numéro d'unité. Le fichier n'a pas de nom.
- Un fichier non temporaire a un nom, l'instruction `open` associe un numéro d'unité à ce nom de fichier.
- Numéro d'unité : nombre entier ≥ 0. L'ensemble des numéros autorisés dépend du compilateur. Conseil : utiliser l'instruction `inquire` pour trouver un numéro disponible.

Numéro d'unité (suite)

- Exemple avec fichier non temporaire :

```
open(unit=1, file="plouf.txt", &
  status="old", action="read", &
  position="rewind")
read(unit=1, fmt=*) x
close(unit=1)
```

- Exemple avec fichier temporaire :

```
open(unit=1, form="unformatted", &
  status="scratch", &
  action="readwrite")
```

  Pas d'argument `file`

Numéro d'unité (suite)

- L'association numéro d'unité – fichier persiste dans tout le programme, entre le moment de `open` et le moment de `close`. On peut par exemple ouvrir un fichier dans une procédure et faire une lecture dans une autre procédure : il suffit de transmettre le numéro d'unité.

Utilisation de `inquire` pour trouver un numéro d'unité

```
subroutine new_unit(unit)
  integer, intent(out):: unit
  logical opened, exist
  !----------------------
  unit = 0
  do
    inquire(unit=unit, opened=opened, &
          exist=exist)
    if (exist .and. .not. opened) exit
    unit = unit + 1
  end do
end subroutine new_unit
```

Utilisation de `inquire` pour trouver un numéro d'unité (suite)

```
integer unit
…
call new_unit(unit)
open(unit, …)
read(unit, fmt=*) …
close(unit)
```

Erreur d'entrée ou sortie

- Une erreur entraîne normalement l'arrêt de l'exécution.
  Exemple :
  ```
  open(..., file="plouf.txt", &
    status="old")
  ```
  et le fichier n'existe pas → arrêt avec un message d'explication généré automatiquement par le compilateur.
  Bonne solution en général.

Erreur d'entrée ou sortie (suite)

- Pour continuer malgré une erreur : argument `iostat` des instructions `open`, `read` et `write`. C'est un argument de sortie de ces instructions, de type entier.
  Exemple :
```
integer iostat
open(…, file="plouf.txt", &
  status="old", iostat=iostat)
```
  `iostat` vaut 0 si et seulement s'il n'y a pas eu d'erreur. Programmer le test.

Instruction `open` : résumé

- Valeurs par défaut à l'ouverture d'un fichier :
  `access="sequential"`
  `form="formatted"`
- Accès séquentiel, format texte (non temporaire) :

```
open(unit=..., file="...", &
  status="...", action="..."[, &
  position="..."][, iostat=...])
```

`position` utile seulement pour `status="old"`

Instruction `open` : résumé (suite)

- Fichier temporaire :

```
open(unit=..., form="unformatted", &
  status="scratch", &
  action="readwrite")
```

Instruction `rewind`

Pour un fichier ouvert en accès séquentiel :
`rewind(unit=...)`
rembobine le fichier. Après cette instruction, la position courante est sur le premier enregistrement.

Format texte : formattage

- Le « formattage » est la conversion entre la représentation interne d'une valeur et sa représentation sous forme de caractères.
  Exemple :
  1.2e-2 ou 0.012 ou … ↔

```
0 0 1 1 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 0 0 1 1 0
s|      E      |                       f
 |     120      |
```

- `read *, x`
  `print *, x`
  Formattage automatique par le compilateur, en fonction du type de `x`

Formattage explicite

- Possibilité de choisir le formattage.
  Par exemple en écriture :
  0.012 ou 1.2e-2
- `write(unit=...,` `fmt=`format[`,` ...]`)` `&`
  liste d'expressions
  `read(unit=...,` `fmt=`format[`,` ...]`)` `&`
  liste de variables

Formattage explicite (suite)

- `print` format`,` liste d'expressions
  est un raccourci, exactement équivalent à :
  `write(unit=*, fmt=`format`) &`
  liste d'expressions
- `read` format`,` liste de variables
  est un raccourci, exactement équivalent à :
  `read(unit=*, fmt=`format`) &`
  liste de variables

Format

- Le format est une chaîne de caractères entre parenthèses :
  `"(…)"`
  Par exemple :
```
print "(…)", x, x + 3.
read(unit, fmt="(…)") x
```
- Entre les parenthèses : liste de « modèles d'édition » (anglais : *edit descriptors*), pour contrôler le formattage de tous les éléments à lire ou écrire.

Modèle d'édition : écriture d'un entier

Modèle `i`w où w est le nombre de caractères

Exemple :

```
print "(i3)", 5
```

donne :

bb`5`

(*b* est un blanc)

Modèle d'édition : écriture d'un réel sans exposant

Modèle `fw.d` où w est le nombre de caractères et d le nombre de chiffres décimaux
Prévoir w ≥ d + 2 (signe et point décimal)
Exemple :

```
print "(f4.1)", 4.86
```

donne :

b`4.9`

Modèle d'édition : écriture d'un réel avec exposant

Modèle `esw.d` pour la forme mantisse multipliée par puissance de 10, avec :
$1 \leq |\text{mantisse}| < 10$
Exemple :

```
print "(es12.3)", 12338.
```

donne :
bbb`1.234E+04`
`E` signifie : $\times 10^{\cdots}$
Prévoir $w \geq d + 6$

Nombre de caractères dans le modèle d'édition

- Pour `i`w, `f`w`.d` et `es`w`.`d, si w est trop petit pour contenir la valeur à imprimer, impression de w astérisques. Pas d'erreur à l'exécution.
- `i0` et `f0.d` pour un nombre de caractères juste suffisant.

Modèle d'édition : écriture d'une chaîne de caractères

Modèle `a`. La chaîne est écrite telle quelle (pas de conversion en fait).

Exemple :

```
write(unit=*, fmt="(a)", &
   advance="no") "x = ?"
read *, x
```

Autre exemple :

```
print "(i2, a)", x, c
```

Modèle d'édition : insertion de blancs

- Modèle n`x` où « n » est le nombre de blancs à insérer.
  Exemple :
  ```
  print "(i2, 5x, a)", x, c
  ```

Autres éléments d'un format

- Facteur de répétition
  Exemple :
```
real x(4)
print "4(es12.3)", x
```
- Insertion de chaînes de caractères
  Exemple :
```
real x(4)
print '4(es12.3, " m ")', x
```

Instruction `format`

- Si on se sert plusieurs fois d'un même format un peu long, il peut être plus clair de ne l'écrire qu'une fois dans une instruction format : étiquette `format (...)`
- « étiquette » est une suite de 1 à 5 chiffres, avec au moins un chiffre ≠ 0.
  Traditionnellement, souvent : 4 chiffres.
- Attention : pas de guillemets ou d'apostrophes autour des parenthèses

Instruction `format` (suite)

- On fait référence au format via son étiquette dans l'instruction d'entrée ou sortie :
  `print` étiquette`,` …
  `write(unit=`…`, fmt=`étiquette[`,` …]`)` …
- L'instruction `format` peut être n'importe où dans l'unité de programme, après les déclarations (avant ou après les instructions qui y font référence). Le plus clair en général : regrouper les instructions `format` à la fin de l'unité de programme.

Instruction `format` : exemple

Au lieu de :

```
print '4(es12.3, " m ")', x
print '4(es12.3, " m ")', y
```

on peut écrire :

```
print 1000, x
print 1000, y
1000 format  4(es12.3, " m ")
```

Lecture avec formattage explicite

Intérêt limité. Principe : dire quelle est le nombre de caractères occupés par la valeur lue ; éventuellement pas de virgules entre les valeurs lues ; pas de guillemets pour encadrer une chaîne de caractères lue.

Lecture avec formattage explicite : exemples d'utilisation

- Lecture de lignes de texte dans un fichier. Ces lignes contiennent éventuellement des séparateurs (espace, virgule…) et ne sont pas encadrées par des guillemets.

```
character(len=200) line
read(unit, fmt="(a)", &
   iostat=iostat) line
```

Lecture avec formattage explicite : exemples d'utilisation (suite)

- Lecture de valeurs en nombre inconnu, séparées par des virgules, sur une ligne → nécessité de lire d'abord la ligne caractère par caractère pour compter le nombre de virgules

```
character c
read(unit, fmt="(a)", &
   advance="no", iostat=iostat) c
```

Lecture avec formattage explicite : exemples d'utilisation (suite)

- Permettre une question sans réponse.

```
character(len=10) name
print *, 'Nom de variable ?'
print *, '(par défaut toutes)'
read '(a)', name
if (name == "") then
  traitement de toutes les variables
else
  traitement de la variable choisie seulement
end if
```

Namelists

- Une « namelist » est un groupe de variables que l'on peut lire ou écrire en une seule instruction
- Déclaration :
  `namelist /`nom de namelist`/ &`
  liste de variables
  Exemple :
  `namelist /coef/ a, b, c`

Namelists (suite)

- Instructions de lecture et écriture :
  `read(`unité logique`, nml=`nom de namelist`)`
  `write(`unité logique`, nml=`nom de namelist`)`
  Exemple :
  ```
  read(unit=*, nml=coef)
  ```
- Le formattage au moment de la lecture ou de l'écriture est implicite.

Namelists (suite)

- Le nom de chaque variable apparaît dans le fichier lu ou écrit, avec la valeur correspondante. Les variables sont séparées par des virgules ou un retour à la ligne. Les lignes lues ou écrites commencent par :
  `&`nom de namelist
  et finissent par le caractère `/`.
  Exemple :

```
&coef a = 1, b = 3
c = 0 /
```

Namelists : intérêts

- Lisibilité du fichier lu ou écrit : les noms des variables sont associées aux valeurs. En lecture, possibilité d'ajouter des commentaires après un point d'exclamation.
  Exemple de fichier lu :

```
&time_control
run_days  = 1 ! run time in days
…
/
```

- Concision de l'instruction de lecture ou écriture

Namelists : intérêts (suite)

- Souplesse
  - Formattage implicite
  - Valeurs par défaut
  - Ordre quelconque des variables dans le fichier
  - Lecture d'une partie de tableau

Namelists : valeurs par défaut

En lecture, possibilité de ne définir qu'une partie des variables. Les variables qui n'apparaissent pas dans le fichier lu conservent leur valeur antérieure. Moyen d'avoir des valeurs par défaut des paramètres d'un programme.

Exemple :

```
a = 1; b = 0; c = 0
read(unit=*, nml=coef)
```

L'utilisateur peut écrire au moment de l'exécution :

```
&coef b = -2 /
```

Namelists : autres exemples

- Ordre quelconque des variables
  ```
  namelist /coef/ a, b, c
  ```
  Exemple de fichier lu :
  ```
  &coef
  b = 3, a = 5 /
  ```
- ```
  integer a(10)
  a = 0 ! default value
  namelist /plouf/ a, b, x
  ```
  Fichier lu possible :
  ```
  &plouf b = 5, a(3) = 1 /
  ```
  ```

Namelists : conclusion

- Utiles
- Distribution avec un programme d'un fichier contenant une namelist pour ce programme
- Utilisées notamment pour les gros programmes scientifiques, qui peuvent fonctionner dans plusieurs configurations. Intérêt de disposer d'un fichier contenant les namelists pour une configuration donnée.
  Exemple : Weather Research and Forecasting (WRF) modeling system.

Appendice

Appendice : formats fixe et libre

- Format libre apparu avec Fortran 90 : disposition libre des caractères sur une ligne dans un programme
- Fortran 77 et avant : format fixe seulement.

Format fixe

- Les six premières colonnes sont réservées.
- Les lignes sont limitées à 72 caractères.
- Un caractère dans la sixième colonne indique une continuation de ligne.
- Un caractère C ou * dans la première colonne signale une ligne de commentaires.
- Les blancs entre les lettres d'un mot sont autorisés.