DEA

« Astrophysique et instrumentations associées »

Fortran 90/95

2004

Benoît Semelin

Quelques liens utiles:

Ce cours n'est pas exhaustif! Pour des renseignements complémentaires:

- Le cours de l'IDRIS (centre national de calcul du CNRS):

http://www.idris.fr/data/cours/lang/fortran/fortran_base/IDRIS_Fortran_cours.pdf

- Passer de f77 à f90 (en anglais):

http://www.nsc.liu.se/~boein/f77to90/f77to90.html

De f77 à f90

- f90 sait faire tout ce que f77 sait faire.
- f90 n'impose pas de format de fichier rigide.
- f90 propose de nouvelles fonctionnalités.
- f90 incite à une programmation plus robuste.
- f90 donne des codes plus compacts.
- f90/95 tente de préparer l'auto-parallelisation.

Déclaration de variable, compléments.

Mémoriser la valeur d'une variable locale:

```
SUBROUTINE ITERATION( )

INTEGER, SAVE  ::  count
```

La variable **count** garde sa valeur entre un appel à la procédure et le suivant.

Déclaration et allocation dynamique:

```
REAL, ALLOCATABLE, DIMENSION(:,:)  :: field

ALLOCATE(field(500,500))
...
DEALLOCATE(field)
```

L'allocation dynamique permet d'économiser la mémoire.

Opérations sur les tableaux

En fortran 90, les opérateurs arithmétiques s'appliquent aussi aux tableaux:

- Initialisation grâce à = :

```
REAL, DIMENSION(20,20,20)  :: field,grad

field=0.
grad=field
...
```

- Tout les éléments de **field** sont initialisés à 0.
- **field** est copié élément par élément dans **grad**.
- **field** et **grad** doivent avoir **la même forme** !
- **Pas besoin de boucle !**

- Opérations terme à terme :

```
REAL, DIMENSION(3)  :: x =(/1.,2.,3./)
REAL, DIMENSION(3)  :: y =(/3.,1.,2./)
REAL, DIMENSION(3)  :: z

z=x+y
z=x-y
z=x*y
z=x/y
z=2.+x
...
```

- Les opérations se font **élément par élément** :
  z=x*y => z(3)=x(3)*y(3)
- Les tableaux doivent avoir la **même forme !**
- Les scalaires prennent **automatiquement** la forme voulue.
- **Attention**, si b(2)=0., z=x/y produit une erreur!

- Appliquer une fonction intrinsèque à un tableau:

```
REAL, DIMENSION(20,20,20)  :: field,grad

field=log(grad)
```

- La fonction est appliquée élément par élément
- Les tableaux doivent avoir la **même forme !**

Fonctions intrinsèques spécifiques aux tableaux

Pour compléter les opérateurs et fonctions génériques, il existe des fonctions spécifiques aux tableaux:

- Produit scalaire : **DOT_PRODUCT**

```
REAL, DIMENSION(20)  :: a,b,c

c=DOT_PRODUCT(a,b)
```

- Réalise le produit scalaire de a et b.
- a et b doivent avoir la même forme (et être des vecteurs)

- Produit matriciel : **MATMUL**

```
REAL, DIMENSION(10,20)  :: a
REAL, DIMENSION(20,10)  :: b
REAL, DIMENSION(10,10)  :: c

c=MATMUL(a,b)
```

- Réalise le produit matricielle de a et b.
- Les conditions habituelles sur les dimensions s'appliquent.

Fonctions qui acceptent un **masque**:

Un masque est un tableau de type LOGICAL de même forme que l'argument principal de la fonction. Il résulte en général d'une condition logique. La fonction s'applique aux éléments de l'argument principal pour lesquels la valeur de l'élément du masque est .TRUE.

Exemples:

- **MAXLOC** (A , A < 0.) : Renvoie un vecteur contenant la position du plus grand élément de A, parmi les éléments qui respectent A < 0 (ceci est le masque).
- **MINLOC** (A , sin(A) < 0.5) : Renvoie un vecteur contenant la position du plus petit élément de A, parmi les éléments qui respectent sin(A) < 0.5

Fonctions sur les tableaux (suite)

Autres fonctions avec masque:

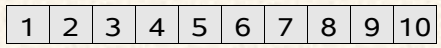
- **COUNT** (MASQUE) : Renvoie le nombre d'éléments TRUE dans le masque : count(A>0.)
- **ALL** (MASQUE): Renvoie TRUE si tout les éléments du masque sont vrais: ALL(A<10.)
- **ANY** (MASQUE): Renvoie TRUE si au moins un élément du masque est vrai: ALL(A=1.)
- **MAXVAL**(A , MASQUE) : Renvoie la valeur maximale de A masqué.
- **MINVAL**(A , MASQUE) : Renvoie la valeur minimale de A masqué.
- **PRODUCT**(A, MASQUE) : Renvoie le produit des éléments de A masqué.
- **SUM**(A,MASQUE): Renvoie la somme des élément de A masqué.
- **MERGE**(A,B,MASQUE): Renvoie un tableau contenant A la où le masque est TRUE, B ailleurs. MERGE(A,B,A>B)

Sections de tableaux: définition

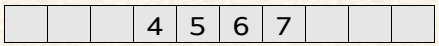
On peut travailler sur des parties d'un tableau.

- On définit une « section » de vecteur par triplet:
  (index_min : index_max : pas)

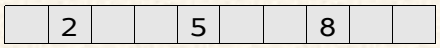
a()



a(4:7:1) , a(4:7)



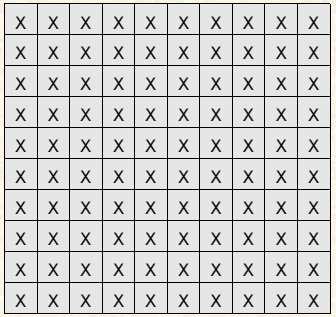
a(2:10:3) , a(2::7)



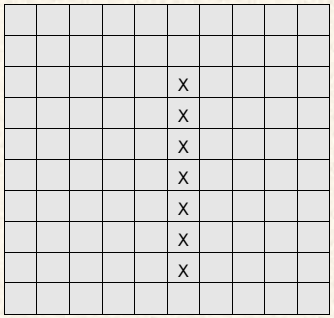
Si ils sont omis, les éléments du triplet ont une valeur par défaut, (1:dimension:1)

- Cela marche avec des tableaux à 2 (et plus) dimensions:

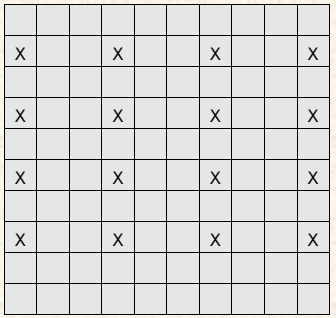
a()



a(3:9,6)



a(2:9:2,1:10:3)



Sections de tableaux: utilisation

Les sections de tableaux peuvent être utilisé pour l'initilisation, avec les opérateurs et les fonction intrinsèques:

```
REAL, DIMENSION(20,20)  :: mx,my
REAL, DIMENSION(4,4) :: coarse_x,coarse_y
REAL, DIMENSION(20) :: z

mx(:,1:20:2)=1.
mx(:,2:20:2)=0.

my=0
my(1:10,1:10)=mx(1:10,1:10)

coarse_x=mx(5:20:5,5:20:5)

z=mx(5,:)*my(:,5)
```

Rappel: les opérations se font entre tableaux de même forme.

Attention: une section de tableau n'a pas la même forme que le tableau!

Toutes les opérations valables sur les tableaux le sont sur les sections de tableaux.

Manipulation de tableaux grâce à « WHERE »

Les sections de tableaux possèdent une géométrie rigide (pas constant).Les masques offrent plus de souplesse (on peut définir leur géométrie à la main!). Il est possible de les utiliser en dehors des fonctions intrinsèques.

Utilisation de WHERE - ELSEWHERE - END WHERE:

f90

```
REAL, DIMENSION(20,20)  :: field

....
WHERE (field >3.)
   field=field*field*field
ELSEWHERE        <---------- (optionnel)
   field=field*field
END WHERE
```

f95

```
REAL, DIMENSION(20,20)  :: field

....
WHERE (field >3.)
   field=field*field*field
ELSEWHERE (field >2.)
   field=field*field
ELSEWHERE
   field=field
END WHERE
```

WHERE est équivalent, en plus compact, à une combinaison de DO et de IF.

Vers l'auto-parallélisation: fonction FORALL (f 95)

FORALL permet de combiner plusieurs DO et IF en une seule instruction:

```
FORALL( i = 1 : N , j = 1 : N , i /= j )
    phi(i)=1./(x(i)-x(j))
END FORALL
```

- Les boucles sur i et j sont spécifiées par des triplets 1:N(:1).
- La boucle n'est exécutée que si i/=j.

Ici, FORALL remplace deux boucles DO imbriquées et un IF.

ATTENTION:

- Le compilateur choisi l'ordre d'exécution des itérations!!!
- Donc, le résultat ne doit pas dépendre de l'ordre!
- x(i+1)=x(i)+x(i+1) ne doit pas être codé avec FORALL.

Pourquoi cette restriction forte?

Cela permet d'activer l'**auto-parallélisation** si le compilateur en est capable. Il distribue alors les itérations sur plusieurs processeurs.

Utilisation des modules

L'utilisation des **modules** produit des codes **plus robustes**, et **plus facile à débuguer**.
Un **module** est une partie de codes écrite dans un fichier séparé et compilée séparément. Il est utilisable par d'autres parties du codes à la demande.

Déclarer des variables communes

```
MODULE VARIABLES
IMPLICIT NONE
SAVE

INTEGER, PARAMETER :: N=50000
REAL, DIMENSION(3,N) :: pos,vel,acc

END MODULE VARIABLES
```

SAVE permet de conserver les variables de VARIABLES d'une procédure qui l'utilise à l'autre.

Procédures dans les modules.

```
MODULE DYNAMIQUE
USE VARIABLES
IMPLICIT NONE

..... autres déclarations.....

CONTAINS
   SUBROUTINE STEP(dt)
      IMPLICIT NONE
      REAL, INTENT(IN) :: dt
         vel=vel+acc*dt
         pos=pos+vel*dt
   END SUBROUTINE STEP

END MODULE DYNAMIQUE
```

Une procédure de module (fonction ou subroutine) est déclarée de façon **explicite:** le compilateur est capable, pour chaque appel à la procédure de vérifier la validité des arguments => débugage facile.

Pointeurs

Un pointeur est une variable qui contient une adresse mémoire. On les définit en f90 de la manière suivante:

```
REAL                :: y
REAL, TARGET :: x
REAL, POINTER :: p

p=>x      <-------- assignation (1)

y=p*p     <-------- utilisation (2)
```

(1) En f90 un pointeur ne peut pointer que vers une variable déclarée comme TARGET.
(2) Si, dans une expression, la valeur d'une variable est attendue et qu'un pointeur apparaît, il est remplacé par la valeur de la variable vers laquelle il pointe.

Pointeur vers un tableau, une section de tableau:

```
REAL, DIMENSION(100,100),TARGET   :: y
REAL, DIMENSION(:,:), POINTER :: p1,p2,p3

p1=> y
p2=> y(20:80:2,20:80:2)
p3=> p2(1::2,1::2)
```

On gagne de la place mémoire!

Fin