Méthodes numériques et éléments de programmation

Guy Munhoven

Institut d'Astrophysique et de Géophysique (Bât. B5c)
Bureau 0/13
eMail: Guy.Munhoven@ulg.ac.be
Tél.: 04-3669771

23 septembre 2014

Méthodes numériques et éléments de programmation
Guy Munhoven
Rappels
Sous-programmes

Plan du cours 2014-2015

Cours théoriques

16-09-2014 *Méthodes numériques pour E.D.O.: introduction*

22-09-2014 *Méthodes de Runge-Kutta; Contrôle du pas; Équations raides*

22-09-2014 *Fortran 95: bases, boucles*

23-09-2014 Fortran 95

- Rappels
- Branchements
- Sous-programmes

23-09-2014 Fortran 95: modules; opérations E/S

29-09-2014 Fortran 95: tableaux

07-10-2014 Fortran 95: compléments

Structure générale d'un programme Fortran
General Structure of a Fortran Program

[PROGRAM *nom_du_programme*]

instructions de spécification et de déclaration

instructions exécutables

[CONTAINS

procédures internes]

END [PROGRAM [*nom_du_programme*]]

Options indiquées entre [...], ces derniers étant à omettre

Instructions de spécification et de déclaration
Specifications and declarations

- Préciser des extensions à utiliser
  - ⇒ p. ex., USE *module*
- Adopter conventions de codage particulières
  - ⇒ IMPLICIT NONE (pas de typage par défaut)
- Définir les *types* de données à traiter
  - ⇒ à valeurs entières, réelles, chaînes de caractères, ...
- Conférer des *attributs* (propriétés particulières) à certaines données
  - ⇒ PARAMETER (constante symbolique)
  - ⇒ DIMENSION(...) (tableaux à plusieurs dimensions)
- Définir interfaces avec d'autres parties du programme

Déclarations
Declarations

- Forme générique d'une déclaration

  type [, *liste d'attributs*] [::] *identificateur* [= *valeur_initiale*]
- :: obligatoire si attributs ou valeurs initiales données

Exemples : déclarations de variables

```
INTEGER i, j, k
REAL :: x = 1.0, x2 = 1E-6
DOUBLE PRECISION :: y = -1.0d0, tol = 1D-12
LOGICAL :: l, option1 = .TRUE., option2 = .FALSE.
COMPLEX :: z
CHARACTER(LEN=80) :: nom_fichier, version_programme
```

Déclarations
Declarations

Exemples d'attributs : `PARAMETER`, `DIMENSION`, ...

Exemples : déclarations de variables avec attributs

```
DOUBLE PRECISION, PARAMETER :: pi = 3.14159265358979323846D+00
DOUBLE PRECISION, PARAMETER :: un_demi = 1D0/2D0
CHARACTER(LEN=*), PARAMETER :: version = 'Version 0.3 du 27NOV2007'
INTEGER, PARAMETER :: n_lons = 64, n_lats = 32, n_vert = 10
DOUBLE PRECISION, DIMENSION(12) :: temp_moy_mens
REAL, DIMENSION(n_lons, n_lats, n_vert) :: vitesse_vent, p_atmos
INTEGER, PARAMETER :: j_semaine = 7
INTEGER, PARAMETER, DIMENSION(12) :: njours_mois = &
                (/ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 /)
```

Initialisation à la déclaration: expressions permises

Initialisation at declaration: allowed expressions

Expression d'initialisation: expression constante faisant intervenir uniquement

- des constantes (identificateurs avec attribut `PARAMETER`)
- des opérations arithmétiques de base
- des exponentiations avec exposant `INTEGER`
- fonctions élémentaires intrinsèques à arguments de type `INTEGER` et/ou `CHARACTER` uniquement
- les fonctions intrinsèques de transformation `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRIM`
- les fonctions intrinsèques d'interrogation (*inquiry functions*) `LBOUND`, `UBOUND`, `SIZE`, `LEN`, `KIND`, ...
- constructeur de tableaux fixes

Méthodes numériques et éléments de programmation
Guy Munhoven
Rappels
Sous-programmes

Types de données : précision et gamme portable

Data Types : Portable Precision

- `SELECTED_INT_KIND(r)` retourne comme résultat
  - identifiant `INTEGER` du type `INTEGER` souhaité, si disponible pour le compilateur utilisé
  - `-1` sinon
- `SELECTED_REAL_KIND(p,r)` retourne comme résultat
  - identifiant `INTEGER` du type `REAL` souhaité, si disponible pour le compilateur utilisé
  - `-1` si la précision demandée n'est pas disponible
  - `-2` si l'étendue demandée n'est pas disponible
  - `-3` si ni précision ni étendue demandées ne sont disponibles
- Identifiants sont souvent, mais non pas obligatoirement, identiques au nombre d'octets par unité $\Rightarrow$ ne pas s'y fier

Types de données : précisions et gammes portables

Data Types : Portable Precision

Exemple d'utilisation des précisions et gammes portables

```
! 'Short integer': -10^5 < valeurs < +10^5
INTEGER, PARAMETER :: si = SELECTED_INT_KIND(5)
! 'Long integer': -10^9 < valeurs < +10^9
INTEGER, PARAMETER :: li = SELECTED_INT_KIND(9)

! 'Simple precision': 6 décimales de précision,
!                      10^-30 < ABS(valeurs) < 10^+30
INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6,30)
! 'Double precision': 16 décimales de précision,
!                      10^-300 < ABS(valeurs) < 10^+300
INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(16,300)

INTEGER(KIND=si) :: i = 10_si       ! short integer
INTEGER(KIND=li) :: j, k=1000000_li ! long integer

REAL(KIND=sp) :: x = 1E-24_sp       ! simple precision
REAL(KIND=dp) :: d = 1E+60_dp       ! double precision
```

Constantes littérales

Literal Constantes

Représentations de 10 comme constante

`10`	`INTEGER`, espèce par défaut
`10_lint`	`INTEGER`, espèce `lint` (si définie, i.e., si une constante `INTEGER` de nom `lint` a été déclarée et initialisée auparavant)
`10.; 1E1; 10.E0`	`REAL`, espèce par défaut
`10.D0`	`REAL`, espèce `DOUBLE PRECISION` par défaut
`10._wp`	`REAL`, espèce `wp` (si définie)

Formes possibles, mais déconseillées

`10_4`	`INTEGER`, espèce 4 (si supportée)
`10._8; 1.E+1_8`	`REAL`, espèce 8 (si supportée)

Opérations entre types différents : conversions
Operations Between Different Types: Conversions

- Conversion de type vers le type le plus fort
  - `INTEGER` $\prec$ `REAL` $\prec$ `DOUBLE PRECISION`
  - `REAL` $\prec$ `COMPLEX` de même espèce
  - exception: si `**` est suivie d'un `INTEGER`, ce dernier ne sera pas converti, quelque soit le type de la base de puissance
    - `(-1.)**2` sera évalué correctement à `(-1.)*(-1.) = 1.`
    - `(-1.)**2.` donne lieu à une exception arithmétique, car calculée comme `EXP(2.*LOG(-1.))`
- Conversions effectuées au fur et à mesure de l'évaluation d'expressions complexes: attention aux pièges et surprises

$\Rightarrow$ Inclure les constantes littérales avec leur type approprié

Contrôle de l'exécution d'un programme
Execution control

- Instructions exécutables suivent les spécifications et déclarations
- Normalement
  - exécution séquentielle des instructions, ligne par ligne
- Possibilité
  - de répéter des blocs d'instructions: *boucles*
  - de sélectionner des blocs d'instructions, d'en ignorer d'autres, selon des critères prédéfinis: *branchements* (alternatives, choix multiples, déviations)
  - d'arrêt: instruction `STOP`

Boucles: actions répétitives
Loops: Repetitive Action

Boucle indéterminée

[*nom_de_boucle*:] DO

    Instructions de la boucle

END DO [*nom_de_boucle*]

Boucle contrôlée par indices

[*nom_de_boucle*:] DO i = *i_ini*, *i_fin* [, *i_inc*]

    Instructions de la boucle

END DO [*nom_de_boucle*]

Boucle contrôlée par condition logique (DO WHILE())

[*nom_de_boucle*:] DO WHILE(*condition_logique*)

    Instructions de la boucle

END DO [*nom_de_boucle*]

Boucles: actions répétitives
Loops: Repetitive Action

Méthodes numériques et éléments de programmation
Guy Munhoven
Rappels
Sous-programmes

- Si un *nom_de_boucle* précède DO, il doit obligatoirement suivre le END DO correspondant.
- Aussi bien l'indice i qui contrôle une boucle, que les *paramètres de boucle* *i_ini*, *i_fin* et *i_inc* doivent être de type INTEGER.
- Il n'est pas permis de modifier directement la valeur de l'indice i à l'intérieur de la boucle.
- La *condition_logique* d'une boucle DO WHILE ne peut pas se réduire à une simple variable logique: en modifiant la valeur de cette variable à l'intérieur de la boucle on modifierait le *critère* de la boucle, ce qui n'est pas permis.

Branchements : exécution conditionnelle
Branching: Conditional Execution

- Instructions normalement exécutées de manière séquentielle
- Branchements et structures de contrôle permettent des ordres d'exécution différents selon des cas à définir
- Deux types majeurs
  - Branchement conditionnel
  - Choix multiple

Branchement conditionnel : `IF-THEN-ELSE`
Conditional Branching: `IF-THEN-ELSE`

- Forme générique

```
[nom_de_branchement:] IF (expression_logique_1) THEN
    Bloc d'instructions 1
[ELSEIF (expression_logique_2) THEN [nom_de_branchement]
    Bloc d'instructions 2]
...
[ELSE [nom_de_branchement]
    Bloc d'instructions n]
ENDIF [nom_de_branchement]
```

- *nom_de_branchement* optionnel pour structurer le programme (p.ex., pour branchements imbriqués, ou `IF`/`ENDIF` éloignés l'un de l'autre), et unique dans l'unité de programme

Branchement conditionnel : IF-THEN-ELSE

Conditional Branching: IF-THEN-ELSE

- Les *expression_logique_i* sont évaluées de manière séquentielle
- Uniquement le '*Bloc d'instructions*' de la première *expression_logique_i* qui est évaluée à `.TRUE.` est exécuté (éventuellement aucun)
- Forme courte sans alternatives, sur une seule ligne

```
IF (expression_logique) instruction
```

⇒ noter l'absence du `THEN` dans cette forme courte

Choix multiple : SELECT CASE

Multiple Choice: SELECT CASE

- Forme générique

```
[nom_de_branchement:] SELECT CASE (var)
    CASE (liste_valeurs_1) [nom_de_branchement]
        Bloc d'instructions 1
    [CASE (liste_valeurs_2) [nom_de_branchement]
        Bloc d'instructions 2]
    ...
    [CASE DEFAULT [nom_de_branchement]
        Bloc d'instructions par défaut]
END SELECT
```

- Possibilité d'inclure un *nom_de_branchement* optionnel pour structurer le programme

Choix multiple : SELECT CASE
Multiple Choice: SELECT CASE

- '*Bloc d'instructions x*' exécuté si *var* est incluse dans la liste de valeurs *liste_valeurs_x* (entre parenthèses), donnée de valeurs isolées *val* et/ou d'intervalles *inf*:*sup*, séparés par des virgules
- CASE DEFAULT fournit un bloc d'instructions à exécuter lorsqu'aucun des cas explicitement inclus n'est applicable
- *var* doit être de type INTEGER, LOGICAL ou CHARACTER de manière à ne considérer que des tests permettant de vérifier des égalités réalisables de manière exacte
- Les différents cas doivent être disjoints
- Les tests ne peuvent pas faire intervenir d'autres variables que *var* : tous les cas possibles doivent pouvoir être distingués déjà à la compilation

Sous-programmes
Subprograms

- Regrouper des tâches bien définies dans des entités délimitées, comme, p.ex.,
  - résolution d'un système d'équations linéaires
  - calcul d'une intégrale
  - calcul d'une fonction ne faisant pas partie des fonctions intrinsèques
- Modularisation de programmes
- Ré-utilisation de parties pour d'autres programmes
- Possibilité de travailler à plusieurs sur un même programme

Sous-programmes
Subprograms

Deux classes : fonctions, sous-routines

Forme générique d'une fonction

```
[type] FUNCTION nom_fonction ([liste_arguments]) [RESULT(var_resultat)]
   instructions de spécification et de déclaration
   instructions exécutables
[CONTAINS
   procédures internes]
END [FUNCTION nom_fonction]
```

Forme générique d'une sous-routine

```
SUBROUTINE nom_sous-routine [(liste_arguments)]
   instructions de spécification et de déclaration
   instructions exécutables
[CONTAINS
   procédures internes]
END [SUBROUTINE nom_sous-routine]
```

Sous-programmes
Subprograms

Méthodes numériques et éléments de programmation
Guy Munhoven
Rappels
Sous-programmes

- Structure semblable à celle d'un programme principal
- Un sous-programme peut faire appel à d'autres sous-programmes.
- Communication avec le programme ou sous-programme appelant via
  - variables de la liste d'arguments, appelées variables muettes ou factices (en Anglais : *dummy variables*);
  - blocs `COMMON` (un concept FORTRAN 77 obsolète en Fortran 90 — à remplacer par des `MODULE`s);
  - des `MODULE`s.
- Appel de fonction par affectation ou dans une expression

  `X = nom_fonction(liste_arguments_appel).`
- Appel de sous-routine par `CALL`

  `CALL nom_sous-routine(liste_arguments_appel).`

Variables de sous-programmes : attributs spéciaux
Special Attributes for Subprogram Variables

Variables muettes

- `INTENT(IN)` — le contenu d'une telle variable est destiné uniquement à être lu et ne peut pas être modifié;
- `INTENT(OUT)` — le contenu d'une telle variable est supposé indéfini au moment de l'appel et ne peut pas être utilisé avant initialisation dans le sous-programme-même;
- `INTENT(INOUT)` — variable peut être considérée comme initialisée et son contenu comme librement modifiable;
- `OPTIONAL` — variable optionnelle, dont la présence ou non peut être testée à l'aide de la fonction logique intrinsèque `PRESENT(`*`optional_var_name`*`)`.

Variables de sous-programmes : attributs spéciaux
Special Attributes for Subprogram Variables

Méthodes numériques et éléments de programmation
Guy Munhoven
Rappels
Sous-programmes

Variables locales

- `SAVE` — le contenu d'une telle variable est sauvegardé et restera disponible lors d'appels ultérieurs:

  en général, une variable locale, à laquelle on n'a pas explicitement donné une valeur spécifique, doit être supposée comme indéfinie (c.-à-d., en pratique, à valeur aléatoire);

  toute variable initialisée à la déclaration reçoit automatiquement l'attribut `SAVE`.
- `EXTERNAL` — cette variable donne le nom d'un sous-programme (de type `FUNCTION` ou `SUBROUTINE`).

Sous-programmes de type fonction
Function Subprograms

- Permet d'élargir la librairie de fonctions intrinsèques.
- Retour du résultat de l'appel de fonction
  - en affectant la valeur de retour à une variable locale de la fonction qui porte le même nom que le fonction;
  - en affectant la valeur à la variable *var_resultat* donnée en argument de l'option `RESULT`.
- Typage du résultat soit par
  - les règles de typage par défaut à partir de la première lettre du nom de la fonction (*déconseillé*);
  - préfixation du mot-clé `FUNCTION` avec le type;
  - déclaration d'un identificateur du même nom que la fonction avec le type désiré au sein de la fonction;
  - déclaration de la variable *var_resultat* de l'option `RESULT` dans la partie de déclaration de la fonction;
  - option `RESULT` obligatoire si le résultat est un tableau.

Sous-programmes de type fonction
Function Subprograms

Exemple d'appel de fonction

```
PROGRAM demo_func
...
INTEGER :: n_termes=10000
DOUBLE PRECISION :: pi, pi_calc ! déclarer aussi la fonction
...
pi = pi_calc(n_termes)
END PROGRAM demo_func
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
DOUBLE PRECISION FUNCTION pi_calc(n)
INTEGER, INTENT(IN) :: n ! Variable muette
INTEGER :: i ! Variable locale
DOUBLE PRECISION :: delta, pm_un = -1D0
pi_calc = 1D0 ! Nom de fonction peut être utilisé comme variable
DO i = 2, n
  delta = 1D0/(2D0*DOUBLE(i)-1D0)
  pi_calc = pi_calc + pm_un*delta
  pm_un = -pm_un
END DO
END FUNCTION pi_calc
```

Alternatives de typage pour fonctions
Alternative Function Typing

Exemples de typage

```
DOUBLE PRECISION FUNCTION pi_calc(n)
INTEGER, INTENT(IN) :: n ! Variable muette
...
pi_calc = ...
END FUNCTION pi_calc
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
FUNCTION pi_calc(n)
DOUBLE PRECISION :: pi_calc
INTEGER, INTENT(IN) :: n ! Variable muette
...
pi_calc = ...
END FUNCTION pi_calc
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
FUNCTION pi_calc(n) RESULT(pi)
DOUBLE PRECISION :: pi
INTEGER, INTENT(IN) :: n ! Variable muette
...
pi = ...
END FUNCTION pi_calc
```

Sous-programmes internes/externes
Internal and External Subprograms

- Sous-routines et fonctions peuvent être incluses dans le programme principal ou d'autres sous-programmes entre les séparateurs `CONTAINS` et `END` ⇒ *procédures internes*
  - appelables uniquement à l'intérieur de l'unité de programme qui les contient
  - permet meilleure structuration des tâches à effectuer
  - *procédures internes* ont accès illimité aux variables de leur unité-hôte, sauf à celles qui y sont redéclarées, et qui restent locales à la procédure interne.
- Une procédure interne ne peut pas contenir de procédure interne.
- Tout sous-programme qui n'est pas contenu dans un autre programme/sous-programme est considéré comme *externe*, même s'il se trouve dans le même fichier.