PETITE INTRODUCTION AU LANGAGE FORTRAN

FRANÇOIS DUCROT

Document d'accompagnement du module « Modélisation et simulation en calcul scientifique »du master 1 Mathématiques, parcours ingénierie mathématique, de l'université d'Angers.

Fortran est un langage de programmation, développé par IBM vers 1955, et destiné à fournir aux scientifiques un moyen simple pour passer de leurs formules mathématiques jusqu'à un programme effectif (son nom est une abbréviation de FORmula TRANslator). Il est très efficace dans le domaine du calcul numérique, et offre de nombreuses bibliothèques de programmes d'analyse numérique.

Fortran a fait l'objet de plusieurs normalisations : fortran 77, fortran 90 et 95, et plus récemment fortran 2003. Le présent document s'intéresse aux versions 90/95 (peu différentes l'une de l'autre).

Le document décrit l'usage de fortran sous unix (linux), basée sur l'utilisation du compilateur `gfortran`, faisant partie de la suite de compilation `gcc` (Gnu Compiler Collection, `http://gcc.gnu.org/fortran`). Il s'agit d'une suite de logiciels sous licence libre, présente dans toutes les distributions linux. Des versions exécutables de `gfortran` pour Windows, Linux, MacOS sont disponibles à partir de la page web `http://gcc.gnu.org/wiki/GFortran`.

Tous les TP seront effectués dans un environnement Linux (distribution Fedora).

Table des matières

Date: 21 septembre 2011.

1. Un exemple

Créons un fichier texte nommé premierProgramme.f90, grâce à l'éditeur de texte emacs. Pour cela, on tape dans une fenêtre de terminal la commande

```
emacs premierEssai.f90
```

Rentrons dans ce fichier le petit programme :

```
! Calcul de la somme des 1/k^2 pour k de 1 a n
program premierEssai
  !Declarations de variables
  implicit none
  integer :: n,k
  real :: s
  !Lecture d'une donnée
  write(*,*) 'Rentrez un entier n :'
```

```
  read(*,*) n
  !Algorithme
  s=0
  do k=1,n
     s=s+1./k**2
  end do
  !Affichage du résultat
  write(*,*) 's= ',s
end program premierEssai
```

Une fois ce fichier écrit et enregistré, on va le compiler, en tapant la commande

```
gfortran premierEssai.f90 -o premierEssai.exe
```

Si le programme a été convenablement tapé, la machine devrait vous rendre la main, après avoir créé un fichier exécutable `premierEssai.exe`. Lançons ce programme comme dans la session suivante :

```
./premierEssai.exe
 Rentrez un entier n :
10000
 s=    1.644725
```

2. Quelques commentaires sur l'exemple

2.1. **Edition de texte.** Quand l'éditeur de texte emacs rencontre un fichier dont le nom se termine par `.f90`, il offre à l'utilisateur un certain nombre de commodités :

- coloration syntaxique,
- indentation automatique quand on frappe la touche `<Tab>`,
- complétion automatique des fins de boucle (quand on frappe `<Tab>` après avoir écrit le mot `end`).

On notera que fortran, contrairement au système unix de la machine, est insensible à la casse des lettres (majuscule ou minuscule). A l'intérieur du fichier, on peut donc écrire indifféremment `premieressai` ou `premierEssai` (que je préfère pour des raisons de lisibilité) ; de même certains font le choix d'écrire tous les mots clés en majuscule. Par contre, pour le système les noms `premierEssai.f90` et `premieressai.f90` désignent deux fichiers différents !

Dans un programme fortran, les lignes ne doivent pas dépasser 132 caractères. En cas de ligne trop longue, on peut la découper en plusieurs parties de la façon suivante :

```
debut de la ligne &
& suite de la ligne &
& et ainsi de suite ...
```

2.2. **Compilation.** L'étape de compilation, lancée par la commande `gfortran`, fabrique un fichier exécutable. Par défaut, cet exécutable se nomme par défaut `a.out`. Dans notre exemple, l'option `-o premierEssai.exe` lui a indiqué le nom de l'exécutable (le `-o` signifie « output »). L'extension `.f90` a indiqué au compilateur que le programme suivait les conventions de la norme f90 (sinon, il aurait pris l'option f77, beaucoup plus restrictive, et aurait décelé des erreurs).

2.3. **Structure du fichier source.**

- Un programme est constiué de toutes les instructions comprises entre `program nom-du-programme` et `end program nom-du-programme`
- Tout ce qui suit le caractère « ! » dans une ligne est considéré comme un commentaire, et n'est pas pris en compte par le compilateur.

– Toutes les variables intervenant dans le programme doivent être définies. C'est l'objet des premières lignes. L'instruction `implicit none` dit au compilateur que si une n'a pas été définie, il doit afficher un message d'erreur. Cette option n'est pas indispensable, mais vivement conseillée, car elle évite bien des erreurs.
A titre d'exemple, essayez de modifier les déclarations de variables, et regardez le résultat.
– Les instructions `write` et `read` permettent de gérer des entrées/sorties entre le programme et le couple écran/clavier

3. Les variables

Une variable est un emplacement mémoire désigné par un nom (expression commençant par une lettre, ne comportant pas de blancs, ni de caractères spéciaux (comme par exemple +,*,/,-). Une instruction de la forme `x=y+z` affecte à la variable `x` la somme des valeurs des variables `y` et `z`.

3.1. Différents types de variables.

(1) Les entiers : ils sont stockés sur 32 bits, ce qui permet de stocker des nombres compris entre -2^{31} et $2^{31}-1$ ($2^{31} \simeq 2.1510^9$). On déclare plusieurs variables entières grâce à une instruction du type
```
integer :: p,q,r,m
```

(2) Les nombres réels : un nombre réel est stocké en virgule flottante sur 32 bits, sous la forme $\pm(1+M)2^e$, où la mantisse M est stockée sur 23 bits et l'exposant sur 8 bits, ce qui permet de stocker des nombres x tels que $1.2\ 10^{-38} \leq |x| \leq 3.4\ 10^{38}$. On déclare des variables réelles grâce au mot clé `real`.
Un nombre réel peut être entré sous la forme `x=1.256e45`. Ainsi `x=1.e0` affecte à la variable `x`, le réel 1.

(3) Les nombres réels en double précision : un nombre réel est stocké en virgule flottante sur 32 bits, sous la forme $\pm(1+M)2^e$, où la mantisse M est stockée sur 52 bits et l'exposant sur 11 bits, ce qui permet de stocker des nombres x tels que $2.2\ 10^{-308} \leq |x| \leq 1.87\ 10^{308}$.
On déclare des variables réelles grâce au mot clé `double precision`.
Un nombre réel peut être entré sous la forme `x=1.256d45`. Ainsi `x=1.d0` affecte à la variable `x`, le réel en double précision 1.

(4) Les complexes : un complexe est un couple de réels en simple précision. Une variable complexe est déclarée grâce au mot clé `complex`.

(5) Les booléens : ce sont les valeurs de vérité « `.true.` »et « `.false.` ». Le mot clé est `logical`. On peut opérer sur les booléens, grâce aux opérateurs `.not.` , `.and.`, `.or.`, `.eqv.`, `.neqv.` et affecter à une variable booléenne le résultat d'un test ; ainsi la séquence
```
logical :: test
test = .not. (2>3)
```
affecte à la variable booléenne `test` la valer `.true.` . Différents opérateurs de test rendent des résultats booléens : `<`, `>`, `<=`, `>=`,`==`, `/=`.

(6) Les chaînes de caractères : une chaîne de caractère est une expression littérale écrite entre des apostrophes (quote en anglais). On déclare une variable de type chaîne de caractères grâce au mot clé `character`, en précisant la longueur de la chaîne de caractère par l'attribut `len`. Ainsi, si on veut utiliser une variable chaîne de caractère qui pourra prendre pour valeur les différents mois de l'année, on définira par exemple
```
character, len(8) :: mois
mois = 'decembre'
```

3.2. **Règles de calcul suivant le type.** Considérons la séquence suivante :

```
integer :: p,q
real :: x
p=1 , q=2
x=p/q
```

Le programme va effectuer le résultat de l'opération 1/2 (opération sur des entiers), dont le résultat est 0, puis va affecter le résultat à la variable réelle x. On a donc donné à x la valeur 0. Inversement, la séquence :

```
integer :: q
real :: p,x
p=1. , q=2
x=p/q
```

considère que p est un nombre réel, et effectue donc l'opération en virgule flottante, ce qui donnera à x la valeur attendue 0.5. Il importe donc de faire très attention aux types des variables qu'on considère.

3.3. **Les commandes `real`, `int` et `nint`.** Soit n une variable entière, la commande `real(n,4)`, ou simplement `real(n)`, rend n, mais vu comme un réel en simple précision (stocké sur 4 octets) et `real(n,8)` rend n, vu comme un réel en double précision (stocké sur 8 octets).
Inversement, si x est une variable réelle, en simple ou double précision, `int(x)` désigne l'entier obtenu en enlevant la partie décimale de x, et `nint(x)` désigne l'entier le plus proche de x.

3.4. **Variables de type paramètre.** Certaines variables correspondent à des grandeurs qu'on déclare une bonne fois pour toutes au début du programme, et qu'on ne voudra plus modifier au fil du programme :

```
integer, parameter  :: p=126
```

Dans cet exemple, on introduit un entier p, qu'on a déclaré comme paramètre, et auquel on affecte en même temps la valeur 126.

4. Les tableaux

Fortran sait gérer des tableaux d'entiers, de réels, de réels en double précision, de booléens. Un tableau est une collection de variables $(t_{i_1,\dots,i_r})$, toutes d'un même type, avec $1 \leq r \leq 7$ (r est appelé le rang du tableau, i_k est la taille du tableau dans la k-ième dimension, et $(i_1, \dots, i_r)$ est appelé le profil du tableau. le produit $i_1 \dots i_k$ est la taille du tableau, c'est à dire le nombre de variables élémentaires dont il est constitué. Un tableau de rang 1 correspond à la notion mathématique de vecteur (sans distinguer entre vecteur ligne ou colonne comme le ferait un logiciel comme scilab), et un tableau de rang 2 correspond à la notion mathématique de matrice.

4.1. **Déclaration.** Comme n'importe quelle variable, un tableau doit être déclaré au début du programme. On définit un tableau en indiquant le type et le profil du tableau. Ainsi

```
real, dimension(25,31,12)  :: t
```

définit un tableau `t` de réels de rang 3, de profil $25 \times 31 \times 12$, indexé par l'ensemble de triplets $[1, 25] \times [1, 31] \times [1, 12]$. On peut souhaiter indexer par des ensembles d'indices plus variés :

```
real, dimension(-2:2,0:6)  :: tt
```

définit un tableau `tt` de réels de rang 2, de profil 5×7, indexé par l'ensemble de triplets $[-2, 2] \times [0, 6]$.
On peut aussi réunir les deux précédentes déclarations en une seule :

```
real :: t(25,31,12), tt (-2:2,0:6)
```

Dans le cas d'un tableau de réels en double précision, d'entiers, ou de booléens, on remplacera le mot clé `real` par `double precision`, `integer`, ou `logical`.

4.2. **Constructeurs de tableau.** Pour construire un tableau de dimension 1, on peut lister ses éléments et les encadrer par `(/ ... /)`. Voici quelques exemples :

- `(/ 1, 2, 3 /)` produit le tableau $[1, 2, 3]$
- `(/ (i,i=1,100) /)` et `(/ (i,i=1,100,2) /)` produisent respectivement le tableau $[1, 2, \ldots, 100]$ et le tableau des nombres impairs compris entre 1 et 100.
- On peut combiner ceci comme dans l'exemple `(/ ( 0,(j,j=1,5),0, i=1,6) /)`, qui rend le tableau de taille 42, constitué de 6 fois la séquence $0, 1, 2, 3, 4, 5, 0$.

La commande `reshape(X,(/m,n/)` permet de fabriquer à partir d'un tableau linéaire de dimension mn un tableau rectangulaire de taille $m \times n$, en remplissant successivement la première ligne, puis la seconde, ...
Ainsi `reshape ( (/ (( i+j,i=1,100),j=1,100) /),(/100,100/))` construit la table d'addition des 100 premiers entiers.

4.3. **Extraction de parties d'un tableau.** Partons d'un tableau M, de dimension 2, indexé par $[1, 5] \times [1, 5]$. On peut en extraire différentes parties :

- `M(1,:)` désigne la première ligne de M
- `M((/1,2/),:)` désigne la sous matrice constituée des deux premières lignes de M
- `M((/1,2/),(/1,2/))` désigne la sous matrice 2×2 en haut à gauche
- `M(:,2:)` désigne la sous matrice constituée des quatre dernières colonnes
- `M(2:4,2:4)`

On peut combiner extraction et affectation pour modifier une matrice :

- `M(1,:)=0` remplit la première ligne de 0
- `M((/1,2/),:)=M((/2,1/),:)` échange les lignes 1 et 2

On peut aussi s'en servir pour accoler deux matrices, par exemple :

```
real(dimension(n,n) :: A,B
real(dimension(n,2*n) :: M
M(:,1:n)=A
M(:,n+1:)=B
```

4.4. **Fonctions opérant sur les tableaux.**

- Si `M` est un tableau d'un certain profil, et `x` est une valeur scalaire, la commande `M=x` affecte la valeur x à tous les coefficients de M.
- Si `M` et `N` sont deux tableaux de mêmes profils, `M+N` et `M*N` produiront les tableaux obtenus en effectuant les sommes et produits des deux tableaux *coefficient par coefficient*.
- On peut appliquer une fonction prédéfinie dans le langage à tous les coefficients d'un tableau ; ainsi `sin(M)` est le tableau dont les coefficients sont les sinus des coefficients de `M`.
- `sum(M)` rend la somme de tous les éléments du tableau M.
- `product(M)` rend le produit de tous les éléments du tableau M.
- `maxval(M)` (resp. `minval(M)`) rend le plus grand (resp. plus petit) élément du tableau M. De même `maxval(M,M>N)` (resp. `minval(M,M>N)`) rend le plus grand (resp. plus petit) élément, parmi les éléments du tableau M qui sont strictement supérieurs à l'élément correspondant du tableau N.
- Si `M` est un tableau de dimension n, `maxloc(M)` (resp. `minloc(M)`) rend un tableau linéaire de longueur n, décrivant la position de la première occurence du plus grand (resp. plus petit) élément du tableau M.
- `where(a<0) a=-a` transforme tous les éléments négatifs du tableau `a` en leur opposé

- `if (all(a>0))  a=0` transforme `a` en le tableau nul si tous ses coefficients sont > 0
- `count(a>0)` rend le nombre de coefficients de a qui sont strictement positifs
- `if (any(a>0))  a=0` transforme `a` en 0 si l'un au moins de ses coefficients est > 0
- `M=cshift(M,3,1)` effectue une permutation circulaire des lignes de M en décallant chaque ligne de 3 places. Le dernier argument est inutile dans le cas d'un tableau linéaire.
- Si `a` et `b` sont deux tableaux de même profil, `max(100,a,b)` produit un tableau de même profil que `a` et `b`, sont les coefficient sont les $\max(100, a_i, b_i)$.
- `matmul(a,b)` effectue le produit matriciel de `a` et `b` dans les cas suivants :
  - si `a` est de profil (m, n) et `b` est de profil (n, p) le résultat est de profil (m, p) ; c'est le produit de la matrice a par la matrice b,
  - `a` est de profil (m, n) et `b` est de profil (n) le résultat est de profil (m) ; c'est le produit à gauche du vecteur colonne b par la matrice a,
  - `a` est de profil (m) et `b` est de profil (m, n) le résultat est de profil (n) ; c'est le produit à droite du vecteur ligne a par la matrice b,
  - dans tout autre cas cela donnera une erreur.
- `dot_product(a,b)` calcule le produit scalaire de deux tableaux linéaires de même taille

4.5. **Allocation dynamique.** Au moment de lancer un programme, on ne connait pas forcément l'étendues des tableaux qui seront utilisés. Fortran 90 nous fournit un mécanisme pour contourner ce problème : l'allocation dynamique

```
programm test
  implicit none
  integer ::n
  real, allocatable :: tab(:,:),vect(:)
  read (*,*) n
  allocate (tab(n,n),vect(n))
  tab = reshape ( (/ (i,i=1,n**2) /) , (/ n,n /) )
  vect = (/ (i,i=1,n) /)
  ...
  deallocate (tab,vect)
  ...
end program test
```

Dans ce programme, on définit un tableau à deux dimensions et un vecteur à une dimension, dont on ne connait pas a priori la taille. Une fois que l'utilisateur aura rentré un entier n, le programme réservera des espaces mémoire pour une matrice de taille $n \times n$ et pour un vecteur de taille n (instruction `allocate`) et les allouera aux variables `tab` et `vect`. La suite du programme remplit `tab` et `vect`, puis les utilise. A la fin de l'utilisation, le programme relache la mémoire réservée (instruction `deallocate`).

5. Structures de contrôle

Comme tout langage structuré, fortran propose différents types de structures de contrôle, qui sont toutes rédigées de la façon suivante

```
mot-cle
  instructions
end mot-cle
```

où `mot-cle` sera suivant les cas `if`, `select`, `do`. Pour plus de clarté dans la programmation, il est possible de nommer les boucles de la façon suivante :

```
nom-de-la-boucle : mot-cle
   instructions
end mot-cle nom-de-la-boucle
```

ceci est particulièrement intéressant en cas de boucles imbriquées.

5.1. **Le bloc IF.**

```
if (expressionBooleenne1 ) then
  bloc1
elseif (expressionBooleenne2 ) then
  bloc2
   ...
else
  dernierBloc
end if
```

Les blocs `elseif` et `else` sont optionnels ; on peut mettre autant de blocs `elseif` que l'on veut. Chacun des blocs est une suite d'instruction fortran.

5.2. **Le bloc SELECT CASE.**

```
select case (expression)
  case liste-de-valeurs-1
    bloc1
  case liste-de-valeurs-2
    bloc2
  ...
  case liste-de-valeurs-n
    blocn
end select
```

où expression est une expression entière ou logique, comme par exemple la sequence suivante, qui en fonction du numéro du mois, indique le nombre de jours :

```
select case (mois)
  case (4,6,9,11)
    nb_jours=30
  case (1,3,5,7,8,10,12)
    nb_jours=31
  case 2
    nb_jours=28
endselect
```

5.3. **Les boucle DO.** Elles sont de la forme :

```
do controle-de-boucle
  bloc
end do
```

où `controle-de-boucle` peut

- ne pas exister : on a une boucle infinie ; il faut donc qu'à l'intérieur de la boucle, il y ait une instruction de sortie.
- être de la forme : `i=1,10` (parcourt les entiers de 1 à 10), ou `i=1,100,5` (parcourt les entiers de 1 à 100, par pas de 5). Dans ces exemples i doit être une variable entière, préalablement déclarée.

– être de la forme : `while (condition-logique)` : la boucle se poursuit tant que la condition est vérifiée.

5.4. **L'instruction WHERE.** Cette instruction permet d'examiner un tableau coefficient par coefficient, et d'effectuer une opération sur les coefficients du tableau qui vérifient une certaine condition. La syntaxe est :

```
where condition
  bloc_d_operations
elsewhere autre-condition
  autre_operation
end where
```

La clause `elsewhere` est facultative (comme pour `elseif`).
Voici un exemple ; ici `a` est un tableau de réels :

```
  where a /= 0.
    a=1./a
  elsewhere
    a=1.
  end where
```

La commande `where` permet d'appliquer des opérations compliquées, soumises à des conditions globalement à un tableau, sans effectuer de boucle ; elle est un substitut efficace à l'utilisation combinée de `do` et `if`.

5.5. **Pseudo-boucles.** Il s'agit d'une construction jouant le même rôle qu'une boucle `do`, qu'on peut utiliser dans un constructeur de tableau ou dans une instruction `write`. la syntaxe générale est

```
(liste de termes, indice=min,max,pas)
```

le pas est optionnel ; s'il n'est pas indiqué, le pas est pris égal à 1. Exemples :

– `write (*,*) (i**2,i=0,100,3)` imprime la liste des carrés des multiples de 3 inférieurs à 100.
– `a=reshape( (/ 1,((0,i=1,5),1,j=1,4) /),(/5,5/))` construit la matrice identité de taille 5×5 et l'affecte à la variable `a`.

5.6. **Sorties de boucle.** La commande `exit` permet de sortir d'une boucle. Par exemple :

```
i=1
do
  if (log(i)>5) then
    exit
  else
    i=i+1
  end if
end do
```

Il peut être utile d'utiliser ici des boucles nommées. Ainsi, dans le programme

```
boucle1: do
  ...
  boucle2: do
    ...
    if condition then
      exit boucle1
```

```
    end if
  end do boucle2
end do boucle1
```

quand la condition du test est vérifiée, on sort directement de la boucle principale `boucle1`, et non seulement de la boucle interne `boucle2`.
La commande `cycle1` permet à l'inverse d'interrompre l'exécution d'un bloc d'instruction dans une boucle, et de passer directement à l'itération suivante. Ainsi, dans le programme

```
do
  bloc-d-instructions1
  if condition then
    cycle
  end if
  bloc-d-instructions2
end do
```

lorsque la condition est vérifiée, le programme saute l'exécution du deuxième bloc d'instructions, et passe directement à l'itération suivante.

6. Structuration d'un programme en briques séparées

Dès qu'on écrit des programmes de taille un peu importante, on a envie de découper les programmes en différentes unités logiques. Ce choix rend les programmes plus lisibles et permet par ailleurs de répartir le développement d'un projet entre plusieurs développeurs, dont chacun est responsable du développement d'une unité.

Par aileurs, un morceau de programme pourra être utilisé plusieurs fois à l'intérieur d'un programme donné, ou aussi être mutualisé dans différents programmes.

Les sous-programmes seront de deux types : procédures (`subroutine`) ou fonctions (`function`).

6.1. La notion de sous-programme.

On veut calculer $\displaystyle\int_0^x e^{-x^2/2}\frac{dx}{\sqrt{2\pi}}$ par la méthode des rectangles ; on va donc avoir à utiliser un grand nombre de fois la fonction $x \mapsto e^{-x^2/2}$. Les deux fichiers de programme suivants effectuent ce calcul, le premier en faisant appel à une fonction, le second en utilisant une procédure. Regardez attentivement les programmes pour comprendre la différence.

1- Programme utilisant une fonction :

```
program int
  implicit none
  real :: s,x,f
  integer :: i
  write(*,*) 'rentrer un nombre positif'
  read(*,*) x
  s=0
  do i=0,1000
     s=s+f(x*i/1000.)
  end do
  write(*,*) s*(x/1000.)
end program int

function f(x)
  implicit none
```

```
  real :: x,f,pi
  pi=4*atan(1.)
  f=exp(-x**2/2)/sqrt(2*pi)
end function f
```

On notera que `f` est une variable utilisée dans la fonction, et doit être définie dans le corps de la fonction.

2- Programme utilisant une procédure :

```
program int
  implicit none
  real :: s,x,f
  integer :: i
  write(*,*) 'rentrer un nombre positif'
  read(*,*) x
  s=0
  do i=0,1000
     call mafonction(x*i/1000.,f)
     s=s+f
  enddo
  write(*,*) s*(x/1000.)
end program int

subroutine mafonction(x,f)
  implicit none
  real :: x,f,pi
  pi=4*atan(1.)
  f=exp(-x**2/2)/sqrt(2*pi)
end subroutine mafonction
```

6.2. **Fonctions.** Une fonction fortran correspond exactement à l'objet mathématique que nous connaissons bien. Elle est définie par le mot clé `function`. Ainsi la fonction $f : \mathbb{R}^3 \to \mathbb{R}$, $(a, b, c) \mapsto a + bc$ peut être programmée comme suit :

```
function f(a,b,c)   ! on met ici toutes les variables d'entrées de la fonction
  implicit none
  real :: a,b,c,f   ! on définit les variables d'entrées ET
                    ! la variable de sortie
  ! on met ici toutes les instructions nécessaires au calcul de f(a,b,c)
  f= a+b*c  ! on termine par affecter la valeur de sortie.
end function f
```

La fonction f pourra alors être appelée dans le programme par une instruction `x=f(a,b,c)`.

6.3. **Procédures.** Une procédure n'a pas de sortie identifiée. Elle prend un certain nombre de variables en argument et effectue des opérations, qui peuvent éventuellement modifier les arguments. Voici différents exemples pour comprendre :

```
subroutine trinome1(x,a,b,c)
  implicit none
  real :: x,a,b,c
  x=a*x^2+b*x+c
end subroutine mafonction
```

Ici les 4 variables ont des rôles différents : a, b et c restent inchangées, alors que x est modifiée par la procédure. dans le second exemple, aucune variable n'est modifiée, mais une action est effectuée (ici un affichage).

```
subroutine trinome2(x,a,b,c)
  implicit none
  real :: x,a,b,c
  write(*,*) "la valeur du trinome est", a*x^2+b*x+c
end subroutine mafonction
```

Dans la définition des variables d'une procédure, il est raisonnable de rajouter l'attribut `intent`, qui peut prendre les valeurs `in` (pour une variable uniquement d'entrée), `out` (pour une variable uniquement de sortie,`inout` (pour une variable à la fois d'entrée et de sortie). Ainsi dans `trinome1`, on peut indiquer

```
  real, intent(in) :: a,b,c
  real, intent(inout) :: x
```

L'utilisation de cet attribut est optionnel, mais permet au compilateur de détecter des erreurs dans les programmes.

On fait appel à une procédure dans un programme en utilisant le mot-clé `call`, par exemple :

```
call  trinome2(x,a,b,c)
```

6.4. **Modules.** Un module est un fichier qui contient des définitions de procédures, de fonctions, de constantes. On peut ensuite inclure un module dans un programme, et celui-ci fait alors appel à tout le contenu du module. Donnons un exemple bref. Je construis un module (`moduleAlgLin`) qui contiendra un certain nombre de primitives de traitement de matrices (dans notre cas, il y aura une procédure `affichage` qui affiche la matrice à l'écran, une fonction `trace` qui calcule la trace d'une matrice, ainsi que la définition de la constante `pi`). On crée un fichier `moduleAlgLin.f90` contenant :

```
module moduleAlgLin
  implicit none
  real, parameter :: pi=3.1415927
contains
  subroutine affichage(x)
    implicit none
    real,dimension(:,:) :: x
    integer :: i,j
    do i=1,size(x,1)
       write(*,*) (x(i,j), j=1,size(x,2))
    end do
  end subroutine affichage
  function trace(x)
    implicit none
    real :: trace
    real,dimension(:,:) :: x
    integer :: i
    trace = 0
    do i=1,size(x,1)
       trace = trace + x(i,i)
    end do
  end function trace
```

```
end module moduleAlgLin
```

puis un fichier `main.f90`, contenant le programme principal :

```
program main
  use moduleAlgLin ! Appel du module moduleAlgLin
  implicit none
  real,dimension(10,5) :: x
  call random_number(x)
  call affichage(x)
  write (*,*) "trace = ", trace(x)
  write (*,*) "pi = ", pi
end program main
```

La ligne `use moduleAlgLin` faisant appel au module doit impérativement avant la ligne `implicit none`. Il ne reste plus qu'à compiler l'ensemble :

```
gfortran moduleAlgLin.f90 main.f90
```

On fera attention de mettre le (ou les) fichier(s) de module en premier, car ils doivent être compilés avant le programme qui y fait appel. L'usage de modules est généralement vivement recommandé dès qu'on écrit des programmes de taille un peu importante, car il est commode et permet de détecter de nombreuses erreurs dès la compilation.
L'utilisation de modules est à conseiller dans la mesure du possible.

6.5. **Compléments sur la compilation.** On a déjà rencontré le cas d'un programme découpé en plusieurs morceaux. Considérons ici le cas d'un programme décomposé en trois parties logiques :
partie1.f90, le programme principal, faisant appel à un module `toto` :

```
program main
  use toto
  ...
  call procedure1()
  ...
end program main
```

partie2.f90, contenant le module `toto` :

```
module toto
  use module tata
  contains
    subroutine procedure1()
    ...
    end subroutine procedure1
end module
```

partie3.f90, contenant le module `tata`, utilisé par le module `toto` :

```
module tata
  ...
end module tata
```

On peut effectuer la compilation en tapant

```
gfortran partie3.f90 partie2.f90 partie1.f90 -o total.exe
```

On notera qu'on a mis d'abord *partie3.f90*, utilisé par *partie2.f90*, puis *partie2.f90*, qui est nécessaire pour *partie1.f90*, et enfin *partie1.f90*. Une compilation dans un autre ordre aurait produit une erreur.

On peut aussi effectuer des compilations séparées :
`gfortran -c partie1.f90 -o partie1.o` (création du fichier objet partie1.o)
`gfortran -c partie2.f90 -o partie2.o` (création du fichier objet partie2.o)
`gfortran -c partie3.f90 -o partie3.o` (création du fichier objet partie3.o)
`gfortran partie1.o partie2.o partie3.o -o total.exe` (édition de lien : crée un fichier exécutable). Ici l'ordre n'est plus important.

Par ailleurs, la compilation `gfortran -c partie2.f90` produit un fichier `toto.mod`. Il s'agit d'un fichier binaire qui peut alors être utilisé pour la compilation directe du programme pricipal `gfortran partie1.f90`, à condition que `toto.mod` et `partie1.f90` se trouvent dans le même répertoire. On peut ainsi fournir à quelqu'un un module, en lui donnant le fichier `toto.mod` correspondant, ce qui lui permettra d'effectuer des compilations à l'aide du module, mais sans lui fournir les sources du module.

6.6. **Complément : déclaration d'interfaces de sous-programmes.** Quand on fait des programmes de taille conséquente, on utilise généralement un programme, qui fait appel à de très nombreuses procédures externes. Pendant l'étape de compilation, le programme et les procédures seront compliées séparément et le compilateur ne vérifiera donc pas si les procédures sont compatibles avec le programme (bon nombre de variables, type de ces variables, taille des tableaux). Les erreurs apparaitront seulement à l'éxécution du programme. Pour remédier à cela, fortran propose la notion d'interface : à l'intérieur du programme principal, après les déclaration et avant le corps du programme, on met une section de déclaration d'interface de la façon suivante :

```
program programmePrincipal
  implicit none
  real :: ....  // toutes les déclarations de variables
  interface
    subroutine sousProgramme1(variable1, variable2)
       real :: variable1
       integer, dimension(10,10) :: variable2
    end subroutine  sousProgramme1
    subroutine sousProgramme2(x,y)
       real :: x,y
    end subroutine  sousProgramme2
    ... // et de meme avec toutes les procedures
  end interface
  // et maintenant on met le corps du programme
end program programmePrincipal

subroutine sousProgramme2(x,y)
  real :: x,y,z
  z=x
  x=y
  y=z
end subroutine  sousProgramme2
```

On voit ici que dans la section `interface`, on met toutes les procédures externes, en indiquant pour chacune leurs variables *externes*. Ainsi, dans `sousProgramme2`, on a déclaré x et y qui sont des variables externes de la procédure, mais pas z qui est interne à la procédure.
L'utilisation de déclaration d'interface est une pratique un peu contraignante, mais qui permet de confier au compilateur une partie du travail de débogage.

7. Tableau de taille indéterminée dans les sous-programmes

Quand on utilise une procédure prenant un tableau comme un argument, ce tableau peut avoir une dimension fixée à l'avance. Par exemple :

```
subroutine sousProgramme (tableau)
  implicit none
  real, dimension(45,25) :: tableau
  ...
end subroutine sousProgramme
```

Mais on peut aussi vouloir rentrer un tableau de taille variable. Pour cela, on a plusieurs solutions :

7.1. **Rentrer comme arguments le tableau et la dimension de ce tableau.** Voici un exemple :

```
subroutine sousProgramme (tableau,n,p)
  implicit none
  integer :: n,p
  real, dimension(n,p) :: tableau
  ...
end subroutine sousProgramme
```

Ceci était la seule solution proposée par fortran 77 ; de ce fait, on retrouve ce style de programmation dans les bibliothèques de programmes initialement développées en fortran 77, comme par exemple Lapack.

7.2. **Utiliser un tableau de taille indéterminée.** Une procédure définie dans un module peut utiliser comme argument un tableau de taille indéterminée en utilisant une déclaration de la forme

```
real, dimension(:,:) :: tableau
```

qui signifie qu'on utilise une variable de type tableau de dimension 2, et d'étendue indéterminée. Voici comment on peut mettre ça en pratique :

```
program monProgramme
  use monModule
  implicit none
  real :: T(100,100)
  ...
  call sousProgramme (T)
  ...
end program monProgramme

module monModule
  contains
    subroutine sousProgramme (tableau)
      implicit none
      real, dimension(:,:) :: tableau
      ...
    end subroutine sousProgramme
end module monModule
```

Comme le module est inclus dans le programme principal, grâce à la commande `use monModule`, le compilateur sait retrouver la taille de T lors de la compilation et réserver l'espace mémoire

nécessaire. Dans le cas d'une procédure externe, ne faisant pas partie d'un module, le compilateur n'a aucune information sur la procédure au moment où il compile le programme principal. On a alors besoin de définir explicitement l'interface de la procédure, mais on ne développera pas cet aspect ici.

7.3. **Tableau automatique.** Une procédure peut utiliser comme variable interne un tableau de taille indéterminée a priori, et dépendant de la taille des arguments de la procédure. Exemple :

```
subroutine sousProgramme (a,b)
  implicit none
  real, dimension(:,:) :: a,b
  real :: c(size(a,1)), d(size(b))
  ...
end subroutine sousProgramme
```

Ici `c` et `d` sont des variables internes dont la taille est fixée automatiquement par les tailles des arguments `a` et `b`.

8. Entrées-Sorties

On a utilisé précédemment des commandes de la forme `write (*,*) "bonjour"` qui affiche à l'écran la chaîne de caractères « bonjour », ou `read (*,*) x`, qui lit un réel entré au clavier et le met dans la variable `x`. La forme plus générale d'une telle commande est

```
write (numero,format) expression
```

(ou de même avec `read`). Dans une telle commande

- `expression` est une liste de variables ou d'expressions, séparées par des virgules, comme `"le carre de ",i, "est egal a ", i**2`, ou `(i**2,i=1,20)` (qui donne la liste des carrés des entiers de 1 à 20)
- `numero` est le numéro d'un périphérique préalablement défini. le caractère * déja utilisé désigne l'entrée standard (pour `read`, le clavier) ou la sortie standard (pour `write`, l'écran). Si on veut lire (ou écrire) dans un fichier, on commence par associer un numéro à ce fichier, comme par exemple
  ```
  open(unit=5,file="nom_du_fichier",status="new")
  do i=1,10
    write (5,*) "le carre de ",i, "est egal a ", i**2
  enddo
  ```
- `format` décrit la façon dont l'entrée ou la sortie est formatée. Le symbole * signifie ici « sans formatage ». Nous n'aurons pas souvent l'occasion d'utiliser d'autre format que le format standard.

8.1. **Les commandes `open` et `close`.** Si on souhaite accéder à un fichier nommé "toto", on lui attribue un numéro (27 dans l'exemple), et on précise son statut (`new` si il s'agit d'un fichier qui n'existe pas encore, ou `old` pour un fichier déja créé au préalable, et `unknown` pour autoriser les deux cas. Ceci donne une séquence du genre :

```
open(unit=27, file='toto',status='unknown')
write (27,*) (i,i**2,i=1,5)
close(27)
```

8.2. **Écriture formatée.** On peut préciser la façon dont les résultats du programme doivent être écrits dans le fichier de sortie. Par exemple

```
 write (27,fmt='(i5)') x     ! x est un entier écrit sur 5 caractères
 write (27,fmt='(f10.5)') x  ! x est un réel  écrit avec 10 caractères,
                             ! dont 5 chiffres après la virgule
```

Les instructions de format sont de la forme :

- `fmt='(nIm)'` : n entiers de m chiffres
- `fmt='(nFm.d)'` : n réels sous forme décimale, de m caractères, en comptant le signe ., dont d chiffres après la virgule
- `fmt='(nEm.d)'` : n réels sous forme scientifique, de m caractères, en comptant le signe ., dont d chiffres après la virgule
- `fmt='(nAm)'` : n chaînes de m caractères.

Par exemple, les instructions

```
character(len=10) :: prenom, nom
integer :: age
prenom='Albert'
nom='Dugenou'
age='103'
write (*,fmt='(2A10,1I3)') prenom, nom, age
```

rendront :

```
albert----Dugenou---103
```

où le signe - a été utilisé pour symboliser un espace. De telles écritures formatées seront utiles pour rendre des résultats lisibles par un être humain (tableau, avec des colonnes bien alignées).

9. Types dérivés

On peut définir des types de données plus élaborés, à partir des types élémentaires définis dans fortran. Voici un exemple : dans un cabinet de vétérinaire, on enregistre les animaux en notant certaines informations. On définit donc un type « animal »

```
type animal
  character(len=20) :: nom,race,couleur
  integer :: age
  logical, dimension(6) :: vaccinations
end type animal
```

Dans un programme, on définira donc des variables de type animal

```
type(animal) :: ponpon, titus, kiki
```

Une telle variable sera ensuite affectée par des instructions du type

```
ponpon%nom = "ponpon"
ponpon%race = "chat de gouttiere"
ponpon%couleur = "gris sale"
ponpon%age = 6
ponpon%vaccinations = (/ .true.,.true.,.true.,.false.,.true.,.false. /)
```

On pourra aussi utiliser un tableau contenant tous nos enregistrements d'animaux :

```
type(animal),dimension(50) :: animaux
```

et récupérer des informations dans cette base de données ; ainsi la commande

```
write(*,*) animaux(31)%age, animaux(31)%vaccinations(2)
```

affichera l'age de l'animal enregistré en 31-ème position dans le tableau et l'état de sa vaccination relativement à la deuxième maladie.

10. Visualiser le résultat d'un calcul

Le résultat d'un calcul apparaît souvent comme un tableau de nombres ; pour visualiser ce résultat, une solution est d'écrire le résultat dans un fichier, puis d'utiliser un outil graphique pour visaliser graphiquement le contenu du fichier. Une solution est d'utiliser le logiciel **gnuplot**. Pour cela, on lance la commande `gnuplot` dans une fenêtre de terminal unix. On se retrouve alors devant une invite `gnuplot>`. On est alors à l'intérieur d'une session de gnuplot, dans laquelle on peut lancer des commandes, comme dans les deux exemples qui suivent. On quitte gnuplot en tapant `exit`. Pour plus de renseignements sur gnuplot, taper `help` à l'intérieur de la session gnuplot.

10.0.1. Tracé du graphe d'une fonction d'une variable réelle : on veut tracer le graphe de $x \mapsto \sin(x)$ pour $x \in [0, 10]$. Commençons par par écrire un programme fortran :

```
program affichage
  implicit none
  integer :: i
  open(unit=10, file='data',action='write')
  do i= 1,1000
     write(10,*) i/100,sin(i/100)
  end do
end program affichage
```

On compile ensuite ce programme, et on l'exécute. Il crée un fichier texte nommé « data », qui contient 1000 lignes, dont les premières sont

```
  9.9999998E-03  9.9998331E-03
  2.0000000E-02  1.9998666E-02
  2.9999999E-02  2.9995499E-02
  3.9999999E-02  3.9989334E-02
  5.0000001E-02  4.9979169E-02
  5.9999999E-02  5.9964005E-02
```

On visualise le graphe de la fonction en tapant dans une session gnuplot la commande `plot 'data' with lines`.

10.0.2. Tracé du graphe d'une fonction de deux variables réelles : on veut tracer le graphe de $(x, y) \mapsto \sin x \cos y$ sur $[0, 10] \times [0, 10]$. Commençons par par écrire un programme fortran :

```
program dessin3D
  implicit none
  integer :: i,j
  integer, parameter :: n=20
  open (unit=5,file="toto",status="unknown")
  do i=0,n
     do j=0,n
        write (5,*) real(i)*10/n,"  ",real(j)*10/n,"  ", &
            sin(real(i)*10/n)*cos(real(j)*10/n)
     end do
     write (5,*) " " // pour insérer une ligne blanche
  end do
end program dessin3D
```

On compile ensuite ce programme, on l'exécute, et on visalise le résultat qui a été stocké dans le fichier `toto` en en lançant dans gnuplot la commande `splot "toto" with lines`. On notera

qu'on a introduit des lignes blanches dans le fichier des données pour le découper en blocs. Cela permet d'obtenir un graphe avec un quadrillage rectangulaire. Regardez ce qui se passe si on n'insère pas ces lignes blanches !

On notera que ce qui permet de visualiser une surface est la structure en fil de fer (quadrillage). Si ce quadrillage est trop sérré, on ne verra plus qu'une tache noire informe. Il importe donc d'utiliser un tableau de nombres de taill raisonnable (20×20 par exemple, même si dans le calcul numérique, on a travaillé pour des raisons de précision avec une matrice beaucoup plus grosse (peut être 1000×1000). Il faudra ainsi demander à notre programme de n'afficher qu'une ligne (rep. colonne) toutes les 50.

11. Quelques fonctions internes de fortran

`abs(x)`	valeur absolue de x (réel ou entier)
`floor(x)`	partie entière du réel x
`sin(x)`, `cos(x)`, `tan(x)`	fonctions trigonométriques
`asin(x)`, `acos(x)`, `atan(x)`	fonctions trigonométriques inverses
`exp(x)`, `log(x)`	exponentielle et logarithme népérien
`conjg(z)`	complexe conjugué du complexe z
`min(a1,a2,...)`	le plus petit des réels $a_1, a_2, \ldots$
`modulo(a,p)`	reste de la division euclidienne de a par p
`sign(a,b)`	fournit $abs(a)$ si b est positif et $-abs(a)$ sinon
`dot_product(a,b)`	rend $\sum a_i b_i$, pour a et b des tableaux de rang 1 de même taille
`matmul(M,N)`	produit matriciel de deux matrices ou vecteurs de tailles compatibles
`transpose(M)`	transposée de la matrice M
`random_number(A)`	procédure qui écrit dans un tableau A des nombres aléatoires
`size(A,i)`	taille du tableau A dans la i-ème dimension

12. Utilisation de la bibliothèque de procédures LAPACK

De nombreuses méthodes d'algèbre linéaire ont déjà été programmées, et sont disponibles dans les bibliothèques de procédures BLAS et LAPACK. Pour utiliser des procédures de ces bibliothèques, il faut, à la compilation, taper une commande du type

```
gfortran -lblas -llapack monprogramme.f90 -o monprogramme.exe
```

La feuille ci-jointe fournit un certain nombre de ces procédures. Par exemple, la procédure `sgesv( )` permet de résoudre un système linéaire de la forme $AX = B$. Pour comprendre précisémment comment marche une telle procédure, il faut regarder la page de manuelle, en tapant dans une fenêtre de terminal une commande comme :

```
man sgesv
```

On obtient alors une réponse :

```
NAME
       SGESV - compute the solution to a real system of linear equations A * X = B,

SYNOPSIS
       SUBROUTINE SGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )

           INTEGER        INFO, LDA, LDB, N, NRHS

           INTEGER        IPIV( * )
```

```
          REAL          A( LDA, * ), B( LDB, * )

PURPOSE
      SGESV computes the solution to a real system of linear equations A * X = B,
     where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

      The LU decomposition with partial pivoting and row interchanges is used
      to factor A as
         A = P * L * U,
      where P is a permutation matrix, L is unit lower triangular,
      and U is upper triangular.  The factored form of A is then
      used to solve the system of equations A * X = B.

ARGUMENTS
      N       (input) INTEGER
              The number of linear equations, i.e.,
      the order of the matrix A.  N >= 0.

      NRHS    (input) INTEGER
              The number of right hand sides, i.e.,
      the number of columns of the matrix B.  NRHS >= 0.

      A       (input/output) REAL array, dimension (LDA,N)
              On entry, the N-by-N coefficient matrix A.
      On exit, the factors L and U from the factorization A = P*L*U;
      the unit diagonal elements of L are not stored.

      LDA     (input) INTEGER
              The leading dimension of the array A.  LDA >= max(1,N).

      IPIV    (output) INTEGER array, dimension (N)
              The pivot indices that define the permutation matrix P;
              row i of the matrix was interchanged with row IPIV(i).

      B       (input/output) REAL array, dimension (LDB,NRHS)
              On entry, the N-by-NRHS matrix of right hand side matrix B.
              On exit, if INFO = 0, the N-by-NRHS solution matrix X.

      LDB     (input) INTEGER
              The leading dimension of the array B.  LDB >= max(1,N).

      INFO    (output) INTEGER
              = 0:  successful exit
              < 0:  if INFO = -i, the i-th argument had an illegal value
              > 0:  if INFO = i, U(i,i) is exactly zero.
              The factorization has been completed,
              but the factor U is exactly singular,so the solution
              could not be computed.
```

Vous trouverez dans la section suivante un document permettant de choisir la procédure Lapack adaptée à votre problème.

13. LAPACK QUICK REFERENCE GUIDE

BLAS and LAPACK guides available from http ://www.ews.uiuc.edu/~mrgates2/.
Reference : *LAPACK Users Guide* from http ://www.netlib.org/lapack/faq.html

13.1. **Prefixes.** Each routine has a prefix, denoted by a hypen - in this guide, made of 3 letters xyy, where x is the data type, and yy is the matrix type.

Data type			
s	single	d	double
c	complex single	z	complex double

Matrix type	**full**	**banded**	**packed**	**tridiag**	**generalized problem**
general	ge	gb		gt	gg
symmetric	sy	sb	sp	st	
Hermitian	he	hb	hp		
SPD / HPD	po	pb	pp	pt	
triangular	tr	tb	tp		tg
upper Hessenberg	hs				hg
trapezoidal	tz				
orthogonal	or		op		
unitary	un		up		
diagonal	di				
bidiagonal	bd				

For symmetric, Hermition, and triangular matrices, elements below/above the diagonal (for upper/lower respectively) are not accessed. Similarly for upper Hessenberg, elements below the subdiagonal are not accessed.

Packed storage is by columns. For example, a 3×3 upper triangular is stored as

$$[\ \underbrace{a_{11}}\ \ \underbrace{a_{12}\ a_{22}}\ \ \underbrace{a_{13}\ a_{23}\ a_{33}}\ ]$$

Banded storage puts columns of the matrix in corresponding columns of the array, and diagonals in rows of the array, for example :

$$\begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{bmatrix} \begin{matrix} \text{1st diagonal} \\ \text{2nd (main) diagonal} \\ \text{3rd diagonal} \\ \text{4th diagonal} \end{matrix}$$

Bi- and tridiagonal matrices are stored as 2 or 3 vectors of length n and $n-1$.

13.2. **Drivers.** Drivers are higher level routines that solve an entire problem.

Linear system, solve $Ax = b$.

`-sv` — solve
`-svx` — expert ; also $A^T x = b$ or $A^H x = b$, condition number, error bounds, scaling
Matrix types [General ge, gb, gt ; SPD po, pp, pb, pt ; Symmetric sy, sp, he, hp]

Linear least squares, minimize $\|b - Ax\|_2$.
`-ls` — full rank, $\text{rank}(A) = \min(m, n)$, uses QR.
`-lsy` — rank deficient, uses complete orthogonal factorization.
`-lsd` — rank deficient, uses SVD.
Matrix types [General ge]

Generalized linear least squares.
Minimize $\|c - Ax\|_2$ subject to $Bx = d$.
`-lse` — B full row rank, matrix $\begin{bmatrix} A \\ B \end{bmatrix}$ full col rank.

Minimize $\|y\|_2$ subject to $d = Ax + By$.
`-glm` — A full col rank, matrix $\begin{bmatrix} A & B \end{bmatrix}$ full row rank.
Matrix types [General gg]

Eigenvalues, solve $Ax = \lambda x$.
Symmetric
`-ev` — all eigenvalues, [eigenvectors]
`-evx` — expert ; also subset
`-evd` — divide-and-conquer ; faster but more memory
`-evr` — relative robust ; fastest and least memory
Matrix types [Symmetric sy, sp, sb, st, he, hp, hb]

Nonsymmetric
`-ev` — eigenvalues, [left, right eigenvectors]
`-evx` — expert ; also balance matrix, condition numbers
`-es` — Schur factorization
`-esx` — expert ; also condition numbers
Matrix types [General ge]

Generalized eigenvalue, solve $Ax = \lambda Bx$
Symmetric, B SPD
`-gv` — all eigenvalues, [eigenvectors]
`-gvx` — expert ; also subset
`-gvd` — divide-and-conquer, faster but more memory
Matrix types [Symmetric sy, sp, sb, he, hp, hb]

Nonsymmetric
`-ev` — eigenvalues, [left, right eigenvectors]
`-evx` — expert ; also balance matrix, condition numbers
`-es` — Schur factorization
`-esx` — expert ; also condition numbers
Matrix types [General gg]

SVD singular value decomposition, $A = U\Sigma V^H$
`-svd` — singular values, [left, right vectors]
`-sdd` — divide-and-conquer ; faster but more memory
Matrix types [General ge]

Generalized SVD, $A = U\Sigma_1 Q^T$ **and** $B = V\Sigma_2 Q^T$
`-svd` — singular values, [left, right vectors]
Matrix types [General gg]

14. Trouver des erreurs dans un programme

Lors de la compilation, le compilateur transforme le fichier texte contenant votre programme en un fichier éxécutable et peut pendant cette phase détecter un certain nombre d'erreurs.

Cependant, d'autres erreurs ne seront pas détectées par le compilateur, et n'apparaitront que lors de l'éxécution, et peut être seulement sous certaines conditions. Voici quelques exemples de fichiers contenant des erreurs :

14.1. **Une erreur de syntaxe.**

```
program a
  implicit none
  real : x  ! on a oublié un :
  x=1.5
end program a
```

donne lieu à une erreur lors de la compilation :

```
In file a.f90:3

  real : x
     1
Error: Syntax error in data declaration at (1)
 In file a.f90:4

  x=1.5
  1
Error: Symbol 'x' at (1) has no IMPLICIT type
```

On constate que le compilateur voit deux erreurs. La première est la faute de syntaxe. La deuxième est une conséquence : à cause de l'erreur, la variable `x` n'a pas été déclarée. On notera également que le compilateur cherche à montrer l'endroit où il détecte la faute en le désignant par un `1`.

14.2. **Une division par zéro.**

```
program a
implicit none
  real :: x
  x=1.5/(1-1)
end program a
```

donne lieu à

```
In file a.f90:4

  x=1.5/(1-1)
      1
Error: Division by zero at (1)
```

14.3. **Une division par zéro non détectée à la compilation.**

```
program a
  implicit none
  real :: x
  x=1
  x=1/(1-x**2)
  write (*,*) x
end program a
```

Le compilateur ne peut détecter une erreur, car celle ci apparait comme résultat d'un calcul. L'exécution se déroule noramalement, mais rend `+Infinity`. Dans d'autres cas, ce type d'erreur amène à une erreur d'exécution.

14.4. **Presque la même chose.**

```
program a
  implicit none
  real :: x
  x=1
  x=sqrt(-x)
  write (*,*) x
end program a
```

Ici le résultat à l'exécution est `NaN` (Not a Number).

14.5. **Une erreur de type.**

```
program a
  implicit none
  real :: x ! ce devrait etre un entier !
  do x=1,10
     write(*,*) x
  end do
end program a
```

La compilation va à son terme, mais donne un avertissement :

```
In file a.f90:4

  do x=1,10
     1
Warning: Obsolete: REAL DO loop iterator at (1)
```

L'exécution se passe bien, mais c'est un coup de chance !

14.6. **Mauvaise taille de tableau.**

```
program a
  implicit none
  real, dimension(10) :: x
  x(11)=1.
end program a
```

La compilation va à son terme, mais donne un avertissement :

```
In file a.f90:4

  x(11)=1.
   1
Warning: Array reference at (1) is out of bounds
```

L'exécution ne donne pas d'erreur, mais on ne peut absolument pas prévoir ce que donnera le programme. A éviter absolument ! dans cet autre exemple, très voisin, l'exécution aboutit à un plantage (`Segmentation fault`), indiquant que le programme a écrit dans un espace mémoire qui n'était pas fait pour ça :

```
program a
  implicit none
  real, dimension(10) :: x
  integer :: i
  i=5
  x(i**2)=1.
end program a
```

15. Un programme commenté : la méthode de la puissance

Cherchons à programmer la méthode de la puissance en fortran. Ecrivons d'abord un sous-programme qui réalise la partie algorithmique.

```
subroutine puissance(M,v,lambda,nb_iter)            !!!! (1)
  implicit none
  double precision, dimension(:,:) :: M             !!!! (2)
  double precision, dimension(:) :: v
  double precision :: lambda,lambda1,prec,norme
  double precision, allocatable,dimension(:) :: w   !!!! (3)
  integer :: i,n,nb_iter
  logical :: test
  n=size(v)
  allocate( w (n) )                                 !!!! (3)
  prec=10**(-15)
  test=.true.
  norme=sqrt(dot_product(v,v))                      !!!! (4)
  v=(1/norme)*v                                     !
  w=matmul(M,v)                                     !
  lambda=dot_product(v,w)                           !
  v=w                                               !
  nb_iter=1
  do while (test .eqv. .true.)                      !!!! (5)
     norme=sqrt(dot_product(v,v))                   !
     v=(1/norme)*v                                  !
     w=matmul(M,v)                                  !
     lambda1=dot_product(v,w)                       !
     test=(abs(lambda-lambda1) > prec*abs(lambda))  !
     lambda=lambda1                                 !
     v=w                                            !
     nb_iter=nb_iter+1                              !
  enddo
  deallocate(w)                                      !!!! (3)
endsubroutine puissance
```

Commentaires :

(1) Cette procédure a 4 variables externes `M, v, lambda, nb_iter`. La matrice M est donnée, ainsi qu'un vecteur initial v. Elle rend la valeur propre la plus grande en module (λ) et le nombre d'itérations quui ont été nécessaires, et affecte à v la valeur d'un vecteur propre associé à λ. Donc v est une variable à la fois d'entrée et de sortie.

(2) Les tableaux M et v (de dimension 2 et 1) sont déclarés avec des tailles non déterminées a priori

(3) Le tableau de dimension 1 w est une variable technique, qui a la même dimension que v. Comme la taille n'est pas fixée a priori, on utilise une allocation dynamique (declaration `allocatable`, et commandes `allocate` et `deallocate`).

(4) On commence par faire une première itération avant de faire tourner la boucle.

(5) Puis on fait tourner la boucle. Le test d'arrêt est que la vriation relative de λ dans deux passages successifs est inférieure à une précision fixée à l'avance.

On écrit ensuite un programme pour appeler ce sous-programme.

```
program test
  implicit none
  integer,parameter :: dim=1000                                  !!!! (1)
  double precision,dimension(dim,dim) :: M
  double precision,dimension(dim) :: v,vv
  double precision :: lambda
  integer :: nb_iter
  interface                                                      !!!! (2)
     subroutine puissance(M,v,lambda,nb_iter)
       double precision, dimension(:,:) :: M
       double precision, dimension(:) :: v
       double precision :: lambda
       integer :: nb_iter
     end subroutine puissance
  end interface
  call random_number(M)
  call random_number(v)
  call puissance(M,v,lambda,nb_iter)
  vv=lambda*v-matmul(M,v)
  write(*,*) "lambda=", lambda
  write(*,*) "norme(Mv-lambda v)=",sqrt(dot_product(vv,vv))
  write(*,*) "nombre d'iterations", nb_iter
endprogram test
```

Commentaires

(1) L'introduction du paramètre `dim` permet de modifier facilement le programme.

(2) Comme la procédure utilise des tableaux de taille non fixée a priori, il faut définir l'interface de cette procédure.

Voici une variante de ce programme, utilisant l'allocation dynamique, permettant de rentrer le paramètre `dim` au cours du programme

```
program test
  implicit none
  double precision,dimension(:,:),allocatable :: M
  double precision,dimension(:),allocatable :: v,vv
  double precision :: lambda
  integer :: nb_iter,dim
  interface
```

```
    subroutine puissance(M,v,lambda,nb_iter)
      double precision, dimension(:,:) :: M
      double precision, dimension(:) :: v
      double precision :: lambda
      integer :: nb_iter
    end subroutine puissance
  end interface
  write(*,*) "entrer la dimension de la matrice"
  read(*,*) dim
  allocate( M(dim,dim), v(dim), vv(dim) )
  call random_number(M)
  call random_number(v)
  call puissance(M,v,lambda,nb_iter)
  vv=lambda*v-matmul(M,v)
  write(*,*) "lambda=", lambda
  write(*,*) "norme(Mv-lambda v)=",sqrt(dot_product(vv,vv))
  write(*,*) "nombre d'iterations", nb_iter
endprogram test
```