Cours de programmation numérique

Introduction au FORTRAN 90

ISIMA 1ère année

J. Truffot, J. Koko, A. Tanguy

ISIMA

Introduction : historique FORTRAN

- **1954** : *IBM Mathematical FORmula TRANslation System*.
- 1958 : deuxième version, ajout des sous-programmes.
- 1972 : FORTRAN 66, premier standard.
- **1980** : **FORTRAN 77**, standard internationnal ISO.
- **1991** : **FORTRAN 90**, nombreuses améliorations.
- 1994 : premiers compilateurs pour FORTRAN 90.
- 1997 : FORTRAN 95, révision mineure.
- 2004 : FORTRAN 2003, programmation orienté objet.

FORTRAN 90 comprend FORTRAN 77 mais il faut parfois le préciser à la compilation.

Chapitre 1:

Généralités

- Caractères usuels
- Commentaires : **!**
- Séparateur d'instruction : **;** ou `fin de ligne`
- Chaîne de caractères : **'** ou **"**

```
x=0                  ! une seule instruction sur une ligne
i=i+1; x=(2*i-1)*h  ! deux instructions sur une ligne
Nom='SATAN'          ! une chaine de caracteres sur delemitee par des '
PNom="Lucifer"       ! une chaine de caracteres sur delemitee par des "
```

Instruction sur plusieurs lignes

- Utilisation du caractère **&**

```
print *,'Montant  HT : ',montant_ht, &
        '          TVA : ',tva        , &
        'Montant TTC : ',montant_ttc
```

- Coupure d'une chaîne de caractères

```
print *,'Entrer un nombre entier &
        &compris entre 10 et 100 '
```


Structure générale d'un programme FORTRAN 90 (format libre)

```
program MONPROG          ! debut du programme
implicit none            ! obligatoire pour eviter
                         !les problemes

! toutes les declarations
...

! instructions executables
...

end program MONPROG   ! fin du programme
```

Les types scalaires

En FORTRAN 90, il existe **5 types de variables scalaires**

`character`	pour les chaînes d'un ou plusieurs caractères.
`logical`	pour les variables booléennes : `.true.` (vrai) ou `.false.` (faux).
`real`	pour les nombres réels.
`integer`	pour les nombres entiers relatifs.
`complex`	pour les nombres complexes ($x + iy$).

La forme générale de déclaration de variable est

```
type, [,liste_attributs ::]  listes_variables
```


Les attributs

Les **attributs** peuvent être :

`parameter`	pour une constante symbolique.
`dimension(...)`	pour un tableau.
`allocatable`	pour un tableau dynamique.
`pointer`	pour un objet accessible par pointeur.
`target`	pour un objet cible potentielle d'un pointeur.
`save`	pour un objet rémanent.
`intent(...)`	pour un paramètre formel.
`optionnal`	pour un paramètre formel optionnel.
`public, private`	pour une entité définie par un module.

Quelques déclarations simples

```
character          :: sexe   ! un caractere
character(len=10)  :: nom    ! 10 caracteres au plus
character(len=*)   :: prenom ! longueur inconnue
logical            :: cel    ! celibataire ou non ?
real               :: taille, poids
integer            :: age
```

Remarque : Les types scalaires présentent des caractéristiques qui varient d'une machine à l'autre.

Les sous-types : le **paramètre `kind`**

Les types scalaires `integer`, `real`, `logical` et `character` sont des noms génériques désignant plusieurs **sous-types** ou **variantes** accessibles grâce au paramètre de type **`kind`** :

```
integer(4)        :: i,j   ! entier court
integer(kind=8)   :: k,l   ! entier long
real(kind=4)      :: x,y   ! reel simple precision
real(8)           :: Tol   ! reel double precision
```

Les sous-types : la **fonction intrinsèque `kind`**

La **fonction intrinsèque `kind`** retourne la valeur associée au :

`kind(x)`	sous-type de la variable x.
`kind(0)`	type entier par défaut.
`kind(0.0)`	type réel par défaut.
`kind(1.0)`	sous-type réel simple précision.
`kind(1.d0)`	sous-type réel double précision.

Déclaration d'un réel double précision quelle que soit la machine :

```
real (kind=kind(1.d0)) ::  x
```


Le typage implicite

En absence, de toute déclaration, les variables dont les noms commencent par :

- `i,j,k,l,m,n` sont considérées comme de type `integer`,
- `a,b,...,h,o,p,...,z` sont considérées comme de type `real`.

⇒ Erreurs difficiles à détecter,

⇒ Inhibition du typage implicite : `implicit none`

Les constantes littérales

- Constantes entières : forme **décimale** avec ou sans signe

```
-2365     +7665452     467498
```

- Constantes réelles : forme **décimale** ou **exponentielle**

```
-0.314159     -.314159
1.e-5    1e-5     3.88e5
1.d0          -.76543d-12
```

- Constantes chaîne de caractères : **apostrophes** ou **guillemets**

```
'jimmy'       "Shiva"
```


Les constantes littérales et les sous-types

Spécification du **sous-type** d'une constante : le caractère _.

```
256_4            ! constante entier court
3.14159_4        ! constante reel simple
687.9865209876_8 ! constante reel double
1_'Selena'       ! chaine 1 octet/caractere
```

Les constantes symboliques

- Utilisation de l'attribut **`parameter`**.
- L'appel à **certaines fonctions élémentaires** est possible lors de la déclaration.

```
real, parameter :: pi=3.141516
character(len=*), parameter :: Name='Satan'
integer, parameter :: Nmax1=10, Nmax2=30
integer, parameter :: Res=mod(Nmax1,Nmax2)
integer, parameter :: Nbv=abs(Nmax1-Nmax2)
```


Les constantes symboliques pour des déclarations portables

Les constantes symboliques peuvent être utilisées pour rendre les déclarations de variables avec variantes un peu plus portables.

```
integer, parameter    :: rdouble=8
integer, parameter    :: ishort=4
integer(kind=ishort) :: i, j
real(kind=rdouble)    :: x, y, z
```

La précision des nombres : `select_int_kind`

`select_int_kind(r)` fournit

- le numéro de variante du type `integer` acceptant **au moins r chiffres décimaux**, i.e. les entiers dans l'intervalle $]-10^r, 10^r[$.
- -1 si aucun sous-type ne correspond à la demande.

La précision des nombres : `select_real_kind`

`select_real_kind(p,r)` fournit

- le numéro de variante du type `real` susceptible de représenter des nombres réels avec une **précision** de p et une **étendue** de r.
- -1 si la précision demandée n'est pas disponible,
- -2 si l'étendue désirée n'est pas disponible,
- -3 si ni l'un ni l'autre ne sont disponibles.

La précision des nombres : example

On peut maintenant écrire des programmes vraiment portables. Voici quelques déclarations avec variantes indépendantes de la machine.

```
integer, parameter :: rprec=select_real_kind(p=9,r=50)
integer, parameter :: iprec=select_int_kind(r=2)
  ...
integer(kind=iprec):: i, j, k
real(kind=rprec)   :: x, y, z
```


Initialisation

L'initialisation d'une variable se fait à la déclaration par simple affectation comme dans l'exemple suivant.

```
real, parameter  :: Pi=3.14159, Rayon=6500
character(len=*) :: Jour="Lundi", Mois="Janvier"
integer          :: i=0, nbr=90
real(8)          :: x=0.d0

! Initialisation avec une expression
real             :: circ=2*Pi*Rayon

! Avec une fonction intrinseque autorisee
real             :: prec=epsilon(1.0)
```

Chapitre 2:

Expressions, instructions élémentaires
- Les expressions
- Les instructions élémentaires

Les expressions

On distingue 3 types d'expressions en FORTRAN :
- les expressions arithmétiques,
- les expressions logiques,
- les expressions de type texte.

Les expressions arithmétiques

Opérateurs arithmétiques usuels par ordre de priorité décroissante :

`**`	élévation à la puissance
`* /`	multiplication et division
`+ -`	addition et soustraction

Exemples d'expressions arithmétiques :

`x-y**3/100.0`	correspond à $x - (y^3/100)$
`a/b/c`	correspond à $a/(bc)$
`-a+c/d`	correspond à $(-a) + (c/d)$
`x**y**z`	correspond à $(x^y)^z$

Les expressions arithmétiques : compléments

Les **parenthèses** permettent de forcer la priorité et d'améliorer la lisibilité du programme :

```
x-((y**3)/100.0)
a/(b/c)
-a+(c/d)
(x**y)**z
```

Quotient entre 2 entiers :

- `3/2` vaut 1
- `3.0/2.0` vaut (approximativement) 1.5

Les expressions logiques

Elles sont construites à partir de **comparaison entre expressions numériques** et d'**opérateurs logiques**.

Les opérateurs de comparaison :

ancienne notation	nouvelle notation	signification
`.lt.`	`<`	inférieur à
`.le.`	`<=`	inférieur ou égal à
`.gt.`	`>`	supérieur à
`.ge.`	`>=`	supérieur ou égal à
`.eq.`	`==`	égal à
`.ne.`	`/=`	différent de

Les expressions logiques

- La **priorité** des opérateurs de comparaison est inférieure à celle de tous les opérateurs arithmétiques :

```
x**2 < a+b
b**2-4.0*a*c .gt.  0
-b+sqrt(delta) < .5*cos(2.0*omega)
```

- **Egalité entre réels :** l'opérateur de comparaison == ne doit être utilisé pour des expressions réelles qu'avec beaucoup de précaution car il n'y a pas absolue égalité entre deux réels.

Les expressions logiques

- Les opérateurs logiques :

opérateur	signification
`.and.`	*et* logique, vrai si les 2 opérandes sont vrais
`.or.`	*ou* logique, vrai si au moins un opérande est vrai
`.not.`	négation
`.eqv.`	équivalence logique, vraie si les 2 opérandes sont tous vrais ou tous faux
`.neqv.`	non équivalence logique

- On peut imprimer une expression logique. On obtient alors un `F` (pour `.FALSE.`) ou un `T` (pour `.TRUE.`).

Les instructions élémentaires : l'affectation

- L'**affectation** se fait avec le symbole `=`.
- La forme la plus simple pour l'affectation est
  `variable = constante`
- La forme générale est
  `variable = expression`

Exemple d'affectation :

```
s=Pi*R**2
h=sin(a)**2
boole=(x<y .or. abs(z)<eps)   ! expression logique
```

Les instructions élémentaires : entrées/sorties

- La saisie des données au clavier :
  `read *,var1,var2,...`

  Les variables `var1,var2,...` sont alors entrées séparées par une virgule.

- L'affichage des données à l'écran :
  `print fmt,element1,element2,...`

- `fmt` est
  - soit le **caractère** $\star$ qui représente la sortie standard,
  - soit une **spécification de format** sous forme d'une chaîne de caractères. On peut spécifier du texte, des nombres, des sauts de lignes, ...

Les instructions élémentaires : entrées/sorties

- Exemple d'impression simple :

```
print *,"a=",a,"  b=",b,"  c=",c
print *,'La solution est x=',x
```

- Exemple d'impression avec une spécification de format :

```
print '("a=",F15.6,"b=",E15.8)',a,b
print '(/"x=",E15.8/)',x
```

Les instructions élémentaires : entrées/sorties

`A`	Impression d'une chaîne de caractères
`Im`	Impression d'un entier sur *m* colonnes
`Fm.n`	Impression d'un réel en notation décimale sur *m* colonnes avec *n* chiffres après la virgule
`Em.n`	Impression d'un réel en notation exponentielle sur *m* colonnes avec *n* chiffres après la virgule
`Gm.n`	Comme `Fm.n` mais si le réel est trop grand, il s'écrira avec un exposant
`/`	Changement de ligne
`x`	Ecriture d'un espace

Les instructions élémentaires : entrées/sorties

- Facteur de répétition :

```
3I4            ⟺  I4,I4,I4
2(I3,F15.8)    ⟺  I3,F15.8,I3,F15.8
```

- Une spécification de format est une chaîne de caractères. Elle peut donc être stockée dans une variable ou une constante symbolique de type chaîne.

Les instructions élémentaires : entrées/sorties

```
program AFFICHE
  implicit none
  real(8) :: x,pi
  integer :: i=12345

  print '(//,27("-"),/,"--- AFFICHAGE D'ENTIERS ---",/&
           &,27("-"),/)
  print '("Entier format adapte          : ",I6)',i
  print '("Entier format non adapte      : ",I4)',i*i
  print '("Avec facteur de repetition    : ",3I6)',i,2*i,3*i

  x=1.d0; pi=4.d0*atan(x)

  print '(//,27("-"),/,"--- AFFICHAGE DE REELS ---",/&
           &,27("-"),/)
  print '("Reel forme naturelle          : ",F12.8)',pi
  print '("Reel forme exponentielle      : ",E15.8)',pi
  print '("Reel forme non adapte         : ",F10.8)',1.d3*pi
end program AFFICHE
```


Les instructions élémentaires : entrées/sorties

```
---------------------------
--- AFFICHAGE D'ENTIERS ---
---------------------------
Entier format adapte        :  12345
Entier format non adapte    : ****
Avec facteur de repetition  :  12345 24690 37035



---------------------------
--- AFFICHAGE DE REELS ---
---------------------------
Reel forme naturelle        :   3.14159265
Reel forme exponentielle    :  0.31415927E+01
Reel forme non adapte       : **********
```

Chapitre 3:

Structures alternatives

L'**alternative simple** s'écrit

```
if (condition) then
   action
endif
```

ou encore

```
if (condition) then
   action1
else
   action2
endif
```

Structures alternatives

Exemple :

```
if (x>y) then
   print *,x,' est plus grand que ',y
else
   print *,x,' est plus petit ou egal a ',y
endif
```

Dans cette exemple, chaque ligne représente une instruction à part entière.

```
if (x>y) then; print *,x,' est plus grand que ',y
else; print *,x,' est plus petit ou egal a ',y ; endif
```


Structures alternatives

Lorsque l'action se limite à une seule instruction, l'alternative simple peut s'écrire aussi :

```
if (condition) action
```

Par exemple, l'alternative simple suivante

```
if (x>imax) then
   imax=x
end
```

peut se réduire à

```
if (x>imax) imax=x
```

Structures alternatives

L'**alternative complète** (avec des "sinon si") est de la forme :

```
if (condition_1) then
  action_1
else if (condition_2) then
  action_2
  ...
else if (condition_n) then
  action_n
else
  action_0
endif
```

Un seul `endif` ferme l'alternative.


```
program RACINEQ2
  implicit none                                    ! ne jamais oublier
  integer, parameter         :: ir8=kind(1.d0)  ! sous-type == real double
  real(kind=ir8), parameter :: eps=1d-12        ! precision des calculs
  real(kind=ir8)             :: a,b,c           ! coef. du trinome
  real(kind=ir8)             :: x1,x2, delta
  complex(kind=ir8)          :: z1,z2           ! pour les racines complexes

  ! lecture des coefficients
  print *,'Entrer les coef. reels: a, b, c '
  read *,a,b,c

  if (abs(a)>eps) then

     delta=b*b-4.0*a*c              ! calcul discriminant

     if (delta> 0) then             ! cas classique : 2 racines reelles
        x1=.5*(-b-sqrt(delta))/a
        x2=.5*(-b+sqrt(delta))/a

        print *,"Deux racines reelles distinctes"
        print '("x1=",F12.8,"  x2=",F12.8)',x1,x2

     elseif (delta<0) then          ! 2 racines complexes
        x1=-.5*b/a                  ! partie reelle des racines
        x2=.5*sqrt(abs(delta))/a    ! partie imag. (valeur absolue)
        z1=cmplx(x1,-x2)            ! conversion -> z1
        z2=cmplx(x1,x2)             ! conversion -> z2
```

```
      if (delta> 0) then            ! cas classique : 2 racines reelles
         x1=.5*(-b-sqrt(delta))/a
         x2=.5*(-b+sqrt(delta))/a

         print *,"Deux racines reelles distinctes"
         print '("x1=",F12.8,"  x2=",F12.8)',x1,x2

      elseif (delta<0) then         ! 2 racines complexes
         x1=-.5*b/a                 ! partie reelle des racines
         x2=.5*sqrt(abs(delta))/a   ! partie imag. (valeur absolue)
         z1=cmplx(x1,-x2)           ! conversion -> z1
         z2=cmplx(x1,x2)            ! conversion -> z2

         print *,"Deux racines complexes :"
         print '("z1=(",2F12.8,")","  z2=(",2F12.8,")")',z1,z2

      else                          ! equation (x+.5*b/a)**2=0
         x1=-.5*b/a;
         print *,'("Une racine double",F12.8)',x1
      endif

   else                             ! equation bx+c=0
      if (abs(b)>eps) then
         x1=-c/b
         print '("Une racine reelle x=",F12.8)',x1
      else; print *,'Equation indeterminee '; endif
   endif
end
```

Voici quelques exemples d'exécution

```
Entrer les coef. reels: a, b, c
1.5, 2, 5.1897654
Deux racines complexes :
z1=( -0.66666667 -1.73649047)  z2=( -0.66666667  1.73649047)

Entrer les coef. reels: a, b, c
3.14159, 10.6573, 2
Deux racines reelles distinctes
x1= -3.19294328  x2= -0.19938353
```

Boucle avec compteur : `do`

Forme générale :

```
[nom:] DO var=debut,fin [,pas]
           instruction
       END DO [nom]
```

avec

`nom`	identificateur de la boucle
`var`	un identificateur d'une variable de type `integer`
`debut, fin`	expressions quelconques de type `integer`
`pas`	idem mais optionnel. Il vaut 1 par défaut.

Exemples

Somme des entiers de 1 à *n* :

```
s=0
do i=1,n       ! le pas vaut 1 par defaut
   s=s+i
end do
```

Somme des nombres impairs :

```
s=0
do i=1,n,2
   s=s+i
end do
```

Boucle "tant que" (`do while`)

La forme générale est la suivante :

```
[nom:] DO WHILE (expression_logique)
           instructions
       END DO [nom]
```

Attention aux boucles infinies !!!

Exemple

Soit la suite récurrente définie par

$$u_{n+1} = \frac{u_n^3 + 3au_n}{3u_n^2 + a}, \quad u_0 = 1$$

où a est un réel strictement positif.

La suite u_n converge vers $\sqrt{a}$, pour $a > 0$ donné.

```
program SUITE_RECC
  implicit none                              ! pour inhiber le typage implicite
  integer, parameter :: IterMax=100          ! nb max d'iterations
  integer, parameter :: ir8=kind(1.d0)       ! sous-type reel double
  real(8), parameter :: eps=1d-6             ! precision des calculs
  real(kind=ir8)     :: a
  real(kind=ir8)     :: u,u1                 ! termes courant et precedant

  integer            :: iter
  real(kind=ir8)     :: erreur

  ! Lecture de a
  print *,'Entrer le reel a >0 :'
  read *,a

  erreur=1
  u1=1
  iter=0
  do while (erreur>eps .and. iter<IterMax)
     iter=iter+1
     u=(u1*u1*u1+3*a*u1)/(3*u1*u1+a)        ! terme courant
     erreur=abs(u-u1)/u                     ! erreur relative entre u et u1
     u1=u
  end do

  print '("La limite de la suite est ",F12.8) ',u
  print '("Apres ",I4," iterations ")',iter

end program SUITE_REC
```

Voici quelques exécutions

```
 Entrer le reel a >0 :
3.14159
La limite de la suite est   1.77245310
Apres    4 iterations

 Entrer le reel a >0 :
32.184789
La limite de la suite est   5.67316393
Apres    5 iterations
```

Sortie anticipée d'une boucle : `exit`

`exit` sert à interrompre le déroulement d'une boucle.

```
alpha=0.67
s=stock               ! stock>0
do i=1,n
   if (s<0) exit      ! on arrete si s devient negatif
   s=s-alpha*real(i)  ! real(i) convertit l'entier i en reel
end do
```

Boucles imbriquées : `exit` met fin qu'à la boucle la plus interne.

```
i=0;
do while (i<m)
   u=1;
   do j=1,n
      s=m*m-i*j
      if (s<0) exit
      u=s*u
   end do
   print *,"i=",i,"  u=",sqrt(u)
end do
```


Sortie anticipée d'une boucle : `exit`

Solution : donner un nom à la boucle et de préciser juste après l'instruction `exit` le nom de la boucle à interrompre.

```
i=0;
Julie : do while (i<m)      ! on donne un nom a la boucle critique
   u=1;
   do j=1,n
      s=m*m-i*j
      if (s<0) exit Julie   ! on sort de la boucle julie
      u=s*u
   end do
   print *,"i=",i,"  u=",sqrt(u)
end do Julie                ! fin Julie
```

Bouclage anticipé : `cycle`

`cycle` permet de passer prématurément au tour de boucle suivant.

```
j=0
Emma : do while (j<10)
   j=j+1
   s=real(j*j)
   do i=1,n
      s=s+i
      if (s>seuil) cycle Emma  ! on passe au prochain tour de boucle
   end do
   print *,"Somme =",s
end do emma
```


Boucle infinie : instruction `do`

Il existe une boucle "sans fin" en Fortran :

```
[nom :] do
    ...
    ...
end do
```

En pratique, il faut arrêter la boucle avec l'instruction `exit` comme dans l'exemple suivant.

```
do
   print *,"Entrez un entier positif"
   read *,i
   if (i>0) exit      ! on arrete si i>0
end do
```

Structure de choix multiple : `select case`

La forme générale est :

```
[nom :] select case (exp_scal)
   case (selecteur) [nom]
      ...
   [ case default [nom]
      ...]
end select [name]
```

Avec :

- `exp_scal` expression scalaire de type `integer` ou `character`
- `selecteur` liste composée de 1 ou plusieurs éléments de la forme
  - valeur
  - intervalle de la forme `[valeur1]:valeur2` ou `valeur1:[valeur2]`

  les valeurs concernées devant être du même type que `exp_scal`.

Exemple

```
program SELECTCASE
implicit none
integer :: n
print *,"Donnez un nombre entier "
read *,n
select case (n)
   case (0)
      print *,"n=0"
   case (1,2)
      print *,"n=1 ou n=2"
   case (3:10)
      print *,"3 <= n <= 10"
   case (11:)
      print *,"n >= 11"
   case default
      print *,"n < 0"
end select
end program SELECTCASE
```

L'instruction `stop`

L'instruction `stop` met fin au programme.

Elle peut s'utiliser n'importe où comme n'importe quelle instruction exécutable.

```
if (...) then
   print *,"Probleme insurmontable -- on arrete tout"
   stop
endif
```


Chapitre 4:

Les tableaux : définitions

- Ensemble ordonné d'éléments de même type
- Identificateur unique
- Chaque élément est repéré par un indice
- **Nombreuses facilités :** opérations globales, manipulation de portion de tableau, affectation conditionnelle, nombreuses fonctions intrinsèques, etc

Rang d'un tableau

- Le **rang d'un tableau** est son nombre de dimensions.
- Un vecteur est de rang 1, une matrice de rang 2, etc...
- Un scalaire est considéré comme de rang 0.
- En Fortran 90, un tableau peut avoir jusqu'à 7 dimensions au maximum.

Etendue et profil d'un tableau

- Dans chaque dimension, un tableau a une **étendue**, qui est le nombre de composantes du tableau dans cette dimension.
- Le **profil** d'un tableau est la suite des étendues de ce tableau selon ses dimensions successives sous forme d'un vecteur d'entiers (soit 1 entier pour un vecteur, 2 pour une matrice, etc.).
- Le produit des étendues représente la **taille** du tableau, i.e. son nombre d'éléments.

Tableaux conformants

- Deux tableaux sont dits **conformants** s'ils ont le même profil.
- Par convention un scalaire est conformant avec tout tableau.

Déclaration

Pour déclarer un tableau, il suffit de préciser l'attribut `dimension`

```
integer, dimension(5)        ::  idx   ! simple vecteur
real(8), dimension(3,4)      ::  A     ! matrice 3 lignes 4 colonnes
real, dimension(-1:10,0:10)  ::  C     ! matrices 12 lignes 11 colonnes
```

Par défaut, la valeur initiales des indices est égale à 1.

```
integer, dimension(5)   :: v         ! declarations
integer, dimension(1:5) :: v         ! equivalentes
```


Exercice

On peut maintenant tester les définitions vues plus haut à partir des déclarations suivantes.

```
real, dimension(-5:4,0:2)   :: x
real, dimension(0:9,-1:1)   :: y
real, dimension(2,3,0:5)    :: z
```

	rang	profil	taille
x			
y			
z			

Exercice

On peut maintenant tester les définitions vues plus haut à partir des déclarations suivantes.

```
real, dimension(-5:4,0:2)  :: x
real, dimension(0:9,-1:1)  :: y
real, dimension(2,3,0:5)   :: z
```

	rang	profil	taille
x	2	(10 3)	30
y	2	(10 3)	30
z	3	(2 3 6)	36

Constructeur de tableau

Un **constructeur de tableau** est une liste de scalaires (de même type!) dont les valeurs sont encadrées par les caractères `(/` et `/)`.

La liste peut être explicite comme

```
(/3, 5, 1, 8, 12/)
```

ou comportée une boucle implicite comme

```
(/ (3*i+1, i=1,3) /)     ! liste (/ 4,7,10 /)
```

Constructeur de tableau

Pour le compteur, on utilise la même règle que dans la boucle `do`, i.e.

```
(/ (expression, compteur=debut, fin [,pas]) /)
```

Par exemple

```
(/ (3*i+1, i=1,6,2) /)
```

représente une liste de 3 entiers: `(/ 4,10,16 /)`.

Constructeur de tableau

`(/ ... /)` est utilisée pour éviter tout conflit avec les nombres complexes.

`c=(0,1)` est un nombre complexe
`c=(/ 0,1 /)` est un tableau

Grâce au constructeur, on peut initialiser un tableau **de rang 1** au moment de sa déclaration ou lors d'une instruction d'affectation.

Pour les tableaux de rang supérieur à 1, on utilisera la fonction `reshape` détaillée plus loin.

Constructeur de tableau

Voici quelques exemples d'initialisation.

```
integer               :: i
integer, dimension(5) :: idx=(/ 2,6,11,8,2 /)
real, dimension(0:90) :: x=(/ (2*i-3, i=0,90) /)
integer, dimension(10):: kx
 ...
kx=(/ (2*i+1,i=1,10) /)
```

Si un compteur est utilisé dans le constructeur, il doit absolument avoir été déclaré avant utilisation.

Déclaration de tableaux constants :

```
integer, dimension(4), parameter :: idx=(/ 2,3,1,2 /)
```


Constructeur de tableau

Constructeur et tableaux de dimension supérieure à 1 :

```
reshape(source,shape)
```

avec

`source` Tableau de rang 1 de type quelconque. Contient la liste d'éléments qui serviront dans l'initialisation.

`shape` Tableau d'entiers non négatifs de rang 1. Contient le profil de la matrice à remplir.

Constructeur de tableau

Considérons les déclarations suivantes :

```
integer, dimension(6)   :: idx=(/ (i,i=1,6) /)
integer, dimension(3,2) :: v=reshape(idx,(/ 3,2 /))
```

L'instruction d'initialisation

```
v=reshape(idx,(/ 3,2 /))
```

est équivalente à

```
v(1,1)=idx(1)       !      |  1   4  |
v(2,1)=idx(2)       ! v =  |  2   5  |
v(3,1)=idx(3)       !      |  3   6  |
v(1,2)=idx(4)
v(2,2)=idx(5)
v(3,2)=idx(6)
```


Affectation globale

Affectation d'une valeur à tous les éléments d'un tableau :
Un scalaire est conformant avec tout tableau !!

```
real, dimension(10) :: u
```

L'instruction d'**affectation globale** suivante

```
u=0.0
```

est équivalente à la boucle

```
do i=1,10
   u(i)=0.0
end do
```

Affectation globale

De même, avec

```
integer, dimension(10,25) :: A
```

l'instruction

```
A=1
```

affecte la valeur 1 aux 250 éléments du tableau `A`.

Initialisation d'un tableau lors de sa déclaration.

```
integer, parameter   :: dim=100
real, dimension(dim) :: x=0
```

le tableau `x` est initialisé à 0 lors de sa déclaration.

Opérations globales : Addition et multiplication

On obtient une **expression tableau**, i.e. une expression qui fournit comme résultat un tableau.

Soit les déclarations

```
integer, parameter   :: dim=100
real, dimension(dim) :: x,y,z
```

L'instruction

```
z=x+y
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)+y(i)
end do
```

Opérations globales : Addition et multiplication

De même, l'instruction

```
z=x*y
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)*y(i)
end do
```

Donc, dans le cas du produit `x*y` il s'agit d'un **produit élément par élément** et non d'un produit scalaire (fonction intrinsèque `dot_product`).

Opérations globales : Addition et multiplication

Un scalaire est **conformant** avec tout tableau.

Donc dans une expression tableau, on peut avoir des scalaires comme opérandes.

```
z=x+y+3.14159
```

est équivalente à

```
do i=1,dim
   z(i)=x(i)+y(i)+3.14159
end do
```

Opérations globales : Tableaux conformants

Toutes ces opérations ne sont possibles que si les tableaux opérandes ont le **même profil**, i.e. le même nombre d'éléments dans chaque dimension.

Il n'est pas nécessaire que les indices aient les mêmes limites.

```
real, dimension(-5:5)  :: x    ! profil (/ 11 /)
real, dimension(11)    :: y,z  ! profil (/ 11 /)
real, dimension(5:16)  :: u    ! profil (/ 11 /)
```

on peut écrire

```
x=z
z=y+u
```


4. Les tableaux | 4.3. Opérations globales sur les tableaux

Opérations globales : Tableaux conformants

La notion de profil devient encore plus importante lorsqu'il s'agit de tableaux à plusieurs dimensions.

```
real, dimension(10,20)       :: a     ! profil (/ 10,20 /)
real, dimension(-2:7,10:29)  :: b     ! profil (/ 10,20 /)
real, dimension(10, 0:19)    :: c     ! profil (/ 10,20 /)
```

Alors l'instruction

```
c=a+b
```

est équivalente à

```
do i=1,10
   do j=1,20
      c(i,j-1)=a(i,j)+b(i-3,j+9)
   end do
end do
```

Evaluation d'une expression tableau

Remarque : La valeur d'une expression tableau est entièrement évaluée avant d'être affectée.

```
x=2*x          ! multiplie tous les elements de x par 2
x=x+1          ! augmente de 1 la valeur des elements de x
```


Lisibilité

Dans les expressions de la forme

```
z=x+y
c=a+b
```

il n'apparaît pas d'emblée que ce sont des tableaux.

Pour augmenter la lisibilité du code, on peut écrire

```
z(:)=x(:)+y(:)
c(:,:)=a(:,:)+b(:,:)
```

Section de tableau, vecteur d'indices

- En pratique, la taille physique d'un tableau est plus grande que la taille effective.
- Initialisation de la partie effective, opérations sur des portions de tableaux.
- Une partie d'un tableau est appelée **section tableau** ou sous-tableau.
- Section régulière : les indices ayant servi à la créer forment une progression arithmétique.
- Section irrégulière : cas contraire.

Section régulière

Définition d'une section régulière

```
var_tableau([debut]:[fin][:pas])
```

avec `debut, fin` et `pas` des expressions **entieres** quelconques.

Les valeurs par défaut sont:

- `debut` la valeur du premier indice du tableau
- `fin` la valeur du dernier indice du tableau
- `pas` 1

Section régulière

Soit le tableau déclaré ci–après

```
real, dimension(100) :: u
```

Voici quelques sections tableau de u

```
u(:)          ! tout le tableau u
u(:50)        ! les 50 premiers elements de u
u(51:)        ! les 50 derniers elements de u
u(10:20)      ! les elements  u(i), 10<=i<=20
u(1:100:2)    ! tous les elements d'indices impairs
```


4. Les tableaux 4.4. Section de tableau, vecteur d'indices

Section régulière

Avec les déclarations suivantes

```
real, dimension(10) :: x
real, dimension(5)  :: y
```

on peut écrire

```
y=x(1:5)          ! on affecte a y les 5 premiers
                  ! elements de x
```

On affecte a y les 5 premiers elements de x.

Section régulière

Une section régulière peut apparaître à gauche d'une instruction d'affectation. On peut donc écrire

```
x(1:5)=1          ! on initialise a 1 les 5 premiers
do i=1,5
   print *,(x(i,j), j=1,5)
end do
```


Section régulière

Comme on l'a déjà signalé, une expression tableau est entièrement évaluée avant d'être affectée.

```
x(2:9)=(x(1:8)+x(3:10))/2
```

On remplace chaque composante de `x`, exceptés les deux extrêmes, par la valeur moyenne des deux composantes voisines.

Sans utiliser de section tableau, il ne faudrait surtout pas écrire

```
do i=2,9
   x(i)=(x(i-1)+x(i+1))/2
end do
```

car le résultat sera fort éloigné de celui escompté.

Section régulière

Remarque : une section régulière est en réalité une pseudo–boucle `do`.

Ainsi dans l'instruction

```
x(1:n)=0
```

il ne se passera rien si $n < 1$, comme dans une boucle `do`.

Vecteur d'indices (section non régulière)

Lorsque les indices d'une section tableau ne forment pas une progression arithmétique, on peut les regrouper dans un **vecteur d'entiers**.

```
real, dimension(5)  :: x
real, dimension(10)   :: y
```

`(/ 1,3,7,10 /)` est un vecteur (constant) d'entiers.

Alors `y((/ 1,3,7,10 /))` est un tableau de 4 éléments constitué des éléments `y(1), y(3), y(7), y(10)`.

Vecteur d'indices (section non régulière)

On peut donc écrire

```
y((/ 1,3,7,10 /)) = 0.
x(2:5)=y((/ 1,3,7,10 /))
```

Ce qui correspond à

```
y(1)=0.;    y(3)=0.;    y(7)=0.;    y(10)=0.
x(2)=y(1); x(3)=y(3); x(4)=y(7); x(5)=y(10)
```


Vecteur d'indices (section non régulière)

En général, pour des raisons évidentes de lisibilité, on préfère utiliser une variable tableau, avec un contenu pouvant évoluer.

```
integer, dimension(4) :: idx=(/ 1,3,7,10 /)
```

l'affectation précédente devient

```
y(idx)=0.
x(2:5)=y(idx)
```

Vecteur d'indices (section non régulière)

Les indices peuvent être répétés dans les vecteurs d'indices comme dans

```
integer, dimension(4) :: idx=(/ 2,2,7,10/)
```

Il n'y a aucune ambiguïté lorsqu'on veut seulement utiliser la valeur de cette section tableau.

```
x(2:5)=y(idx)
```

correspond à

```
x(2)=y(2); x(3)=y(2); x(4)=y(7); x(5)=y(10)
```


Vecteur d'indices (section non régulière)

On voit tout de suite qu'on ne peut pas écrire

```
y(idx)=x(2:5)   ! INTERDIT
```

puisque `y(2)` recevrait 2 valeurs distinctes.

D'une manière générale, lorsqu'une section tableau apparaît à gauche d'une affectation, elle ne doit pas faire intervenir deux fois le même élément.

Cas des tableaux à plusieurs dimensions

Pour un tableau de rang supérieur à 1, tout ce qu'on vient de voir est valable pour chaque dimension.

```
real, dimension(5,10) :: a
real, dimension(5,5)  :: b
```

`a(1:5,1:5)` est un tableau de rang 2 de profil `(/ 5,5 /)`, comme `b`.

On peut donc écrire

```
b=2*a(1:5,1:5)+1
```


Cas des tableaux à plusieurs dimensions

Combinaison de sections régulières et irrégulières :

`a(2,1:5)` représente un tableau de rang 1 de 5 éléments

`a(2:2,1:5)` représente un tableau de rang 2 de profil `(/ 1,5 /)`.

Permuter deux lignes i et j d'une matrice

```
integer, parameter      :: ndmax=100   ! dimension physique des tab.
integer                 :: nd          ! dimension reelle (effective)
real, dimension(ndmax)        :: x     ! vecteur de travail
real, dimension(ndmax,ndmax)  :: A
...
x(1:nd)=A(i,1:nd)                      ! recopie la ligne i dans x
A(i,1:nd)=A(j,1:nd)                    ! recopie la ligne j
                                       ! dans la ligne i
A(j,1:nd)=x(1:nd)                      ! recopie x dans la ligne j
...
```


Combinaison linéaire des lignes i et j d'une matrice

```
integer, parameter      :: ndmax=100   ! dimension physique des tab.
integer                 :: nd          ! dimension reelle (effective)
real                    :: c1,c2       ! coeff. reels de la
                                       ! combinaison lineaire
real, dimension(ndmax,ndmax)  :: A
...
A(i,1:nd)=c1*A(i,1:nd)+c2*A(j,1:nd)    ! en une ligne seulement !!
...
```

Créer une matrice par bloc

Soit A_1 et A_2 des matrices carrées d'ordre n. On veut créer une nouvelle matrice A de la forme

$$A = \begin{pmatrix} A_1 & O \\ O & A_2 \end{pmatrix}$$

où O est une matrice carrée d'ordre n ne contenant que des 0.

```
integer, parameter     :: nmx =100    !  dim. physique de A1 et A2
integer, parameter     :: nmx2=2*nmax !  dim. physique de A
integer                :: n           !  dim. effective de A1 et A2.
integer                :: n2          !  dim. effective de A (n2=2*n)
real, dimension(nmx,nmx)   :: A1, A2
real, dimension(nmx2,nmx2) :: A
...
A(1:n2,1:n2) = 0.                     ! on initialise
A(1:n,1:n) = A1(1:n,1:n)              ! bloc A1
A(n+1:n2,n+1:n2) = A2(1:n,1:n)        ! bloc A2
...
```


L'instruction `where`

Les fonctions mathématiques élémentaires usuelles s'appliquent également aux tableaux.

Et dans ce cas, elles retournent un tableau de même profil.

```
y(1:n)=sqrt(x(1:n))
```

est équivalent à

```
do i=1,n
   y(i)=sqrt(x(i))
end do
```

L'instruction `where`

Problème : domaine de définition de la fonction $\sqrt{x_i}$.

L'instruction `where` sélectionne les éléments d'un tableau suivant un test.

La forme générale de l'instruction `where` est

```
where (expression_logique)
  bloc_1
elsewhere
  bloc_2
end where
```


L'instruction `where`

Voici un exemple d'utilisation.

```
real, dimension(10) :: x,y
...
where (x>0)
  y=log(x)
elsewhere
  y=1
end where
```

est équivalent à

```
real, dimension(10) :: x,y
integer             :: i
...
do i=1,10
   if (x(i)>0) then
      y(i)=log(x(i))
   else
      y(i)=1
   endif
end do
```

L'instruction `where`

Lorsque `bloc_2` est absent et `bloc_1` se résume en une seule instruction, on peut utiliser la forme simplifiée

```
where (expression_logique) instruction
```

comme dans

```
where (x>0) y=sqrt(x)
```


Tableaux dynamiques

Problème : étendues des tableaux inconnueas lors de l'écriture du programme.

Fortran 77 : «surdimensionner» les tableaux en question.

Fortran 90 : allocation dynamique de mémoire.,
l'attribut `allocatable` (e rang du tableau doit être connu).

```
real, dimension(:),   allocatable :: vecteur
real, dimension(:,:), allocatable :: matrice
```

Tableaux dynamiques

Allocation : instruction `allocate` à laquelle on indiquera le profil désiré.

```
allocate(vecteur(n))                 ! vect. de taille n
allocate(vecteur(n1:n2))             ! vecteur(i), n1<=i<=n2
allocate(matrice(m,n))               ! mat. de taille m*n
allocate(matrice(m1:m2,n1:n2))
allocate(vecteur(n),matrice(m,n)) ! allocation simultannee
```

La fonction intrinsèque `allocated` permet de savoir si un tableau a été déjà alloué ou non.

En cas d'échec d'allocation, il y a arrêt de l'exécution. Pour éviter ce comportement brutal, un paramètre optionnel `stat` permet de savoir si l'allocation a réussie ou echouée.

Tableaux dynamiques

```
program ALLOC
  implicit none
  real, dimension(:,:), allocatable :: a     ! tableau dynamique
  integer                           :: m,n   ! futur profil du tablea
  integer                           :: aerr  ! pour l'erreur
                                             ! d'allocation

  print *,'Entrer le profil de la matrice (m,n) :'
  read *,m,n
  ...
  if (.not. allocated(a)) then    ! a n'est pas encore alloue
     allocate(a(m,n),stat=aerr)   ! allocation de a : profil (m,n)
                                  ! recuperation de l'err dans aerr
     if (aerr /= 0) then          ! si aerr<>0, l'alloc a echoue
        print *,''Erreur dans l'allocation du tableau a :''
        stop
     endif
  endif
  ...
  deallocate(a)                   ! on libere l'emplacement de a
  ...
end program ALLOC
```

Méthode de la puissance

Soit A une matrice réelle d'ordre n. On suppose que les valeurs propres de A sont ordonnées de la façon suivante

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

avec λ_1 de multiplicité 1. Etant donné un vecteur initial q^0, de norme euclidienne 1, considérons pour $k = 1, 2, \ldots$ la méthode itérative suivante, connue sous le nom de *méthode de la puissance*

$$\begin{aligned} v^k &= Aq^{k-1} \\ q^k &= \frac{v^k}{\| v^k \|} \\ \lambda^k &= (Aq^k, q^k) \end{aligned}$$

On montre que λ^k tend vers λ_1, la plus grande valeur propre de A, lorsque $k \longrightarrow +\infty$. On arrête les calculs si

$\frac{|\lambda^k - \lambda^{k-1}|}{|\lambda^k|} < \varepsilon$, où $\varepsilon > 0$ est la précision des calculs.

Méthode de la puissance

Le programme ci–apprès calcule une valeur approchée de λ_1 pour la matrice de Hilbert H définie par

$$H_{ij} = \frac{1}{i+j-1}, \quad i, j = 1, \ldots, n.$$

Tous les tableaux utilisés sont dynamiques. On utilise les fonctions intrinsèques suivantes

- `dble` convertit un entier en réel double
- `dot_product` produit scalaire de deux vecteurs
- `matmul` produit matrice/matrice ou matrice/vecteur avec les contraintes mathématiques usuelles sur le produit matriciel.

```fortran
program HILBERTVP
  implicit none
  ! Plus grande valeur propre de la matrice de Hilbert
  ! Methode de la puissance
  character(len=*), parameter :: fmt='("Iter=",I4,&
                                      &"  lambda=",E15.8,"  Err=",E15.8)'
  integer, parameter          :: IterMax=250          ! nb max d'iterations
  real(8), parameter          :: eps=1.d-8            ! precision du resultat
  real(8), dimension(:,:), allocatable :: a           ! matrice de Hilbert
  real(8), dimension(:),   allocatable :: v,q         ! vect. de travail
  real(8)             :: lambda                       ! plus grande val. prop
  real(8)             :: lambda1, erreur
  integer             :: n,i,j,iter

  ! lecture de la taille de la matrice
  print *,'Entrer la taille de la matrice de Hilbert '
  read *,n

  ! allocation des tableaux a,v,q,aq
  allocate(a(n,n))
  allocate(v(n))
  allocate(q(n))


  ! remplissage de la matrice de Hilbert
  do i=1,n
     a(i,:)=(/ (1.d0/dble(i+j-1), j=1,n) /)   ! utilisation d'une liste
  end do
```

```fortran
  ! initialisation de q tel que |q|=1
  q=0; q(1)=1.d0

  lambda1=1.d0; erreur=1
  iter=0;
  do while (erreur>eps .and. iter<IterMax)    ! boucle principale
     iter=iter+1
     v=matmul(a,q)                            ! v=Aq
     q=v/sqrt(dot_product(v,v))               ! q=v/|v|
     lambda=dot_product(q,matmul(a,q))        !  (Aq,q)
     erreur=abs(lambda-lambda1)/abs(lambda)   ! erreur relative
     lambda1=lambda
     print fmt,iter,lambda,erreur             ! affichage
  enddo

  print '("La plus grande valeur propre est : ",F15.10)',lambda
  print '("Nombre d''iterations necessaires : ",I4)',iter
end program HILBERTVP
```

Méthode de la puissance

Avec des tableaux statiques, il aurait fallu préciser les sections de tableaux concernés lors de l'appel des fonctions intrinsèques. Par exemple pour le produit scalaire, il aurait fallu écrire

```
dot_product(v(1:n),v(1:n))
```


Méthode de la puissance

Voici quelques exemples d'exécution.

```
Entrer la taille de la matrice de Hilbert
4
Iter=    1  lambda= 0.14911498E+01  Err= 0.32937658E+00
Iter=    2  lambda= 0.15000983E+01  Err= 0.59652573E-02
Iter=    3  lambda= 0.15002128E+01  Err= 0.76327551E-04
Iter=    4  lambda= 0.15002143E+01  Err= 0.97031100E-06
Iter=    5  lambda= 0.15002143E+01  Err= 0.12333994E-07
Iter=    6  lambda= 0.15002143E+01  Err= 0.15678210E-09
La plus grande valeur propre est :    1.5002142801
Nombre d'iterations necessaires :    6
```

Méthode de la puissance

```
Entrer la taille de la matrice de Hilbert
25
Iter=   1  lambda= 0.18439228E+01  Err= 0.45767793E+00
Iter=   2  lambda= 0.19431849E+01  Err= 0.51082167E-01
Iter=   3  lambda= 0.19511112E+01  Err= 0.40624852E-02
Iter=   4  lambda= 0.19517082E+01  Err= 0.30586424E-03
Iter=   5  lambda= 0.19517529E+01  Err= 0.22916229E-04
Iter=   6  lambda= 0.19517562E+01  Err= 0.17162877E-05
Iter=   7  lambda= 0.19517565E+01  Err= 0.12853585E-06
Iter=   8  lambda= 0.19517565E+01  Err= 0.96262572E-08
La plus grande valeur propre est :    1.9517565153
Nombre d'iterations necessaires :    8
```


Méthode de la puissance

```
Entrer la taille de la matrice de Hilbert
1000
Iter=   1  lambda= 0.19917362E+01  Err= 0.49792548E+00
Iter=   2  lambda= 0.23071073E+01  Err= 0.13669546E+00
Iter=   3  lambda= 0.24056663E+01  Err= 0.40969537E-01
Iter=   4  lambda= 0.24332490E+01  Err= 0.11335741E-01
Iter=   5  lambda= 0.24405716E+01  Err= 0.30003547E-02
Iter=   6  lambda= 0.24424821E+01  Err= 0.78220227E-03
Iter=   7  lambda= 0.24429781E+01  Err= 0.20301830E-03
Iter=   8  lambda= 0.24431066E+01  Err= 0.52628135E-04
Iter=   9  lambda= 0.24431400E+01  Err= 0.13638216E-04
Iter=  10  lambda= 0.24431486E+01  Err= 0.35339411E-05
Iter=  11  lambda= 0.24431508E+01  Err= 0.91569552E-06
Iter=  12  lambda= 0.24431514E+01  Err= 0.23726861E-06
Iter=  13  lambda= 0.24431516E+01  Err= 0.61479285E-07
Iter=  14  lambda= 0.24431516E+01  Err= 0.15930047E-07
Iter=  15  lambda= 0.24431516E+01  Err= 0.41276765E-08
La plus grande valeur propre est :    2.4431516130
Nombre d'iterations necessaires :   15
```

Entrées/Sorties de tableaux

Une instruction d'entrée–sortie peut contenir n'importe qu'elle forme de tableau (éléments, sous–tableau, tableau).

```
integer, dimension(5) :: m
real, dimension(5,5)  :: x
```

On peut seulement faire apparaître les éléments, comme dans

```
read  *, m(1), m(2)
print *, x(1,2), x(1,3), x(1,5)
```


Entrées/Sorties de tableaux

On peut aussi utiliser le nom du tableau ou des expressions tableaux, comme dans

```
read  *, m        ! equivalent a read *, m(1),m(2),m(3),m(4),m(5)
print *, x+1      ! equivalent a print *,x(1,1)+1,x(2,1)+1,x(3,1)+1,...
```

L'ordre de la liste est celui d'arrangement des éléments en mémoire, c'est–à–dire **colonne par colonne**.

On peut aussi utiliser une section de tableau. Ainsi les instructions

```
read  *, m((/1,4,2/))
print *, x(1:3,2)
```

sont équivalentes à

```
read  *, m(1), m(4), m(2)
print *, x(1,2), x(2,2), x(3,2)
```

Entrées/Sorties de tableaux

Toutefois, dans le cas d'une lecture, il faut éviter que, dans une même section, le même élément ne soit cité deux fois. Ainsi

```
read  *, m((/2,4,2/))
```

est **incorrect** car équivalent à

```
read  *, m(2),m(4),m(2)
```


4. Les tableaux 4.4. Section de tableau, vecteur d'indices

Entrées/Sorties de tableaux

Une alternative aux sections tableau est l'utilisation de listes implicites comme dans

```
read  *,  (m(i),i=1,4)
```

Pour obtenir l'affichage d'un tableau de rang 2 suivant l'ordre naturel, on combine liste implicite et boucle

```
do i=1,5
   print *,(x(i,j), j=1,5)
end do
```

Affichage de la matrice de Hilbert

```
program HILBERTMAT
  implicit none
  character(len=10), parameter  :: fmt='(8F15.8)'  ! constante format
  integer, parameter            :: nmax=10         ! taille max
  real(8), dimension(nmax,nmax) :: h               ! matrice de Hilbert
  integer                       :: n               ! taille reelle
  integer                       :: i,j

  print *,'Entrer la taille de la matrice '
  read *,n

  ! remplissage de la matrice de Hilbert
  do i=1,n
     do j=1,n
        h(i,j)=1.d0/dble(i+j-1)
     end do
  end do

  ! affichage
  print '(//"Matrice de Hilbert d''ordre : ",I3/)',n
  do i=1,n
     print fmt,(h(i,j),j=1,n)
  end do
end program HILBERTMAT
```


Affichage de la matrice de Hilbert

Voici quelques résultats d'exécution.

```
Entrer la taille de la matrice
3


Matrice de Hilbert d'ordre :   3

    1.00000000     0.50000000     0.33333333
    0.50000000     0.33333333     0.25000000
    0.33333333     0.25000000     0.20000000
```

Affichage de la matrice de Hilbert

```
Entrer la taille de la matrice
5


Matrice de Hilbert d'ordre :    5

    1.00000000     0.50000000     0.33333333     0.25000000     0.20000000
    0.50000000     0.33333333     0.25000000     0.20000000     0.16666667
    0.33333333     0.25000000     0.20000000     0.16666667     0.14285714
    0.25000000     0.20000000     0.16666667     0.14285714     0.12500000
    0.20000000     0.16666667     0.14285714     0.12500000     0.11111111
```


Affichage de la matrice de Hilbert

On décide de remplacer la déclaration de la constante de format par

```
character(len=10), parameter    :: fmt='(8E15.8)'  ! constante format
```

On obtient comme affichage:

```
Entrer la taille de la matrice
4


Matrice de Hilber d'ordre :    4

0.10000000E+01 0.50000000E+00 0.33333333E+00 0.25000000E+00
0.50000000E+00 0.33333333E+00 0.25000000E+00 0.20000000E+00
0.33333333E+00 0.25000000E+00 0.20000000E+00 0.16666667E+00
0.25000000E+00 0.20000000E+00 0.16666667E+00 0.14285714E+00
```