# Chapitre 6:







## Procédures et fonctions

- **Interface implicite** : contrôle limité de la cohérence des arguments
- **Interface explicite** : meilleur contrôle de la cohérence des arguments

- **Fonctions** ou sous-programmes "expression"
- **Procédures** ou sous-programmes "instruction"

- **Module** : fiabilise les communications entre programmes et sous-programmes

# Procédures : interface implicite

```
subroutine trinome(a,b,c,x1,x2)
  implicit none                    ! tres important
  real, intent(in)  :: a,b,c       ! arguments donnees (coefficients)
  real, intent(out) :: x1,x2       ! arguments resultats
                                   ! (racines eventuelles)
  real              :: delta       ! discriminant (variable locale)

  delta=b*b-4.0*a*c;
  if (delta<0) then
     print *,"Delta<0  : Pas de racines reelles"
     x1=0; x2=0
  else
     x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
  endif
end subroutine trinome
```





# Procédures : interface implicite

- En-tête (`subroutine`) : liste des arguments sans leurs types (**arguments formels**)
- **Arguments effectif** : ceux utilisés lors de l'appel de la procédure
- Déclaration à l'intérieur de la procédure
- Attribut `intent` : précise la vocation des arguments pour vérifications supplémentaires à la compilation

## Procédures : interface implicite

- `intent(in)` argument donnée.
  Sa valeur ne doit pas être modifiée dans la procédure.
  A l'appel, l'argument effectif peut être une variable existante, une constante littérale, une expression,...

- `intent(out)` argument résultat.
  La procédure ne doit pas utiliser sa valeur mais seulement lui en fournir une.
  A l'appel, l'argument effectif doit toujours être une variable existante.

- `intent(inout)` argument donnée et résultat.
  La procédure peut utiliser sa valeur mais doit aussi lui en fournir une.
  A l'appel, l'argument effectif doit toujours être une variable existante.





## Procédures : interface implicite

Pour l'appel, on utilise le mot clé `call`

```
program racine
  implicit none                        ! ne pas oublier
  real  :: a,b,c                       ! coef. du trinome
  real  :: x1,x2                       ! solutions eventuelles

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a, b, c                      ! lecture a,b,c

  call trinome(a,b,c,x1,x2)            ! appel a la procedure trinome

  print *,'Les racines sont : ',x1,x2 ! affichage solutions x1,x2
end program racine
```

## Procédures : interface implicite

La procédure `trinome` n'est visible du programme principal qu'a travers son en–tête. Il n'y a donc aucun contrôle de cohérence des arguments lors de la compilation.

```
call trinome(1,0,-4,x1,x2)  ! correct a,b,c const. dans la procedure
call trinome(1,0,a*a,x1,x2) ! correct var avec attribut intent(in)
call trinome(a,b,c,1.5,x1)  ! argument constante numerique --> DANGER
call trinome(a,b,c,i,j)     ! pas correct si i,j de type integer
call trinome(a,c,x1,x2)     ! pas correct oubli de b
```





## Procédure interne

- Procédure à l'intérieur du programme principal ou de procédures externes.
- Niveau d'emboîtement limité à 1.
- Accessible qu'à son hôte
- Exemple : programme `racine` et procédure interne `trinome`.

```
program racine
  implicit none                                  ! instruction globale
  real  :: a1,b1,c1                              ! variables globales
  real  :: y1,y2                                 ! variables globales

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a1,b1,c1                               ! lecture coef.

  call trinome(a1,b1,c1,y1,y2)                   ! appel a  trinome

  print *,'Les racines sont : ',y1,y2            ! affichage solutions

  !--- Fin partie executive de racine ------------

  contains                          ! mot cle indiquant la presence
                                    ! de sous-prog. internes

  subroutine trinome(a,b,c,x1,x2)
    real, intent(in)  :: a,b,c      ! arguments donnees (coefficients)
    real, intent(out) :: x1,x2      ! arguments resultats (racines eventuelles)
    real              :: delta      ! discriminant (variable locale)
    delta=b*b-4.0*a*c;
    if (delta<0) then
      print *,"Delta<0  : Pas de racines reelles"
      x1=0; x2=0
    else
      x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
    endif
  end subroutine trinome

end program racine   ! Fin de racine
```



# Procédure interne

- La procédure interne a accès à *toutes les variables* définies par son hôte.

- **Interface explicite** : meilleur contrôle de la cohérence des arguments.

```
call trinome(a,b,c,1.5,x1)  ! argument constante numerique --> DANGER
call trinome(a,b,c,i,j)     ! pas correct si i,j de type integer
call trinome(a,c,x1,x2)     ! pas correct oubli de b
```

## Procédure interne : inconvénients

2 inconvénients majeurs :

- Invisible de l'extérieur,
- Programmation lourde et non modulaire.





## Les fonctions

2 types de fonctions

- les fonctions intrinsèques, c'est–à–dire fournies avec le langage,
- les fonctions définies par l'utilisateur.

Définition d'une fonction voisine de celle d'une procédure.

Fonction interne ou externe.

# Fonctions externes : interface implicite

Forme générale d'une fonction externe :

```
[type] function nomfct(liste_arguments)
  implicit none
  [type nomfct]
  ! declaration arguments
  ...
  ! declaration variable locale
  ...
  nomfct=...    ! valeur de la fonction
end function nomfct
```

Le type du résultat de la fonction est précisé
– soit dans **l'en–tête**
– soit à **l'intérieur de la fonction**.



```
function fibonac(n)
  implicit none
  integer, intent(in) :: n          ! declaration argument donnee
  integer             :: fibonac    ! declaration resultat
  ! variable locales
  integer             :: u0, u1     ! 2 premiers termes de la suite
  integer             :: u2         ! terme courant
  integer             :: i

  u0=1; u1=1
  do i=2,n
     u2=u1+u0;
     u0=u1;
     u1=u2
  end do

  fibonac=u2                        ! resultat de la fonction fibonac
end function fibonac
```

```
program FIBO
  implicit none
  integer :: n                  ! terme de la suite
  integer :: fibonac            ! on declare le type de la fonction

  print *,"Entrer un entier >=2 "
  read  *,n

  print *,"Le n-ieme terme de la suite de Fibonacci est :",fibonac(n)

end program FIBO
```

L'exécution du programme `FIBO` donne:

```
Entrer un entier >=2
10
Le n-ieme terme de la suite de Fibonacci est : 89


Entrer un entier >=2
25
Le n-ieme terme de la suite de Fibonacci est : 121393
```



## Utilisation d'une fonction

On peut utiliser la fonction `fibonac` comme n'importe quelle fonction intrinsèque :

- dans une affectation simple comme

  `ii=fibonac(n)`

- dans une expression arithmétique

  `k=i*fibonac(10)+n*fibonac(n)`

Mais le compilateur doit connaître le type de la fonction `fibonac`.

C'est pour cela que dans le programme `FIBO` on déclare la variable `fibonac`.

## Fonctions internes : interface explicite

- Idem procédures,
- A placer entre les mots–clé `contains` et `end` du programme ou du sous–programme hôte,
- Plus besoin de déclarer le type de la fonction dans le programme hôte puisque ce dernier a une visibilité imprenable sur sa fonction interne.





## Variables locales à sous–programme

Dans un programme principal, les variables ont leur emplacement mémoire défini une fois pour toutes: elles sont dites **statiques**.

Les variables locales à un sous–programme sont gérées différemment :

- Emplacement mémoire attribuer qu'au moment où l'on commence à exécuter le sous–programme
- Emplacement libéré à la fin de l'exécution du sous–programme.

On dit que les variables locales sont **automatiques**.

Conséquence : *par défaut* leur valeur n'est pas conservée d'un appel à un autre.

## Variables locales à sous–programme

Imposer un emplacement permanent pour une variable locale et, ainsi, conserver sa valeur d'un appel au suivant.

- Utiliser l'attribut `save`,

```
subroutine ...
  ...
  integer, save :: p
  real, dimension(10), save :: x
  ...
```

- Initialisation de la variable locale à la déclaration.





## Variables locales à sous–programme

Voici en exemple, une procédure qui se contente de comptabiliser le nombre d'appels et de l'écrire.

```
subroutine JeMeCompte
  implicit none
  integer  :: nb_appels=0    ! variable locale -> statique

  nb_appels=nb_appels+1
  print *,'Appel No ',nb_appels
end subroutine JeMeCompte
```

# Tableaux en argument d'une procédure externe

- Rappel : la procédure externe est une unité de compilation indépendante,
- Problème : connaître le profil d'un tableau transmis en argument.
- Solution : transmettre les étendues du tableau.

Remarque : solution héritée du FORTRAN 77, on verra une solution plus adaptée à la programmation moderne.





# Tableaux en argument d'une procédure externe

Un vecteur en argument $\longrightarrow$ passer aussi la taille en argument.

ATTENTION à ne pas dépasser la dimension physique du tableau.

```
subroutine moyenne(x,n,som)
  implicit none
  integer                          :: n     ! declaration dim.
  real, intent(in), dimension(n) :: x     ! declaration vect. x
  real, intent(out)                :: som   ! moyenne

  ! variables locales
  integer            :: i

  som=0
  do i=1,n                    ! somme sur les elements x(i)
    som=som+x(n)
  end do
  som=som/real(n)             ! real convertit l'entier n en reel
 end
```

Programme appellant le sous–programme `moyenne` pour calculer la moyenne de *n* nombres.

```
program calcul_moyenne
  implicit none
  integer, parameter    :: nmax=100    ! dim. physique de x
  integer               :: n           ! dim. reelle du vect. x
  real, dimension(nmax) :: x           ! x(i), i=1,..,n
  integer               :: i
  real                  :: ss

  ! Lecture des donnees
  print *,'Entrer n : '
  read *, n

  do i=1,n
    print *,'Entrer x(',i,')'
    read *, x(i)
  end do

  ! Calcul de la moyenne
  call moyenne(x,n,ss)      ! on ne transmet que l'etendue reelle n

  print *, 'La moyenne est ',ss
end
```



## Tableaux en argument d'une procédure externe

- C'est l'adresse du premier élément à utiliser qui est transmise au sous-programme.
- La dimension indique le nombre d'éléments suivants à utiliser.
- On peut donc transmettre seulement un bout du vecteur :

```
...
n1=10
call moyenne(x(n-n1+1),n1,ss)
...
```

- Dans la procédure `moyenne`, on aura un vecteur à 10 composantes commençant à `x(n-n1+1)`, i.e. les 10 derniers éléments de `x`.

# Tableaux en argument d'une procédure externe

Pour les arguments *tableaux à plusieurs dimensions* il faut absolument **transmettre les dimensions physiques** et effectives.

Exemple : produit de deux matrices $A = (a_{ij})$ et $B = (b_{ij})$

$A$ est une matrice $m_a \times n_a$ et $B$ est une matrice $m_b \times n_b$.

Matrice produit $C = (c_{ij}) = A \cdot B$ est une matrice $m_a \times n_b$ dont les éléments sont donnés par la formule

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Ce produit matriciel n'est possible que si $n_a = m_b$.



```
program matrice
  implicit none        ! ne jamais oublier
  integer, parameter          :: nmax=100    ! dim. physique des tableaux
  real, dimension(nmax,nmax) :: A,B,C       ! declaration des tableaux
  integer                     :: ma,na       ! dim. reelles de A
  integer                     :: mb,nb       ! dim. reelles de B
  integer                     :: i,j

  ! Lecture des dimensions
  print *, 'dimension de A : ma na '
  read *, ma,na
  print *, 'dimension de B : mb nb '
  read *, mb,nb

  if (na /= mb) then
    print *,'Le produit matriciel est impossible'
    stop        ! on arrete le programme
  endif
  ...
  call ProduitMat(nmax,ma,na,nb,A,B,C)
  ...
end
```

```
subroutine ProduitMat(dmx,mx,nx,ny,x,y,z)
  implicit none    ! Ne jamais l'oublier
  integer      :: dmx       ! dim. physique des tableaux
  integer      :: mx,nx,ny  ! dim. reelles avec my=nx
  real, dimension(dmx,nx)  :: x
  real, dimension(dmx,ny)  :: y,z

  ! Variables locales
  integer      :: i,j,k

  do i=1,mx
  do j=1,ny
     z(i,j)=0.
     do k=1,nx
        z(i,j)=z(i,j)+x(i,k)*y(k,j)
     end do
  end do
  end do
end
```

Forme plus compacte pour `ProduitMat` utilisant la fonction intrinsèque `dot_product`.

```
subroutine ProduitMat(dmx,mx,nx,ny,x,y,z)
  implicit none                               ! ne jamais l'oublier
  integer      :: dmx                         ! dim. physique des tabealux
  integer      :: mx,nx,ny                    ! dim. reelles avec my=nx
  real, dimension(dmx,nx)  :: x
  real, dimension(dmx,ny)  :: y,z

  ! Variables locales
  integer      :: i,j

  do i=1,mx                                   ! x(i,1:nx) ligne i de x
    do j=1,ny                                 ! y(1:nx,j) colonne j de y
      z(i,j)=dot_product(x(i,1:nx),y(1:nx,j)) ! produit scalaire
    end do                                    ! ligne*colonne
  end do
end
```

## Tableaux en argument d'une procédure externe

Remarque : les matrices sont stockées en mémoire **colonne après colonne**.

En mémoire, **une matrice n'est qu'un long vecteur** composé de colonnes de la matrice.

Comme pour tout vecteur, un programme (ou un sous–programme) peut donc transmettre seulement une portion de ce «super–vecteur» à un autre.

Par exemple, si je veux faire la moyenne des éléments de la deuxième colonne de la matrice `A`

```
call moyenne(A(1,2),ma,ss)
```





## Bloc interface

- **Bloc interface** : évite les inconvénients de la procédure interne tout en conservant la fiabilité dans la transmission des arguments.
- Permet de donner une visibilité totale sur l'interface d'une procédure externe.
- Copie de la partie déclarative des arguments formels du sous–programme.
- Inséré dans chaque unité de programme faisant appel au sous–programme externe.

```
program racine
  implicit none
  real  :: a,b,c
  real  :: x1,x2

  !------------- BLOC INTERFACE  ------------------------------
  interface
    subroutine trinome(a,b,c,x1,x2)
      real, intent(in)  :: a,b,c      ! arguments donnees
      real, intent(out) :: x1,x2      ! arguments resultats
    end subroutine trinome
  end interface
  !------------------------------------------------------------

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a, b, c

  call trinome(a,b,c,x1,x2)    ! Appel a la procedure trinome

  print *,'Les racines sont : ',x1,x2
end program racine
```



## Bloc interface

- Visibilité totale du programme `racine` sur la procédure externe `trinome`.
- Meilleur contrôle de cohérence des arguments lors de la compilation.
- `trinome` reste une unité de compilation indépendante qui n'a pas accès aux variables du programme `racine`.

## Modules

Le bloc interface d'un sous–programme est à copier dans chaque unité de programme utilisant cette procédure :( .

Solution : le **module** ! :D

```
module nom_module
  [declaration]
  [ contains
      sous programmes exportables]
end module nom_module
```





## Modules

- La partie déclaration ne doit pas contenir de fonctions, ni de format, ni déclarer d'objets automatiques.
- Le mot clé `contains` indique la présence de sous–programmes exportables dans le module.
- Par défaut, toute entité déclarée dans un module est exportable.
- Un module peut réserver des entités pour son usage interne: déclaration `private` ou attribut `private`.

# Modules

```
module m_mod
  integer          :: k
  integer, private :: i,j      ! declaration avec attribut private
  private          :: sprog1   ! declaration private

  contains                     ! mot cle indiquant la presence
                               ! de sous-prog
  subroutine sprog1
    ...
  end subroutine sprog1

  subroutine sprog2
    ...
  end subroutine sprog2
end module m_mod
```





# Modules

Le mode d'accès par défaut est donc `public`. Mais ceci peut être inversé grâce à la déclaration `private` placée juste après l'en–tête du module.

```
module m_mod
  private
  integer, public   :: k       ! var. exportable
  public            :: sprog2  ! sous-prog. exportable

  contains                     ! mot cle indiquant la presence
                               ! de sous-prog
  subroutine sprog1
    ...
  end subroutine sprog1

  subroutine sprog2
    ...
  end subroutine sprog2
end module m_mod
```

# Modules

Avant d'être utilisé, un module doit être compilé séparément.

Module avec bloc interface

```
module bi_trinome
interface
    subroutine trinome(a,b,c,x1,x2)
      real, intent(in)  :: a,b,c      ! arguments donnees
      real, intent(out) :: x1,x2      ! arguments resultats
    end subroutine trinome
end interface
end module bi_trinome
```

`bitrinome.f` : fichier contenant le module avec bloc interface.

→ après compilation on obtient
`bitrinome.o` : le fichier objet
`bi_trinome.mod` : les informations du module (porte le **nom donné dans l'en–tête du module**).





# Modules

On accède aux entités d'un module grâce à l'instruction `use` qui utilise le fichier `bi_trinome.mod`.

Le programme `racine` utilisant la procédure `trinome` devient alors:

```
program racine
  !---------------- acces au module bloc interface -------------
  use bi_trinome
  !-------------------------------------------------------------
  implicit none
  real  :: a,b,c
  real  :: x1,x2

  print *,'Entrer les coef. a, b, c du trinome :'
  read *, a, b, c

  call trinome(a,b,c,x1,x2)       ! Appel a la procedure trinome

  print *,'Les racines sont : ',x1,x2
end program racine
```

# Modules

## Module avec sous–programme

```
module mtrinome

  contains                          ! mot cle indiquant la presence
                                    ! de sous-prog

  subroutine trinome(a,b,c,x1,x2)
    implicit none
    real, intent(in)  :: a,b,c      ! arguments donnees
    real, intent(out) :: x1,x2      ! arguments resultats
    real :: delta                   ! variable locale
    delta=b*b-4.0*a*c;
    if (delta<0) then
       print *,"Delta<0  : Pas de racines reelles"
       x1=0; x2=0
    else
       x1=-.5*(b-sqrt(delta))/a; x2=-.5*(b+sqrt(delta))/a
    endif
  end subroutine trinome
end module mtrinome
```





# Interface explicite : nouvelles possibilités

Une interface explicite est obtenue dans les cas suivants :

- sous–programme interne,
- bloc interface,
- module avec bloc interface,
- module avec sous–programme.

# Passage de tableaux de "profil implicite"

Plus besoin de se préoccuper du profil du tableau à transmettre.

Fonction intrinsèque : `size(array [,dim])`

Retourne la taille de `array` ou l'étendue de `array` dans la dimension indiquée via `dim`.



```
subroutine ProduitMat(x,y,z)
  implicit none                              ! ne jamais l'oublier
  real, dimension(:,:), intent(in)  :: x,y   ! rang de x, y
  real, dimension(:,:), intent(out) :: z     ! rang de z

  ! Variables locales
  integer      :: mx,ny                      ! dim. reelles avec my=nx
  integer      :: i,j                        ! compteur

  mx=size(x,dim=1)                           ! nb. de lignes de x
  ny=size(y,dim=2)                           ! nb. de colonnes de y

  do i=1,mx
  do j=1,ny
     z(i,j)=dot_product(x(i,:),y(:,j))       ! produit scalaire
  end do                                     ! ligne*colonne
  end do
end subroutine ProduitMat
```

```
program matrice
  implicit none                              ! ne jamais oublier
  integer, parameter         :: nmax=100     ! dim. physique des tableaux
  real, dimension(nmax,nmax) :: A,B,C        ! declaration des tableaux
  integer                    :: ma,na        ! dim. reelles de A
  integer                    :: mb,nb        ! dim. reelles de B
  integer                    :: i,j
  !------------------- BLOC INTERFACE --------------------------------
  interface
    subroutine ProduitMat(x,y,z)
      real, dimension(:,:), intent(in)  :: x,y  ! rang de x, y
      real, dimension(:,:), intent(out) :: z    ! rang de z
    end subroutine ProduitMat
  end interface
  !-------------------------------------------------------------------

  ! Lecture des dimensions
  print *, 'dimension de A : ma na '
  read *, ma,na
  print *, 'dimension de B : mb nb '
  read *, mb,nb

  if (na /= mb) then
    print *,'Le produit matriciel est impossible'
    stop        ! on arrete le programme
   endif
  ...
  call ProduitMat(A(1:ma,1:na),B(1:mb,1:nb),C(1:ma,1:nb))
  ...
end program matrice
```



## Tableaux automatiques

On peut faire varier les profils des tableaux locaux d'un appel à un autre.

Exemple : échange de la valeur de deux matrices

```
subroutine SwapMat(x,y)
  implicit none                                  ! ne jamais l'oublier
  real, dimension(:,:)  :: x,y                   ! matrices a echanger

  real, dimension(size(x,dim=1),size(x,dim=2)) :: z   ! matrice
                                                      ! auxiliaire

  z=x; x=y; y=z                                  ! C'est tout!!!

end subroutine SwapMat
```

```
program EchangeMatrice
  implicit none                    ! toujours present
  integer,parameter  :: nmax=100   ! taille max. des tableaux
  real,dimension(nmax,nmax) :: a,b ! tableaux a echanger
  integer                  :: n    ! taille reelle des tableaux

  !------------- bloc interface ----------------------------------
  interface
    subroutine SwapMat(x,y)
      real, dimension(:,:)  :: x,y   ! matrices a echanger
    end subroutine SwapMat
  end interface
  !-------------------------------------------------------------
  ...
  call SwapMat(a,b)                    ! on echange tout le contenu
  ...
  call SwapMat(a(1:n,1:n),b(1:n,1:n))  ! on echange que les blocs utiles
  ...

end  program EchangeMatrice
```



## Fonction à valeur tableau

Permet d'utiliser des fonctions directement dans des expressions tableaux.

Exemple : la matrice de Hilbert d'ordre $n$,

$$H_{ij} = 1/(i+j-1) \qquad i,j = 1,2,\ldots,n$$

```
function HilbertMat(n)
  implicit none                                  ! Ne jamais oublier
  integer, parameter  :: r8=kind(1.d0)           ! travail en double
  integer, intent(in) :: n                       ! taille matrice
  real (kind=r8), dimension(n,n) :: HilbertMat ! matrice resultat
                                                 ! (ajustable)
  ! variables locales
  integer i,j

  do i=1,n
     HilbertMat(i,:)=(/ (1.d0/dble(i+j-1),j=1,n) /)
  end

end function HilbertMat
```

# Fonction à valeur tableau

Utilisation dans n'importe quelle expression tableau
**à condition que son interface soit connue**.

```
program MHilbert
   implicit none
   integer, parameter :: nmx=100
   integer, parameter :: ir8=kind(1.d0)
   real, dimension(nmx,nmx) :: H

   integer            :: i, n=4
   !---------------------- Bloc interface ----------------------
   interface
      function HilbertMat(n)
        integer,parameter    :: r8=kind(1.d0)
        integer, intent(in) :: n
        real(kind=r8),dimension(n,n) :: HilbertMat
      end function HilbertMat
   end interface
   !------------------------------------------------------------
   ...
   H(1:n,1:n)=HilbertMat(n)+1   ! matrice de Hilbert d'ordre n + 1
   ...
end program MHilbert
```





# Arguments à mot clé

Repérer les arguments d'un sous–programme par le nom même de l'argument formel.

Exemple avec bloc interface de la procédure externe `trinome` :

```
interface
  subroutine trinome(a,b,c,x1,x2)
    real, intent(in)  :: a,b,c      ! arguments donnees
    real, intent(out) :: x1,x2      ! arguments resultats
  end subroutine trinome
end interface
```

les appels suivants seraient rigoureusement équivalents :

```
call trinome(a1,b1+a1,c1,x,y)              ! appel par position
call trinome(a=a1,b=a1+b1,c=c1,x1=x,x2=y)  ! par mot cle ordre
call trinome(c=c1,x1=x,a=a1,b=a1=b1,x2=y)  ! par mot cle desordre
call trinome(a1,a1+b1,c1,x2=y,x1=x)        ! mixage
```

# Arguments optionnels

- L'attribut `optional` permet de déclarer certains arguments comme optionnels.
- Leur présence lors de l'appel sera testée grâce à la fonction intrinsèque `present`.

Exemple : Soit à écrire une procédure `maxmin` qui cherche l'élément minimum et maximum d'un vecteur passé en argument. En option, on pourra avoir en sortie l'indice de l'élément minimum ou maximum. Nous avons choisi le module avec procédure comme interface explicite.



```
module mmaxmin
  contains
  subroutine maxmin(v,vmax,vmin,imax,imin)
    real, dimension(:),intent(in) :: v           ! vecteur a analyser
    real, intent(out)             :: vmax,vmin   ! elt. max. et min. dans v
    real, optional, intent(out)   :: imax,imin   ! arguments optionnels

    real, dimension(1)            :: rg          ! rg de taille 1!!!

    ! pour la recherche, on utilise les fct. intrinseques maxval et minval

    vmax=maxval(v)           ! maxval -> elt. max. dans v (intrinseque)
    vmin=minval(v)           ! minval -> elt. min. dans v (intrinseque)

    ! test de presence des arguments optionnels
    ! on utlise les fct. intrinseques maxloc et minloc

    if (present(imax)) then
       rg=maxloc(v)          ! maxloc -> indice elt. max. dans v (intrinseque)
       imax=rg(1)
    endif
    if (present(imin)) then
       rg=minloc(v)          ! minloc -> indice elt. min. dans v (intrinseque)
       imin=rg(1)
    endif
  end subroutine maxmin
end module mmaxmin
```

Les fonctions `maxloc` et `minloc` **renvoient un vecteur**.

# Arguments optionnels

```
program PROGMINMAX
   use mmaxmin           ! acces au module de maxmin
   implicit none
   integer, parameter    :: nmax=5
   real, dimension(nmax) :: v=(/ 1.,2.,9.,4,-8. /)
   real                  :: vmin,vmax
   integer               :: rgmin,rgmax

   !----- appel avec tous les arguments ---------
   call maxmin(v,vmax,vmin,rgmax,rgmin)
   !----- appel sans les arguments optionnels ---
   call maxmin(v,vmax,vmin)
   !----- appel sans rgmin ----------------------
   call maxmin(v,vmax,vmin,rgmax)
   !----- appel sans rgmax (mot cle indispensable)
   call maxmin(v,vmax,vmin,imin=rgmin)

   !-- impossible d'omettre un argument en cours de liste
   call maxmin(v,vmax,vmin,,rgmin)   ! INTERDIT

end program PROGMINMAX
```





# Partages de données

Placer les variables à partager dans un module.

- Pour y accéder depuis une unité de compilation, il suffit d'utiliser `use`.
- L'attribut `save` sera nécessaire si la variable à exporter n'est pas initialisée .

```
module mdonnees
  implicit none
  integer, parameter :: nmax=10             ! variable statique
  integer, dimension(nmax), save :: coeff   ! save obligatoire
end module mdonnees

subroutine impdonnees
  use mdonnees
  integer  :: i

  print '("Nb elements :",I4)',nmax
  print '("Coeff=",10I3)',(coeff(i),i=1,nmax)
end subroutine impdonnees
```

# Partages de données

```
program PARTGDONNEES
  use mdonnees        ! acces au module mdonnees
  integer    :: i

  coeff=(/(i,i=1,nmax)/)  ! initialisation du tab. coeff

  call impdonnees

end program PARTGDONNEES
```

Après compilation séparée et édition de liens, l'exécution donne :

```
Nb elements :  10
Coeff=  1  2  3  4  5  6  7  8  9 10
```





# Chapitre 7:

## Fonctions intrinsèques

- Quelques fonctions numériques intrinsèques
- Quelques fonctions mathématiques intrinsèques
- Quelques fonctions de précision
- Quelques fonctions relatives aux tableaux

## Quelques fonctions numériques intrinsèques

Voici quelques fonctions numériques usuelles.

| | |
|---|---|
| `abs(a)` | valeur absolue. |
| `dble(x)` | conversion en double précision. |
| `int(a)` | partie entière du réel `a`. |
| `min(a1,a2,...)` | valeur minimale des arguments. |
| `max(a1,a2,...)` | valeur maximale des arguments. |
| `mod(a,p)` | reste de la division de `a` par `p`. |
| `real(a)` | conversion en réel. |





## Quelques fonctions mathématiques intrinsèques

| | |
|---|---|
| `acos(x), asin(x)` | arccos$x$ et arcsin$x$, avec $\|x\| \leq 1$. |
| `atan(x)` | arctan$x$ |
| `cos(x), sin(x), tan(x)` | cos$x$, sin$x$ et tan$x$. |
| `cosh(x), sinh(x), tanh(x)` | cosh$x$, sinh$x$ et tanh$x$. |
| `exp(x), log(x), log10(x)` | $e^x$, $\ln x$ et $\log x$. |
| `sqrt(x)` | $\sqrt{x}$, $x \geq 0$. |

## Quelques fonctions de précision

Le résultat dépend du sous-type de l'argument fourni :

| | |
|---|---|
| `range(x)` | retourne la valeur entière maximale de l'exposant $r$ tel que $\|x\| < 10^r$ (entier) ou $10^{-r} < \|x\| < 10^r$ (réel) soit représentable. |
| `precision(x)` | retourne le nombre de chiffres décimaux significatifs. |
| `epsilon(x)` | retourne la quantité considérée comme négligeable devant 1. |
| `tiny(x)` | retourne la plus petite valeur réelle représentable. |
| `huge(x)` | retourne la plus grande valeur réelle représentable. |





## Interrogation sur le profil

| | |
|---|---|
| `shape(source)` | retourne le profil du tableau. |
| `size(array [,dim])` | retourne la taille ou l'étendue de la dimension indiquée via `dim`. |
| `lbound(array [,dim])` | retourne les bornes inférieures du tableau ou seulement de la dimension spécifiée via `dim`. |
| `ubound(array [,dim])` | comme `lbound` mais renvoie les bornes supérieures. |

## Interrogation sur le profil

Soit la déclaration suivante

```
integer, dimension(-2:27,0:49) :: T
```

Voici le résultat de l'application des fonctions précédentes au tableau `T`.

`shape(T)` vaut `(/30,50/)`
`size(T)` vaut `1500`
`size(T,dim=1)` vaut `30`
`ubound(T)` vaut `(/27,49/)`
`ubound(T,dim=2)` vaut `49`
`lbound(T)` vaut `(/-2,0/)`
`lbound(T,dim=1)` vaut `-2`





## Interrogation sur le contenu

Les fonctions

```
minloc(array [,mask])
maxloc(array [,mask])
```

fournissent un vecteur d'entiers dont les valeurs repèrent un élément respectivement minimal et maximal.

`mask`, s'il est présent, doit être un tableau logique conformant avec le tableau `array`.

En pratique, `mask` est un «filtre» sous forme d'expression logique de sorte que seuls sont pris en compte les éléments de `array` associés à une valeur `.true.` de `mask`.

## Interrogation sur le contenu

Soit le tableau d'entier suivant

```
integer, dimension(5) :: v=(/ 2,-1,10,3,-1 /)
```

Alors on a

`minloc(v)` vaut `(/ 2 /)`, i.e. `v(2)` élément minimal

`maxloc(v)` vaut `(/ 3 /)`, i.e. `v(3)` élément maximal

Les tableaux renvoyés par `minloc` et `maxloc` dans le cas ci–dessus sont **des vecteurs de taille 1**, i.e. le rang de `v`.





## Interrogation sur le contenu

Soit maintenant la matrice `a` déclarée comme suit

```
integer, dimension(0:2,-1:2) :: a
```

dont le contenu est

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

Alors on a

`maxloc(a,mask=a<5)` vaut `(/ 2,2 /)`,i.e. `a(2,2)` est le max des $a_{ij} < 5$

`minloc(a,mask=a>5)` vaut `(/ 3,3 /)`,i.e. `a(3,3)` est le min des $a_{ij} > 5$

## Fonctions de réduction

Les fonctions

```
minval(array [,dim][,mask])
maxval(array [,dim][,mask])
```

fournissent les **éléments extrémaux** du tableau `array`.

`mask` sert de «filtre» comme pour `minloc` et `maxloc`.

Si `dim` est présent, la recherche se fait sur toutes les sections de `array` qu'on peut obtenir en fixant tous les indices sauf celui relatif à la dimension spécifiée par `dim`.

Par exemple, pour une matrice,

- dim=1 revient à effectuer une recherche par colonne,
- dim=2 revient à effectuer une recherche par ligne.





## Fonctions de réduction

Soit le tableau `A` dont le contenu est

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

`minval(A)` vaut `1`

`maxval(A)` vaut `6`

`minval(A,dim=1)` vaut `(/ 1,3,5 /)`

`minval(A,dim=2)` vaut `(/ 1,2 /)`

`maxval(A,dim=1)` vaut `(/ 2,4,6 /)`

`maxval(A,dim=2)` vaut `(/ 5,6 /)`

`minval(A,dim=1,mask=A>1)` vaut `(/ 2,3,5 /)`

`maxval(A,dim=2,mask=A<3)` vaut `(/ 1,2 /)`

## Fonctions de réduction

Somme et produit des éléments d'un tableau.

```
sum(array [,dim][,mask])
product(array [,dim][,mask])
```

Le sous type résultat est celui de `array`.

Les arguments optionnels `dim` et `mask` fonctionnement de la même manière qu'avec `minval` et `maxval`.

Si `array` est vide ou si `mask=.false.` `sum` retourne `0` et `product` retourne `1`.





## Fonctions de réduction

Soient `x=(/ 2,5,-6 /)` et $A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

Alors on a

| | |
|---|---|
| `sum(x)` | vaut `1` |
| `product(x)` | vaut `-60` |
| `sum(A,dim=1)` | vaut `(/ 3,7,11 /)` (somme sur les colonnes) |
| `sum(A,dim=2,mask=A<2)` | vaut `(/ 1,0 /)` (somme sur les lignes, `mask=A<2` $\Rightarrow$ ligne 2 vide, `sum` retourne 0) |
| `product(A,dim=2)` | vaut `(/ 15,48 /)` |
| `product(A,dim=1,mask=A>4)` | vaut `(/ 1,1,30 /)` |

## Fonctions de multiplications

En Fortran 90, il existe deux fonctions de multiplication :

`dot_product(vector_a,vector_b)`

retourne le **produit scalaire** de deux vecteurs passés en argument.

`matmul(matrix_a,matrix_b)`

effectue le produit de *deux matrices* ou *d'une matrice et d'un vecteur*.
Les arguments doivent respecter les contraintes usuelles sur le **produit matriciel**.





## Fonctions de multiplications

Soit les vecteurs `v1=(/ 2,-3,-1 /)` et `v2=(/ 6,3,3 /)`.

`dot_product(v1,v2)` vaut `0`
`dot_product(v1(1:2),v2(1:2))` vaut `3`

Soient `v=(/ 2,-4,1 /)` et $A = \begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$

`matmul(A,v)` vaut `(/ 29,-7,10 /)`
`matmul(A(1:2,1:2),v(2:3))` vaut `(/ -18,5 /)`

# Fonctions de transformations

La fonction

```
transpose(array)
```

fournit la transposée d'**une matrice** (i.e. tableau de rang 2).

Donc si `array` est de profil `(/m,n/)`, le tableau resultat sera de profil `(/n,m/)`.