Les commandes de base de LINUX

Les commandes de gestion des répertoires et des fichiers

pwd (affiche le chemin absolu du répertoire courant)
ls (list, affiche les répertoires et les fichiers du répertoire actif)
ls (affiche seulement les noms)
ls toto* (affiche les fichiers commençant par toto)
ls -l (affiche le format long : types + droits + Nbre de liens + ...)
cd (change directory)
cp chemin (vers le répertoire dont le chemin absolu est donné)
cd .. (répertoire parent)
cd ~ (répertoire de base)
cd - (répertoire précedent)
cd / (répertoire racine)
cp (copie)
cp rapport*.txt sauvegarde
cp * dossier (copie
mv (move, renomme et déplace un fichier)
mv source destination
mv * dossier (déplace tous les fichiers du répertoire actif vers le répertoire dossier)
mkdir (créer un répertoire)
mkdir répertoire
rmdir (effacer un répertoire)
rmdir dossier (supprime un répertoire vide)
rm (remove, éfface!!!)
rm -R (enlèvement récursif!!!)
rm fichier
rm -i fichier (interactivement, avec demande de confirmation)
rm -f fichier (avec force, sans demande de confirmation)
rm -r fichier (avec récursivité, avec les sous répertoires)
rm -rf dossier (supprime le répertoire et tou son contenu, sans confirmation)

Les commandes de recherche

grep (recherche les occurences de mots à l'intérieur de fichier)
grep motif fichier
grep -i motif fichier (sans tenir compte de la casse)
grep -c motif fichier (en comptant les occurences)
grep -v motif fichier (inverse la recherche, en excluant le "motif")
grep expression /répertoire/fichier
grep [aFm]in /répertoire/fichier
grep "\$" *.txt

Les commandes d'édition

more ("pager" qui affiche page par page sans retour en arrière, "h" affiche l'aide contextuelle)

more fichier
more fichier1 fichier2
more *.txt

cat (concatenate avec le code de fin de fichier eof=CTRL + D)
cat fichier-un fichier-deux > fichier-un-deux
cat -n fichier > fichier-numéroté (crée un fichier dont les lignes sont numérotés)
cat -nb fichier (affiche sur la sortie standard les lignes numéroté, sauf les lignes vides)

head (affiche les 10 premières lignes d'un fichier)
head -n22 fichier (affiche les 22 premières lignes)

vi (l'éditeur en mode texte universel)

emacs (l'éditeur GNU Emacs multi fonction pour l'édition, les mails, les news, la programmation, la gestion des fichiers,...)

xemacs (l'éditeur GNU Emacs sous X)

diff (différence entre deux fichiers, utiles pour chercher les modifications)

diff fishier1 fichier2

Les commandes d'impression et de conversion

lp (la commande d'impression sur les systèmes Unix Système V)

lpr (la commande d'impression sur les systèmes BSD et Linux)
lpr fichier
echo $PRINTER

lpc status (affiche l'état de la file d'attente)

lpq (affiche les travaux d'impression et leur numéro)

lprm (supprime un travail d'impression avec son numéro comme argument)

gv ("ghostview" permet de visualiser des fichiers POST SCRIPT)
gv fichier.ps

a2ps (convertit les fichiers ASCII en POST SCRIPT)
a2ps -4 fichier -P fichier-post-script

Les commandes de compilation et d'execution

f77 compile un programme en fortran 77

f77 program.f (la terminaison .f indique que le fichier program.f est écrit en f77

./a.out exécution

Les autres commandes

cal (calendar)
cal 2002

date (affiche la date, le mois, l'heure et l'année du jour. Les messages d'erreur et les e-mails sont toujours datés avec la date système)
date -s

wc ("word & count", affiche le nombre de lignes + mots + caractères)
who | wc -l (affiche uniquement le nombre de lignes)

spell (programme de correction orthographique)

```
cat rapport.txt | spell > faute.txt
```

read (lit dans un script shell la ligne saisie à partir de l'entrée par défaut, le clavier)

L'Éditeur vi

vi est un éditeur de fichiers qui contiennent des lignes de texte. Il fonctionne en mode écran; le nom **vi** provient du mot *visual*. I

1 Quelques commandes essentielles

- Démarrer l'éditeur : **vi nom_du_fichier_à_éditer** (vous êtes en mode **commande**)
- Sauvegarder un fichier : **:w nom_du_fichier**
- Quitter l'éditeur en sauvegardant le fichier: **:x**
- Quitter sans sauvegarder : **:q!**

1 Commandes de base

- **Pour entrer du texte** :

  Ces commandes vous amènent en mode **insertion**.

  - **a** ajoute du texte à la droite du curseur
  - **i** insère du texte à la gauche du curseur
  - **o** intercale une ligne vide au-dessous du curseur
  - **O** intercale une ligne vide au-dessus du curseur

  En mode insertion :

  retour insère une fin de ligne

  Lorsque vous êtes en mode insertion taper ECHAP (ou ESC) pour revenir au mode commande

- **Pour remplacer du texte** :

  - **r** le caractère tapé remplace le caractère pointé par le curseur
  - **R** remplace plusieurs caractères [taper **ECHAP** (ou **ESC**) pour revenir au mode **commande**]

- **Pour déplacer le curseur dans le texte** :

  flèches pour se déplacer d'un caractère vers la gauche ou la droite, ou d'une ligne vers le haut ou le bas [ou utiiliser les touches **h** (gauche), **j** (bas), **k** (haut), **l** (droite)]

  Par ligne :

  - **0** se positionne au début de la ligne
  - **$** se positionne à la fin de la ligne
  - **retour** se positionne au premier mot de la ligne suivante

  D'un écran :

  - **^f** (peser simultanément sur les touches **CTRL** et **f**) avance d'un écran
  - **^b** (peser simultanément sur les touches **CTRL** et **b**) recule d'un écran

  Pour aller à une ligne en particulier :

  - **#G** positionne le curseur à la ligne **#**
  - **1G** positionne le curseur au debut du fichier
  - **G** positionne le curseur à la dernière ligne du fichier
  - **^g** (peser simultanément sur les touches **CTRL** et **g**) revèle le numéro de la ligne courante

- **:set nu** affiche les numéros de lignes

- **Pour enlever, remplacer ou copier une partie du texte** :

  x détruit le caractère pointé par le curseur et place dans le tampon

  #x détruit # caractères et place dans le tampon

  dd détruit la ligne courante et place dans le tampon

- **#dd** détruit # lignes à partir de la ligne courante et place dans le tampon

- **yy** copie la ligne courante dans le tampon

- **#yy** copie # lignes consécutives dans le tampon
- **p** insère le contenu du tampon à la droite du curseur (si 1 ou quelques caractères dans le tampon)

- **p** insère le contenu du tampon à la ligne suivante (si 1 ou quelques lignes dans le tampon)

- **P i**nsère le contenu du tampon à la ligne précédente

1 Commandes plus complexes

- **Recherche d'une chaîne de caractères particulière** :

- /**chaîne** cherche chaîne en avançant vers la fin du fichier;
- **n trouve** la prochaine occurrence de la dernière chaîne recherchée.

- **Substitution** :

- **:s**/**ceci**/**cela**/**options** substitue la première occurrence de **ceci** par **cela** dans la ligne courante ; l'option **g** substitue toutes les occurrences dans la ligne courante ; l'option **c** demande de confirmer la substitution
  :3,9s/**ceci**/**cela** remplace aux lignes**3 à 9** la première occurence de **ceci** par **cela**.
- **:%s**/**ceci**/**cela** remplace dans tout le fichier la première occurence de ceci par cela
- **:%s**/**ceci**/**cela**/**g** remplace dans tout le fichier toutes les occurences de **ceci** par **cela**

1 Et d'autres commandes encore !

- ~ change la casse (majuscules/minuscules) d'une lettre
- **J** joint la ligne courante à la suivante
- **.** repète la dernière commande
- **#commande** exécute une commande # fois
- **u** annule la dernière commande
- **U** annule les commandes affectant la ligne courante

- **:3,9d** élimine les lignes **3 à 9**

notes de cours Fortran 77

notes de cours Fortran 77 : table des matières

Les TP et TD 2004/2005 (sujet et correction) sont ici

1) Introduction

voir /pat/internet/techinfo

- structure de l'ordinateur (CPU, mémoire, interfaces, bus)
- les caractéristiques du CPU, les différentes vitesses dans un PC, les caches.
- les périphériques
- le langage machine (petit exemple), les 5 générations de langages

2) La gestion de l'ordinateur

- BIOS, OS, DOS
- les disques et leur gestion (FAT, répertoires, adressage absolu et relatif)
- les Files Systems (FAT12, 16, 32, Unix, NFS...)
- sous Unix : arborescence classique, montage/démontage, *mkdir*, *rmdir*, *cd*, *ls*, *rm*
- la compilation, l'édition de liens. En pratique (*kwrite prog.f & | gcc prog.f -o prog | prog*). Si vous cherchez un compilateur, je vous conseille le meilleur : GNU G77 (www.gnu.org). Il est disponible dans toutes les distributions Linux, pour Windows j'en ai fait une copie ici.

3) Bases du fortran 77

- le format de la ligne (C2345+789.....72CCCC) : en colonne 1, un C (ou *) indique que la ligne est un *commentaire* (ne sera pas prise en compte par le compilateur). Les colonnes 1 à 5 peuvent contenir un label (numéro de ligne par exemple). S'il y a un caractère en colonne 6 (un + par exemple), cette ligne est la continuation de la suivante. Les colonnes 7 à 72 contiennent les instructions. A partir de la colonne 73, plus rien n'est lu.
  Les compilateurs actuels (dont g77) acceptent un commentaire en fin de ligne : il commence par ! (et va jusquà la fin de la ligne). C'est même conseillé en premier caractère d'une ligne de commentaire (au lieu de C) pour la compatibilité avec F90 et plus.
- les *identificateurs* (noms des objets) : utilisez les caractères (A-Z,0-9), ni espace ni accent ni - (réservé à la soustraction). Le premier doit être une lettre, pas un chiffre. Le compilateur transforme toutes les minuscules en majuscules (sauf si entre apostrophes '). 6 caractères maxi (31 maxi en général, mais il vaut mieux qu'il n'y en ait pas deux avec les mêmes 6 premiers). Les identificateurs sont séparées par un ou plusieurs espaces, que l'on peut omettre si le caractère suivant n'est pas (A-Z,0-9).
- les types de variables de base : *integer* (en général jusqu'à +ou- 10^9), *real* (en général avec 7 chiffres significatifs), *double precision* (en général avec 15 chiffres significatifs). Remarque : on peut (quand on sait exactement ce qu'on fait) imposer le nombre d'octets des variables : integer *2 et *4, real *4 et *8). Il existe aussi les *character, logical* et *complex* (voir plus loin)
- les *opérateurs arithmétiques*, par priorité décroissante : (1) - unaire, (2) ** puissance, (3) * et /, (4) + et -. En cas de calcul entre deux entiers, le résultat est entier. Dès qu'il y a au moins un réel, le résultat est réel. Dès qu'il y a un double précision, le résultat est double précision.
- les constantes : 1 (entier), 100.0 ou 1E2 (réel), 1D0 (double précision)
- petit programme (premprog.f)

```
program calcul                                  entête
implicit none                 pour plus de sécurité, nous refusons
integer nb                          les déclarations implicites
real pu,pt
print *,'combien de pièces ?'                   écrire à l'écran
read *,nb                                  lire au clavier
print *,'prix unitaire ?'
read *,pu
pt=pu*nb                  affectation: le calcul est effectué puis
print *,'cela fera ',pt,' en tout'   le résultat est stocké dans pt
stop
end program calcul                              clôture
```

le résultat est (en rouge, tapé par l'utilisateur du programme):

```
 combien de pièces ?
5
 prix unitaire ?
1.25
 cela fera   6.25 en tout
```

structure du programme :

- entête (*program* nom au choix)
- déclarations (implicites et explicites) : les variables non déclarées commençant par (I-N) sont considérées entières, les autres réelles. Accepter les déclarations implicites est souvent source d'erreurs difficiles à localiser (un erreur de frappe crée une autre variable au lieu d'un message d'erreur), je vous conseille de toujours commencer par "*implicit none*"
- instructions : *print* affiche à l'écran, *read* lit au clavier. L'affectation se note : variable = expression, ou contenant = contenu, ou où = combien. Le programme calcule d'abord combien vaut l'expression à droite du =, puis stocke le résultat dans la variable nommée à gauche du =. "*stop*" arrête le programme, il est optionnel en dernière instruction du programme.
- clôture du programme (end est suffisant, mais pour plus de clarté on le complète par le nom du programme).

4) Le séquencement des instructions

4.1) type logical

Abordons d'abord le type "*logical*", ses valeurs .TRUE. .FALSE., les opérateurs associés (comparaisons .LT. , .GT., .LE., .GE., .EQ., .NE.; booléens .AND. .OR. .NOT.). Exemple : *test = (x.ge.3).and.(x.lt.10)* : x dans [3,10[.

4.2) l'instruction conditionnelle (if)

forme standard (archaïque)

```
IF (expression logique) instruction unique
```

si l'on veut rendre conditionnelle l'exécution de plusieurs lignes, f77 permet :

```
IF (expression logique) THEN
  une ou plusieurs instructions
ENDIF
```

Si l'on a plusieurs condition EXCLUSIVES on peut même :

```
IF (expression1) THEN
  une ou plusieurs instructions
ELSEIF (expression2) THEN
  une ou plusieurs instructions
ELSEIF (expression3) THEN
  une ou plusieurs instructions
ELSE
  une ou plusieurs instructions
ENDIF
```

Le dernier ELSE (obligatoirement sans expression logique) est optionnel.

Encore plus complexe, parmi les instructions conditionnelles on peut utiliser des *if ... endif*. On nomme cela des *if imbriqués*. Un *endif* correspond toujours

au dernier *if* qui n'en était pas pourvu.

exemple (il manque l'entête, les déclarations, la saisie des coefficients et surtout un test) :

```
      delta=b*b-4*a*c
      if (delta.GT.0) then
        x1=(-b-sqrt(delta))/(2*a)
        x2=(-b+sqrt(delta))/(2*a)
        print *,'2 racines: ',x1,' et ',x2
      elseif(delta.EQ.0) then
        x=-b/(2*a)
        print *,'1 racine double: ',x
      else
        print *,'aucune racine réelle'
      endif
```

(voir seconddeg.f)

Remarque sur la division : il faut TOUJOURS vérifier que le dénominateur est non nul (le célèbre domaine de définition cher à tous vos profs de maths depuis le collège).

4.3) les boucles

forme standard (archaïque)

```
      DO label indice=val.initiale, val.finale, incrément
        une ou plusieurs instructions
label   CONTINUE
```

Le label est un entier dans]0,99999] (doit rentrer dans les 5 premiers caractères de la ligne). L'incrément est optionnel (1 par défaut). La boucle sera effectuée avec l'indice (que vous aurez déclaré avant en *integer*) allant de '*val.initiale*' à '*val.finale*' compris, par pas de '*incrément*' . Il ne faut JAMAIS tenter de modifier l'indice, val.initiale, val.finale ou incrément en cours de boucle. En sortie de boucle, l'indice ne vaut pas nécessairement la valeur finale, ça dépend des compilateurs.

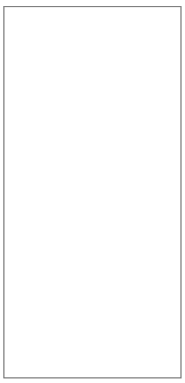
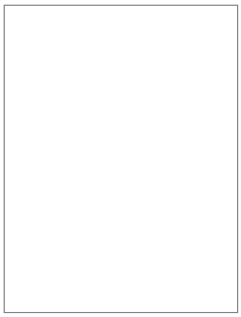
exemple :

```
      do 10 i=1,10
      print *,i,'è coucou'
10      continue
```

qui donne

```
1è coucou
2è coucou
3è coucou
4è coucou
5è coucou
6è coucou
7è coucou
8è coucou
9è coucou
10è coucou
```

En fait, le *continue* ne peut fermer que la dernière boucle ouverte (mais on peut imbriquer des boucles). Le label n'apporte rien (que peut-être la clarté).

exemple :	boucles imbriquées : OK	chevauchement impossible :
DO 10 ----- *-----* *do 20 ----* *-----* *20 continue* *do 30 ----* *-----* *30 continue* *-----* *10 continue*		

Fortran77 propose donc une écriture sans label :

```
DO indice=val.initiale, val.finale, incrément
  une ou plusieurs instructions
ENDDO
```

Mais ces boucles ne peuvent se faire que si leur nombre est fixé d'avance. C'est pourquoi la plupart des compilateurs acceptent d'autres boucles, mais elles ne sont pas standardisées dans fortran 77. Celle qui me semble la plus fréquente (et acceptée par les normes suivantes) est :

```
DO while (condition logique)
  une ou plusieurs instructions
ENDDO
```

exemple :

```
DO while(x.gt.1)
  x=x/2
ENDDO
```

si x est inférieur à 1 avant de commencer la boucle, aucune division par 2 n'est effectuée.

Deux instructions spécifiques aux boucles ont été rajoutées dans G77 : *EXIT* (sortir immédiatement de la boucle) et *CYCLE* (passer directement à la prochaine itération).

Autres écritures de boucles existantes, suivant les compilateurs (mais je ne vous les conseille pas) :

```
while (condition) do
  instructions
enddo

do
  instructions
while(condition)

do
  instructions
```

```
until (condition)
```

4.4) le saut (goto)

```
GOTO label
  ------
label   CONTINUE
```

Le label doit être unique dans un programme, il peut se situer au dessus ou au dessous du GOTO. L'instruction en face du label n'est pas nécessairement un *continue*, ce peut être n'importe quelle autre instruction (qui sera exécutée lors du saut). Les goto sont créateurs de nombreux problèmes, (dans les programmes complexes ou buggés, uniquement) c'est pourquoi les programmeurs expérimentés essayent de s'en passer. Plusieurs GOTO différents peuvent sauter au même label. On peut sortir par GOTO de n'importe où, mais on ne peut pas pointer vers l'intérieur d un bloc d'instructions (IF, DO, Subroutine)

exemple :

```
1       do 10 i=deb,fin
        -----
        GOTO 5: ici un goto 10 passe au prochain i, goto 1 recommence à deb.
        -----
5        continue
        -----
10      continue
       ----
       GOTO 5: d'ici, ce GOTO est impossible. Mais GOTO 1 le serait
```

Une écriture fréquente est le IF (condition) GOTO label, servant par exemple pour créer des boucles "tant que" qui soient compatibles avec tout compilateur.

5) Les sous-programmes et fonctions

Un sous-programme est le regroupement sous un même nom d'un groupe d'instructions (et de variables, dites locales). Le programme et les sous-programmes partagent des variables via ce que l'on appelle les paramètres (et à la rigueur le COMMON, voir plus loin)

Déclaration d'un sous-programme :

```
SUBROUTINE nom (liste d'arguments formels)
déclarations locales (y compris arguments formels)
instructions (dont RETURN pour quitter la subroutine, optionnel en
        dernière instruction)
END SUBROUTINE nom
```

appel d'un sous-programme :

```
CALL nom (liste d'arguments réels)
```

exemple :

```
SUBROUTINE echange (x,y)
implicit none
```

```
real x,y,tampon
tampon=x
x=y
y=tampon
RETURN
END SUBROUTINE echange
```

autre part :

```
real a,b
a=10
b=15
CALL echange (a,b)
```

Les arguments réels doivent être exactement du même type que les arguments formels, aussi nombreux et dans le même ordre (même pas de transformation automatique d'entier en réel). Les arguments sont dits "passés par adresse", ce qui signifie que toute modification à l'argument formel est répercutée à l'argument réel (j'en parle parce que dans d'autres langages c'est différent).

Une fonction, quand à elle, a un résultat. Elle est appelée dans une expression

```
type FUNCTION nom (liste d'arguments formels)
déclarations locales (y compris arguments formels)
instructions (dont RETURN pour quitter la fonction, précédé d'une
        affectation au nom de la fonction)
END FUNCTION nom
```

exemple:

```
real function puiss(x,n)
implicit none
real x,p
integer i,n
p=x;
do i=2,n
      p=p*x
enddo
puiss=p
return
end function puiss
```

appel :

```
real z,puiss
z=2.3
print *, 'calcul: ', puiss(z,10)+1
```

Remarque : ici z reste à 2.3, car dans la fonction x est inchangé. Si l'on avait modifié x dans la fonction, on aurait obtenu ce que l'on appelle un "effet de bord".

Exemple de sous-programme à effet de bord :

```
real function factorielle(n)
integer res,n
res=1
do while (n.GT.1)
      res=res*n
```

```
        n=n-1
    enddo
    factorielle=res
    return
    end function factorielle
```

Cette fonction calcule bien les factorielles, mais ces instructions :

```
integer nb,factorielle
nb=5
print *, factorielle(nb),' est la factorielle de ', nb
```

donneront ce résultat

```
120 est la factorielle de 1
```

C'est assez déplorable, nb ne vaut plus 5 ! Conclusion : ne modifiez un argument formel que si c'est nécessaire.

Dans le programme appelant la fonction, son type doit être déclaré. Dans l'exemple de la factorielle si vous l'oubliez (et avez oublié "*implicit none*"), il la considérera *real*, ce qui donnera des résultats aberrants !

6) Les tableaux

On peut regrouper sous un seul nom plusieurs variables simples, c'est un tableau unidimensionnel ou vecteur. On les déclare ainsi :

```
type nom (indice début: indice fin)
```

La plupart du temps, on omet l'indice début (qui sera 1), l'indice fin sera alors aussi le nombre de composantes. Il existe dans ce cas une seconde méthode pour déclarer le tableau :

```
type nom
dimension nom(nombre)
```

exemple :

```
real vecteur(3)
```

ou

```
real vecteur
dimension vecteur(3)
```

On utilise les composantes d'un tableau dans toute expression par

```
nom(indice)
```

exemple

```
x=(vecteur(i)-vecteur(1))/vecteur(i+1)
```

Attention, c'est au **programmeur** de vérifier qu'il utilise toujours des indices dans les limites déclarées ! La dimension de tous les tableaux doit être connue du compilateur, pour qu'il puisse gérer la mémoire utilisée par le programme,

avant même que la première instruction n'ait été exécutée . La dimension doit donc être une constante, elle ne peut pas être une variable que vous demanderiez à l'utilisateur par un *read*. Par contre rien ne vous empêche de déclarer une dimension assez importante, et n'utiliser qu'une partie du tableau, variable à chaque utilisation du programme, mais toujours inférieure à la dimension déclarée.

On ne peut, dans un programme, que traiter les composantes d'un tableau, pas son intégralité (par exemple, pour copier un tableau il faut, dans une boucle, copier toutes ses composantes). Par contre on peut passer un tableau en argument d'un sous-programme ou fonction.

exemple :

```
real x(5)
print *, 'moyenne: ',moy(x,5)
---------
real function moy(x,nb)
real x(*),som
integer nb,i
som=x(1)
do i=2,nb
  som=som+nb(i)
enddo
moy=som/nb
end function moy
```

Dans la déclaration, la dimension du tableau n'a besoin d'être donnée que dans l'entité qui réserve la place nécessaire en mémoire, pas pour ceux qui reçoivent le tableau en argument, qui même si on leur donne la dimension ne s'en servent pas. C'est pourquoi il vaut mieux indiquer * (certains programmeurs dimensionnent à 1 les tableaux reçus en arguments, mais c'est peu clair). Une autre solution est de la passer en argument (ou même une dimension inférieure à la dimension réelle, mais pas plus grande), dans la fonction ci-dessus on déclarerait `real x(nb)`.

Vous pouvez voir ici un programme de base sur les tableaux (tab-sp.f)

Pour représenter une matrice, on utilise un tableau bidimensionnel. La déclaration est :

```
type nom (deb lig:fin lig , deb col:fin col)
```

l'appel est

```
nom(indice ligne, indice colonne)
```

Dans la pratique, le compilateur range en mémoire toute la première ligne de la matrice, puis la seconde, la troisième... Mais tant que vous n'êtes pas spécialistes, il vaut mieux l'ignorer. Rappelez-vous qu'en général le premier indice est la ligne, le second la colonne (licol).

On peut étendre à des dimensions supérieures, mais en se limitant à la 7è dimension. Exemple :

```
integer rubiks(3,3,3)
```

Attention, en cas de passage en arguments de tableaux multidimensionnels, toutes les dimensions doivent être déclarées exactement, sauf la dernière qui peut être * (en général on les donne toutes). Mais la solution idéale est de passer les dimensions en arguments (obligatoirement la dimension exacte, sauf pour la dernière qui peut être inférieure si c'est vraiment utile).

exemple :

```
program matrice
integer maxlig,maxcol
parameter(maxlig=5,maxcol=3)
real m(maxlig,maxcol)
integer il,ic
do il=1,maxlig
 do ic=1,maxcol
  m(il,ic)=il+(ic/10.0)
 enddo
enddo
call affmat(m,maxlig,maxcol)
end

subroutine affmat(mat,diml,dimc)
implicit none
integer diml,dimc
real mat(diml,dimc)
integer il,ic
do il=1,diml
  print *,('M(',il,ic,')=',m(il,ic),ic=1,dimc)
enddo
end
```

7) Déclarations particulières : constantes, initialisation

On peut déclarer une constante, c'est à dire une valeur fixée par le programmeur, qui ne pourra pas changer lors de l'exécution du programme. On s'en sert principalement pour déclarer les tailles de tableau (ce qui permettra par la suite de la changer plus facilement)

exemples :

```
real tva
parameter (tva=19.6)
integer largeur,hauteur
parameter (largeur=5,hauteur=2)
real table(largeur,hauteur)
do i=1,largeur
   ----etc -------
call subroutine(table, largeur, hauteur)
   ----etc -------
```

Il est également possible d'initialiser des variables. C'est à dire fixer une valeur initiale à une variable, mais cette valeur pourra changer au cours du programme. On utilise pour cela la déclaration DATA :

```
DATA liste de variables / liste de valeurs initiales /
```

exemple :

```
real x,y
data x,y / 0.0 , 0.5 /
```

on peut remplacer une suite de valeurs identiques par "nombre*valeur". On peut également initialiser des tableaux :

```
real t(2,5)
data t / 5*0.0, 1.0, 4*0.0 /
```

Même si un *data* se trouve dans une subroutine, l'initialisation ne sera effectuée qu'une fois, au début de l'exécution du programme.

8) Les chaînes de caractères

On peut stocker des caractères dans une "chaîne", qui est en fait un tableau de caractères. On le déclare par CHARACTER*dim NOM. Comme tout tableau, la dimension est importante et doit être définie lors de l'écriture du programme. Si l'on n'utilise pas la totalité de la dimension déclarée, le compilateur complète avec des espaces (qu'il rajoute en fin de chaîne).

Une constante chaîne de caractères est délimitée par des apostrophes ('). Pour inclure une apostrophe dans une chaîne, il faut la doubler (''). // est l'opérateur ce concaténation : il permet de "coller" deux chaînes pour n'en former qu'une. On peut extraire une partie d'une chaîne : nom(début:fin).

exemples :

```
character*10 x,y
character*20 z
x='bonjour'
x(4:6)='soi'
y='c''est moi'
y(7:7)=y(5:5)
z=x//y
```

ce qui donne dans z : "bonsoir c'est toi"

On peut créer des tableaux de chaînes. Par exemple : character*10 txt(20)

On peut aussi comparer des chaines. Par .EQ. et .NE. mais aussi à l'aide des fonctions LGE, LGT, LLE ou LLT. Par exemple, LGE(c1,c2) est .TRUE. si a1 est supérieur ou égal à c2 (dans l'ordre alphabétique).

Il existe d'autres fonctions spécifiques aux chaînes de caractères : ichar(i) donne le caractère dont le code ASCII est l'entier i, char(c) donne le numéro de code du caractère c. Pour trouver la position de la sous-chaîne sc dans la chaîne plus longue lc : index(lc,sc).

9) Les changements de type et fonctions intrinsèques

9.1) mélanges de types

Toutes les variables sont stockées en mémoire sous forme d'une suite de 0 et de 1. Mais chaque type de variable est codé différemment. Pour info, les entiers sont en binaire signé, les réels sont en virgule flottante 32 bits et les double précision en 64 bits. Une même valeur sera donc codée par une suite de 0 et de 1 différente suivant qu'elle est dans une variable entière, réelle ou double précision. Le compilateur ne sait faire des opérations arithmétiques (+ - * / **) que sur des valeurs codées de la même manière. Si vous lui demandez un calcul entre deux variables de type différent, le compilateur doit avant tout en convertir une pour qu'elles soient de même type. Il sait convertir automatiquement un entier en réel et un réel en double précision (mais c'est tout). Donc, suivant le type des deux opérandes, le résultat de l'opération sera :

	integer	**real**	**double précision**
integer	integer	real	double précision
real	real	real	double précision
double précision	double précision	double précision	double précision

La conversion n'est faite que lorsque c'est nécessaire. Dans la plupart des cas, c'est la meilleure solution car la plus rapide. Il n'y a que dans le cas de la division que le résultat est différent suivant les cas. En effet, la division de deux entiers donne un entier (et un reste, mais il est ignoré). Par exemple, si x et y=2 sont réels, i=5 et j=3 sont des entiers, alors x=y+i/j met 3 dans x : la division est prioritaire à l'addition, donc on l'effectue en premier. Les deux opérandes i et j sont de même type donc 5/3 donne 1 (reste 2); y et 1 sont de type différent donc 1 est converti en 1.0, et additionné à y pour donner 3.0 qui est stocké dans x.

De même, l'affectation (variable=calcul) calcule d'abord l'expression à droite du signe =, ce n'est qu'après qu'il regarde le type de la variable devant recevoir le résultat pour faire, s'il le faut, une conversion. Vous pouvez donc écrire x=i mais pas i=x.

exemple :

```
double precision x,y,z
x=5/3
y=5.0/3.0
z=5D0/3D0
print 10,x,y,z
```

10 format (f18.15)

affichera :

1.000000000000000
1.666666626930237
1.666666666666667

ATTENTION : la conversion n'est automatique que dans ces cas. En particulier, elle n'est pas effectuée entre les arguments réels et formels d'un sous-programme ou fonction

9.2) conversion explicite

Vous pouvez par contre indiquer au compilateur quelles conversions il doit effectuer. Pour cela, on utilise les fonctions suivantes :

argument résultat	**integer**	**real**	**double précision**
integer (partie entière)		ifix (ou int)	idint
integer (entier le plus proche)		nint	idnint
real	float (ou real)		sngl
double précision	dfloat	dble	

Ici aussi, faites attention aux divisions : float(5/3) ne vaut pas float(5)/float(3)

9.3) fonctions intrinsèques

En plus des fonctions de conversion, Fortran propose diverses fonctions, qu'il est inutile de déclarer puisque déjà connues. Toutes les fonctions ci-dessous retournent une valeur de même type que leur argument.

	integer	**real**	**double précision**
racine carrée		sqrt	dsqrt
exponentielle		exp	dexp
log népérien		alog	dlog
log base 10		alog10	dlog10

valeur absolue	iabs	abs	dabs
minimum	min0	amin1	dmin1
maximum	max0	amax1	dmax1
sinus		sin	dsin
cosinus		cos	dcos
tangente		tan	dtan
arccos		acos	dacos
arcsin		asin	dasin
arctan		atab	datab

10) Variables communes

Pour que différents sous-programmes puissent échanger des données, la solution la plus évidente est de les passer en arguments. Mais quand un certain nombre de données doivent être utilisées dans toutes les parties d'un programme, on choisira plutôt le common. Il consiste à donner un nom à une liste (ordonnée) de variables à partager.

```
common/nom/variable1,variable2,...
```

Le common sera déclaré dans chaque entité de programme (sous-programme, fonction) qui doit accéder aux variables communes. Le nom définit le common. Les variables communes par contre sont définies par leur ordre, et pourraient avoir des noms différents dans toutes les entités (mais je vous le déconseille fortement), et doivent y être déclarées. Il peut y avoir plusieurs common différents, mais une variable ne peut appartenir qu'à un seul common.

Exemple : dans un programme d'analyse de l'évolution de la température, toutes les entités doivent connaître les différentes températures (et leur nombre)

```
      real maxi()
      common/temp/nb,t
      integer nb,i
      real t(100),m
      m=t(1)
      DO 100 i=2,nb
            if(t(i).gt.m)m=t(i)
100     continue
      maxi=m
      return
      end
c
      program main
```

```
common/temp/nb,t
integer nb
real t(100)
---- etc ----
end
```

Les variables des différents common peuvent être initialisées (si nécessaire) dans une entité nommée bloc data qui ne peut comporter que des common, des déclarations et des data :

```
block data
common/temp/nb,t
integer nb
real t(100)
data nb /0/
end block data
```

Ne mettez qu'un seul block data dans un programme !

Pour éviter de recopier partout les mêmes common, on peut utiliser l'instruction

```
INCLUDE 'nomd1fichier'
```

Le fichier désigné est inclus à cet endroit. Attention, cette possibilité n'est pas disponible partout, mais sous g77 elle l'est (comme d'ailleurs toules les directives # du C).

11) Entrées, sorties et formats

11.1) ouverture et fermeture du flux

Le programme peut lire (resp. écrire) des données depuis (resp. vers) l'extérieur par l'intermédiaire d'une "unité logique". Une unité logique (un flux) correspond à une interface de l'ordinateur qui transmet un flot de données entre la carte mère et un de ses périphériques. Ce sont principalement les fichiers sur disque, le clavier et l'écran, mais peut aussi être un lecteur de bandes, un port RS232, la carte son,... à condition que le système d'exploitation y donne accès (dans /dev sous Unix).

La première chose à faire avant d'utiliser un flux est de l'ouvrir. C'est une sorte de "demande de réservation" faite à l'OS, qui peut la refuser (si il est déjà ouvert par un autre programme, si il n'existe pas, si vous n'avez pas les bons droits d'accès,...). Chaque flux ouvert est associé à un "numéro d'unité logique" qui est un entier positif (en général entre 9 et 99), qui nous servira à désigner le flux dans la suite du programme. Dans le cas d'un fichier, l'ouverture se fait par l'instruction OPEN :

```
open(UNIT=numéro,FILE='nom du fichier',STATUS='état',ERR=label,IOSTAT=variable entière)
```

- Si le numéro est donné en premier, il n'est pas nécessaire de le précéder par UNIT=. C'est le programmeur qui fixe le numéro, il doit être unique dans le programme et ne peut être variable (mais un parameter est

possible).

- Le nom du fichier peut être donné directement entre apostrophes, ou être une variable chaîne (c'est également le cas pour d'autres arguments mais moins courant).
- Le statut est optionnel, il vaut soit NEW si le fichier doit obligatoirement ne pas exister avant (pour ne pas l'écraser), soit OLD s'il doit exister avant (pour la lecture par exemple), soit SCRATCH si le programme doit le créer le temps du programme et le détruire à la fin, soit UNKNOWN (ou ne rien mettre) s'il doit l'ouvrir, après l'avoir créé si nécessaire. APPEND permet de se placer à la fin du fichier (pour rajouter du texte à la suite).
- En cas de refus d'ouverture, la variable désignée par IOSTAT reçoit un code d'erreur (0 si ok), et le programme saute directement au label désigné par ERR (s'il sont omis, le programme s'arrête en cas d'erreur).
- On peut aussi utiliser le fichier en accès direct (ACCESS='DIRECT') en imposant la taille des enregistrements (RECL=entier), qui seront soit directement écrits en binaire dans le fichier (FORM='UNFORMATED') soit traduits en ASCII (FORM='FORMATED').

Il y a deux unités qui sont automatiquement ouvertes au début du programme : 5 pour le clavier et 6 pour l'écran.

Lorsque vous avez fini d'utiliser un flux, il faut le refermer, en particulier pour en donner accès aux autres. Tout flux non encore fermé le sera automatiquement à la fin du programme. Pour cela, on utilise l'instruction :

```
close(unit=numéro,IOSTAT=variable, ERR=label)
```

exemple :

```
open(10,FILE='resultats.txt')
---- etc ----
close(10)
```

11.2) lecture - écriture

Pendant qu'il est ouvert, on écrit dans un flux par

```
write (unité,format) liste de variables
```

et l'on lit dans un flux par

```
read (unité,format) liste de variables
```

Ces deux écritures acceptent un format plus complet :

```
(UNIT=unité, FMT=format, IOSTAT=variable, ERR=label, END=label)
```

Les options sont identiques à l'open, plus END qui donne le label où doit sauter le programme quand on est arrivé à la fin du fichier (pratique quand on n'en connaît pas la longueur).

En cas d'écriture à l'écran, l'unité est 6, mais on peut l'appeler *. On peut même utiliser

```
print format, liste de variables
```

Pour la lecture au clavier, l'unité est 5, mais on peut l'appeler *. On peut également utiliser :

read format, liste de variables

La liste de variables contient des variables ou expressions séparées par des virules, mais on peut également utiliser une "boucle implicite" :

(expression contenant un indice, indice=début,fin,pas)

Par défaut, le pas vaut 1.

exemples :

```
read *,a,b
write (*,*) 'a vaut',a,'et b vaut',b
write (6,100) a,b,( t(i),i=1,10),c
print *, ( (i+3*j , i=1,3),' -',j=0,4)
```

la dernière ligne affichera :

1 2 3 - 4 5 6 - 7 8 9 - 10 11 12 - 13 14 15 -

11.3) déclaration du format

Le format peut-être soit * (format libre : le compilateur choisit le format d'affichage qui lui paraît le plus adapté), soit un label (format imposé : au label donné, le programmeur déclare le format d'affichage), soit une variable chaîne de caractères (format calculé : le format est créé par le programme, et peut donc varier suivant les calculs précédents).

La déclaration du format imposé (qui doit être dans la même entité de programme) est de la forme :

label FORMAT(liste de composants)

Les composants du format sont :

- 'texte entre apostrophes' : le texte est écrit tel quel
- In : entier sur n caractères (avec espaces devant si le nombre est plus petit)
- An : chaîne sur n caractères (avec rajout d'espaces en fin si nécéssaire)
- Fn.p : réel sur f caractères dont p après la virgule (F5.2 donne 12.45)
- En.p : réel en notation scientifique dont p pour la partie décimale (l'exposant en prend 4, ne pas oublier le . et peut-être le signe -) (E9.2 donne -2.45E+09)
- Gn.p : réel en Fn.p si possible, En.p s'il est trop grand
- Dn.p : idem En.p mais pour un réel double précision
- X : un espace (ou ignorer un caractère, dans le cas d'un read)
- / : un retour à la ligne (ou ignorer la fin de la ligne dans un read)

Un composant précédé d'un entier n est répété n fois. Pour répéter plusieurs composants, il faut les mettre entre parenthèses : 3(5X,2I1) : 3 fois (5 espaces et 2 entiers d'un chiffre).

Exemples :

```
      write (*,10) nb,moy
10     format ('la moyenne des ',I2,'températures est de ',F5.1,' degrés')
      write (*,11) ((mat(i,j),i=1,4),j=1,4)
11     format ('la matrice est',4(/,'|', 4F5.2,'|')
```

Chaque read ou write correspond à une ligne. Si l'on donne une liste de plusieurs variables dans un read, il faudra absolument entrer toutes les valeurs (entre espaces ou virgules) avant la touche Entrée. Mais certains compilateurs (G77 par exemple) considèrent qu'un caractère '$' en fin de format demande de ne pas retourner à la ligne.

remarque : on peut également écrire ou lire dans une variable chaîne de caractères, il suffit de remplacer le numéro d'unité par le nom de la chaîne. De même, le format peut être une chaîne.

```
      character*40 fmt
      integer i,nbchif
      data i /123/ , nbchif /4/
      write(fmt,10) nbchif
10     format(' ''*'',I',I1,','''*''')
      write (*,*) 'pour vérification FMT vaut:',fmt
      write (*,fmt) i
```

ceci affichera :

```
      pour vérification FMT vaut:( '*',I4,'*')
      * 123*
```

12) Que n'ai-je pas dit ?

Plein de choses. En particulier, il existe un type et des fonctions intrinsèques spécialement conçus pour le traitement des nombres complexes : le type complex.
