L3STE module maths appliquées 2016-17

(texte de Ghislain Picard, LGGE)

1 Généralités

1.1 Utilisation de l'informatique dans un contexte scientifique.

Deux alternatives:

1. Utiliser des logiciels existants. Exemple: traitement de texte, tableur, logiciel de traitement d'image, logiciel dédié à un instrument, logiciel dédié à un champ disciplinaire.
2. Développer un logiciel/programme spécifique. Par exemple, un modéle climatique nécessite d'être codé par une équipe de scientifiques pendant plusieurs années.

Choix du langage: Traditionnellement, Fortran est le langage le plus utilisé car il existe de très nombreuses librairies scientifiques: Fortran 77 très souvent et maintenant Fortran 90 de plus en plus. C/C++ est aussi utilisé en mathématique et informatique. On utilise aussi de plus en plus des langages interprétés vectoriels comme Matlab, Scilab, Python, pour les "petits" calculs.

1.2 Les différents language de programmation.

La liste est longue: langage machine, BASIC, C, C++, Fortran 77, Fortran 90, Pascal/Delphi, Java, Lisp, Shell script, Perl, PHP, Scilab, Matlab, IDL, Python, CAML, PROLOG, SQL, LOGO, HTML, ... des milliers. Pourquoi des milliers ? Parce que chacun est mieux adaptés à un usage particulier + pour des raisons historiques.
On peut tenter de les classer selon divers critères:

- Niveau d'abstraction d'un langage de programmation (langage machine, langage généraliste, langage spécialisé).
- Langage compilé (C, Fortran), interprété (BASIC, Scilab), ou pseudo-compilé (Java, Python).
- Langage typé (C, Fortran) ou à typage automatique (Scilab).
- Langage vectoriel (Scilab, Matlab, Fortran 90) ou scalaire (C, Fortran 77).
- Langage procédural (C, BASIC), langage objet (C++, CAML), langage fonctionnel (LISP), autre (PROLOG, CLIPS).
- Libre (Scilab) ou propriétaire (Matlab).

Mais de plus en plus ces différences s'estompent. Ainsi par exemple Python est une langage pseudo-compilé, procédural, objet, fonctionnel et vectoriel.

Pourquoi Scilab pour ce cours ?

1. La syntaxe ressemble à celle de Matlab, Fortran ou IDL, qui sont des standards pour les applications scientifique en recherche et dans l'industrie.
2. Scilab est un langage vectoriel, de haut niveau. Il s'apprend plus rapidement que Fortran.
3. Scilab est un logiciel libre de grande qualité. Autres logiciels libres similaires à Scilab: Octave et R.

2 Structure du langage Scilab.

Biblio:
Le site principal de Scilab: http://www.scilab.org/
Introduction de J.Ph Chancelier: http://cermics.enpc.fr/~jpc/mopsi/scilab.pdf
Documentation en ligne (en Anglais): http://scilabsoft.inria.fr/product/man-eng/index.html
Il est recommandé de télécharger l'introduction de J.Ph Chancelier, d'imprimer les pages 1 à 35 et de les lire après le premier TP.

2.1 Eléments de base

On parle ici de variables et opérations mathématiques de base.

2.1.1 Variables

- Variable: Une variable est un espace en mémoire de l'ordinateur qui peut contenir une valeur numérique, des caractères ou pour les languages de haut niveau des objets plus complexes (ex: vecteur et matrice dans SCILAB). Une variable a un **nom** et une **valeur**.
- Assignation: donner une valeur à une variable.

```
--> x = 2
```

Cette commande **assigne** la valeur 2 à la variable nommée x.

```
--> compteur = 0
```

Une variable SCILAB peut être nommée avec des lettres et des chiffres (mais doit commencer par une lettre). Il faut toujours donner des noms explicites dans les programmes! Comme en math ou en physique, on a des habitudes: i, j, k, l, m s'utilisent plutôt pour des entiers; a, b, c, d pour des réels constants; x, y, z pour des réels variables. Ces habitudes ont l'avantage de permettre une lecture des programmes plus rapide.
Autre exemple:

```
--> dx = 1.5e-2
--> dx = 1.5d-2
```

1.5e-2 signifie 1.5×10^{-2}. Attention 10e2 signifie $10 \times 10^2 = 1000$ et non 10^2 comme on pourrait le croire.
Autre exemple:

```
--> a = ''Bonjour''
```

La variable nommé a contient alors une chaine de caractère (string en Anglais).

2.1.2 Opérations mathématiques de base

Mode "calculette"

```
--> 2 + 2
   ans = 4
```

Quelques opérateurs: +, -, *, /, ^ ou ** (puissance).

Quelques fonctions mathématiques:

sqrt()	la racine carrée
cos()	le cosinus
int()	la partie entière
round()	l'arrondi au plus proche
abs()	la valeur absolue

Quelques constantes: %pi, %e

En général une expression mathématique s'execute de gauche à droite. Mais il y a aussi des règles de priorité des opérateurs similaires à celles en math:

- La multiplication et la divison sont prioritaires sur l'addition et la soustraction.
- La puissance est prioritaire sur la multiplication et la divison.
- Les fonctions sont prioritaires sur tout.

Ces règles sont importantes car on peut toujours s'en sortir avec des parenthèses mais beaucoup de parenthèses réduisent énormément la lisibilité!
Quelques exemples:

```
--> 2*9/3
--> 21/7/3
--> 21/7*3
--> 21/(7*3)

--> 2+3*4
--> 2+3 *4
--> (2+3)*4

--> 5**3*4+1
--> 4*5**3+1
--> 5**3 * 4 + 1
--> ((5**3) * 4) + 1

--> cos(1.5*%pi)**2
```

Evitez les parenthèses, utilisez les espaces pour la lisibilité, faites des calculs intermédiaires!

2.1.3 Variable et opérations mathématiques

```
--> x = 5 + 2
--> y = x + 3
```

Dans le premier cas, une valeur est assignée à x. Dans le second cas, la valeur de x est utilisée. Quelles valeurs contiennent x et y après l'exécutation de ces 2 lignes ?

Exemple réel, calcul du nombre de Richardson en micro-météo:

```
--> vaporsaturation = 6.1078 * 100 * exp( 17.502*(temp-273.16)/(temp-32.18) )
--> qsatsurf = eps * psatsurf / (p0 - (1-eps) *  psatsurf)
--> rib = 9.81 * z1t / windspeed**2 * ( (tempair-tempsurf)/tempair + (q1-qsatsurf) / (q1 + eps/(1-eps) ) )
```

Un cas plus complexe mais très courant: l'**incrémentation** de variable.

```
--> x = 5
--> x = x + 1
```

En une seule ligne, la valeur de x est utilisée puis est. Quelle valeur contient x après ces deux lignes ?
Autre exemple:

```
--> x = 2
--> x = x * x
--> x = x * x
```

Cet exemple montre l'importance de l'éxécution **séquentielle** propre à l'informatique. Imaginez la signification de cette écriture en mathématique!

2.2 Structures de contrôle.

Instructions du langage qui permettent de faire des embranchements conditionnels et des itérations (=boucles). Ces instructions romptent l'exécution séquentielle et sont les piliers de la programmation.

2.2.1 Embranchements conditionnels

Ils permetent d'exécuter differentes branches en fonction du résultat d'un test logique.
Exemple:

```
--> x = 2
--> y = 5
--> if ( x < y ) then
       disp('x est plus petit que y');
    end
```

if, **then** et **end** sont des instructions de SCILAB. La fonction disp permet d'afficher à l'écran une chaine de caractère (abréviation de "display").
Exemple:

```
--> x = 2
--> y = 5
--> if ( x < y ) then
       x = y
    end
```

Quelles sont les valeurs de x et y après cette série d'instructions ? Même question avec l'initialisation suivante:

```
--> x = 7
--> y = 5
--> if ( x < y ) then
       x = y
    end
```

Exemple un peu plus complexe, le "else":

```
if (x < y) then
   disp('x est plus petit que y');
else
   disp('x est plus grand que y');
end
```

Exemple réel, comment traduire ce calcul mathématique ?

$$f_h = \begin{cases} \frac{1}{1+10Ri_B} & Ri_B \geq 0 \\ 1 - \frac{10Ri_B}{1+10C_{Hneutral}\sqrt{|Ri_B|16\frac{z1}{z_0}}} & Ri_B < 0 \end{cases} \quad (1)$$

```
if ( rib >= 0 ) then
   // stable condition
   fn=1/(1+10*Rib)
else
   // unstable condition
   fz=0.25*sqrt(roughness/z1t)
   fn=1 - 10*Rib / (1 + 10*chn*sqrt(abs(Rib))/fz )
end
```

Le symbole // permet des mettre des commentaires, c'est à dire des lignes ignorées par SCILAB, mais très utile pour la lisibilité du programme.

Encore plus complexe, le "elseif":

```
if (x < y) then
   disp('x est plus petit que y');
elseif ( x == y ) then
    disp('x est égale à y');
else
    disp('x est plus grand que y');
end
```

Des tests logiques plus sophistiqués avec les opérateurs logiques:
Le ET logique:

```
if ( x>5 & x<10 ) then
     disp('x est compris entre 5 et 10');
end
```

Il faut que les 2 conditions soient VRAIES pour que l'ensemble soit VRAI.
Le OU logique:

```
if ( x>5 | x<-5 ) then
     disp('x est plus grand que 5 ou inférieur à -5 ');
end
```

Il suffit qu'une seule des 2 conditions soient VRAIES pour que l'ensemble soit VRAI.

En résumé, les opérations de comparaison et opérateurs logiques utiles:

==	égal
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
&	et
|	ou

2.2.2 Itérations (ou boucles dans le langage courant)

Deux instructions: "for" pour boucler un nombre connu de fois, "while" pour boucler de façon conditionnelle.

```
for i=0:5
   disp (i)
end
```

Se lit "pour i de 0 à 5 exécute le bloc jusqu'à l'instruction end".
En général, on aura plutôt:

```
n=5;
for i=0:n
    disp (i)
end
```

La même chose avec le "while":

```
i=0;
while (i<=5)
   disp(i)
   i=i+1;
end
```

Se lit " tant que i<=5 exécute le bloc jusqu'à l'instruction end".
Le "while" est plus compliqué et on utilisera autant que possible le for. Mais le "while" s'avère nécessaire dans les cas où le nombre d'itérations n'est pas connu au moment d'entrer dans la boucle pour la première fois, mais dépend de ce qui se passe dans la boucle. L'exemple le plus classique est un calcul itératif dont on verifie la convergence par le residu entre deux itérations.
Remarques:

1. Pour les conditions (dans les if et les while) les parenthèses sont facultatives en Scilab mais obligatoire dans beaucoup d'autres langages, on veillera donc à les utiliser pour l'habitude.
2. Il faut impérativement **indenter** et commenter les codes pour la lisibilité.
3. Le ; en fin de ligne permet d'empecher Scilab d'afficher les résultats intermédiaires. En général, on en met partout sauf à la fin des lignes if, else, end, for et while.

2.3 Notion de vecteur

Dans SCILAB, une variable peut être un vecteur de contenu de nombreuses valeurs et même une matrice. C'est très pratique pour faire du calcul algébrique (matrice et vecteur) et du calcul sur des séries de données (comme dans Excel, mais en beaucoup plus pratique). Dans ce cours et dans les TP, on fera essentiellement du calcul sur des séries.
Plus concrètement:

```
--> x=[1,2,3,4,5]
```

x est maintenant un vecteur (ligne) et on peut faire des opérations mathématiques sur ce vecteur simplement:

```
--> 2*x
--> x+1
```

Si y est aussi un vecteur de même taille:

```
--> y=[7,8,9,10,11]
--> x + y
--> x+1
```

Toutefois, pour les multiplications il faut utiliser .* à la place de *

```
--> x .* y
```

2.3.1 Autres méthodes pour construire un vecteur

Un vecteur avec des valeurs de 1 à 5

```
-->x=1:5
```

Un vecteur avec des valeurs de 1 à 5 par pas de 0.5

```
-->x=1:5:0.5
```

Exemple, calcul de l'insolation maximale (sans atténuation atmosphérique) en fonction de l'heure

```
doy=1             // 1er janvier
lat=45.183 /180*%pi   // Grenoble
lon= 5.717 /180*%pi

solartime=0:24

declination= 0.39785*sin(4.868961+0.017203*doy + 0.033446*sin(6.224111 + 0.017202*doy));
h = 15*(solartime-12.0)/180*%pi;
elevation=asin(cos(lat)*cos(declination)*cos(h) + sin(lat)*sin(declination));

energy=1370.0 * sin (elevation)
```

On peut aussi construire des vecteurs vides (pour les remplir plus tard), des vecteurs de zéros, de uns et par extension de n'importe quelle valeur:

```
-->x=[]
-->x=zeros(5,1)
-->x=ones(5,1)
-->x=10*ones(5,1)
```

Pour information, on peut aussi créer des matrices, et des "tableaux" à 2 dimensions ou plus

```
-->x=ones(3,3)

 x  =

!   1.    1.    1. !
```

```
!   1.    1.    1. !
!   1.    1.    1. !


-->x=eye(3,3)

 x  =

!   1.    0.    0. !
!   0.    1.    0. !
!   0.    0.    1. !
```

2.3.2 Accès aux éléments d'un vecteur

Souvent, on veut faire des traitements complexes sur les valeurs des vecteurs qui ne sont pas prévues dans Scilab. Pour cela, on a besoin d'accéder aux élements des vecteurs. On utilise la meme notation qu'en mathématique.

```
--> x=[1,4,8,16,32]
-->x(1)
-->x(2)
```

"x(1)" s'utilise comme n'importe quelle variable et peut en particulier être affacté:

```
--> x=[1,4,8,16,32]
x(1)=0
```

En général, on utilise les vecteurs en combinaison des boucles. Un exemple simple: imaginons que la somme de deux vecteurs n'existe pas en SCILAB (i.e. x+y):

```
x=[1,2,3,4,5]
y=[7,8,9,10,11]
z=[]

for i=1:5
   z(i)=x(i)+y(i);
end

disp(z)
```

Et de facon plus générale

```
x=[1,2,3,4,5]
y=[7,8,9,10,11]
z=[]

for i=1:length(x)
   z(i)=x(i)+y(i);
end

disp(z)
```

Autres exemples: voir exercice (à la fin) et les prochains TP.

2.4 Notion de fonctions (et procédures)

Quand un programme devient un peu gros (> de 100 lignes), on apprécie de décomposer les taches en morceau. Il y a 2 phases pour les fonctions: la déclaration et l'utilisation.

- La déclaration:

```
function y=sommeentier(n);
    y=0
    for i=1:n
y=y+i
    end
endfunction
```

- Et on l'appelle ainsi:

```
-->sommenentier(4)
```

Avec plusieurs résultats et plusieurs arguments:

- Déclaration:

```
function [u,v]=sommedifference(x,y); u=x+y; v=x-y; endfunction
```

- Appel:

```
[a,b]=sommedifference(1,2);
```

Important: A l'intérieur d'une fonction on peut lire la valeur d'une variable défini "au-dessus", mais on ne peut pas la modifier. On dit que la fonction a son propre contexte d'exécution:

```
-->a=4
   function y=plusa(x);
y=a+x; // a est accessible dans la fonction
                  (a appartient au contexte appelant la fonction)
   endfunction
   plusa(5)

ans=
9.


-->a=4;
   function y=changea(x);
a=x;
y='dans la fonction, a vaut '+string(a);
   endfunction

   changea(5)
   disp(a)


 ans=
 dans la fonction, a vaut 5
    4.
```

Donc, changea, ne change pas a !
Comment ca marche ? Voir figure 1.
Pourquoi fait on quelque chose de si compliqué ?
Parce que c'est très pratique ! Imaginez si toutes les variables étaient accessibles partout dans un programme qui contient des milliers de ligne de code (longueur typique d'un modèle) écrit par plusieurs personnes. Il faut donc voir une fonction comme une boite étanche: seuls entrent les arguments de la fonction, seuls sortent les résultats.
Les procédures ou sous-routines sont des fonctions qui ne renvoient pas de valeur. Les procédures n'existent pas en Scilab. En C et dans beaucoup d'autres langages, les fonctions et procédures sont très semblables, elles se déclarent et s'appellent avec la même syntaxe. En Fortran par contre, la distinction est nette entre les fonctions et procédures. Exemple en Fortran:

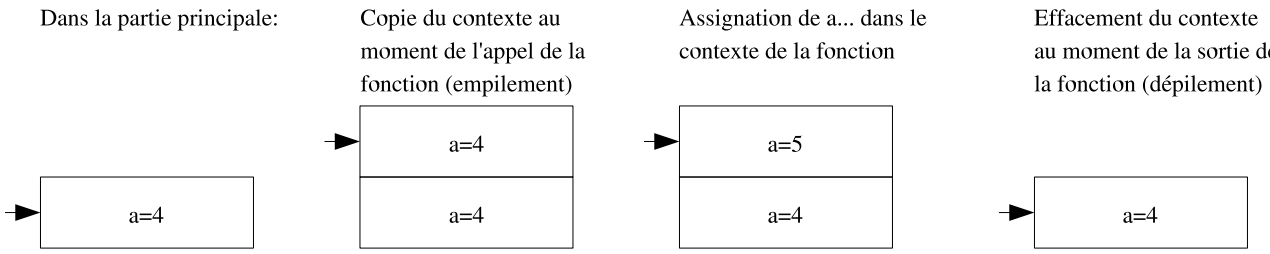

Figure 1: Empilement et dépilement des contextes de travail au moment de l'exécution d'une fonction. Le contexte contient toutes les variables.

- Déclaration:

```
subroutine affiche_resultat (X)
integer :: x
print *,'le resultat est ',x
end


function somme(a,b)
integer :: a,b
somme=a+b
end
```

- Appel:

```
call affiche_resultat(3)

c=somme(a,b)
```

3 Exercices

Exercice I: Compléter pour calculer la somme des entiers de 1 à 5

```
-->
   somme=0;
   for i=1:5

   end
   disp(somme);
```

Exercice II: Ecrire un programme qui calcule le factoriel de 10.

Exercice III: Modifier le programme (I) pour calculer la somme des entiers de 1 à *n*, avec *n* défini en début de programme, en utilisant une boucle `while`.

Exercice IV: Écrire un programme qui renvoie la valeur maximale parmi un vecteur de valeurs défini en début de programme.

Exercice V: Écrire un programme qui renvoie le maximum d'une fonction sur un intervalle donné.
Exemple avec la fonction $y = -3/2 \times x^2 + x + 5$, sur l'intervalle [0:5].
Obtenir le résultat analytiquement, et le comparer avec le résultat du programme.