Université Abdelmalek Essaâdi
Ecole Nationale des Sciences Appliquées – Tétouan
Département TITM
Filière: 2AP-1
Module : Informatique 2

Initiation à Matlab

Polycopié du cours

Khamlichi Abdellatif

2014-2015

Table des matières

Introduction

Ce cours est une initiation au logiciel *Matlab* et à la programmation dans cet environnement.
On y exposera les bases de *Matlab* et de la programmation d'algorithmes mathématiques élémentaires à l'aide de ce langage.
Après la présentation de quelques éléments de base de *Matlab*, on introduira les principales opérations usuelles sur les scalaires, les vecteurs et les matrices. On verra ensuite comment utiliser des fichiers *script* (M-file) et *function* avec *Matlab* avant de présenter certaines opérations graphiques offertes par ce logiciel.
Dans la section 5 on apprendra la syntaxe des tests et des différentes boucles de programmation en *Matlab*. Finalement, dans la section 6 on verra quelques fonctions plus avancées existant dans *Matlab*.
Il faut rappeler ici qu'un langage scientifique comme *Matlab* exige rigueur et maîtrise parfaite. Sa simplicité apparente ne doit pas cacher l'effort qu'il faut fournir pour apprendre correctement la syntaxe qu'il utilise et savoir ensuite comment traduire dans ce langage les algorithmes mathématiques. Un langage de programmation ne peut se substituer au travail amont qui consiste à traduire sous forme d'algorithme et ensuite d'organigramme ce que l'on veut programmer.

Présentation de Matlab

Matlab est un langage de programmation, mais il est beaucoup plus que ça. Il s'agit en fait de ce qu'on appelle une console d'exécution (*shell*) qui partage certaines des caractéristiques des consoles DOS ou UNIX.

Comme toutes les consoles, *Matlab* permet d'exécuter des fonctions, d'attribuer des valeurs à des variables, d'effectuer des opérations mathématiques, de manipuler des matrices, de tracer facilement des graphiques, etc.

La figure 1 présente l'écran *Matlab* de base: la fenêtre de commandes.

Le symbole >> s'appelle prompt ou bien invite de Matlab. Il invite l'utilisateur à taper une commande.

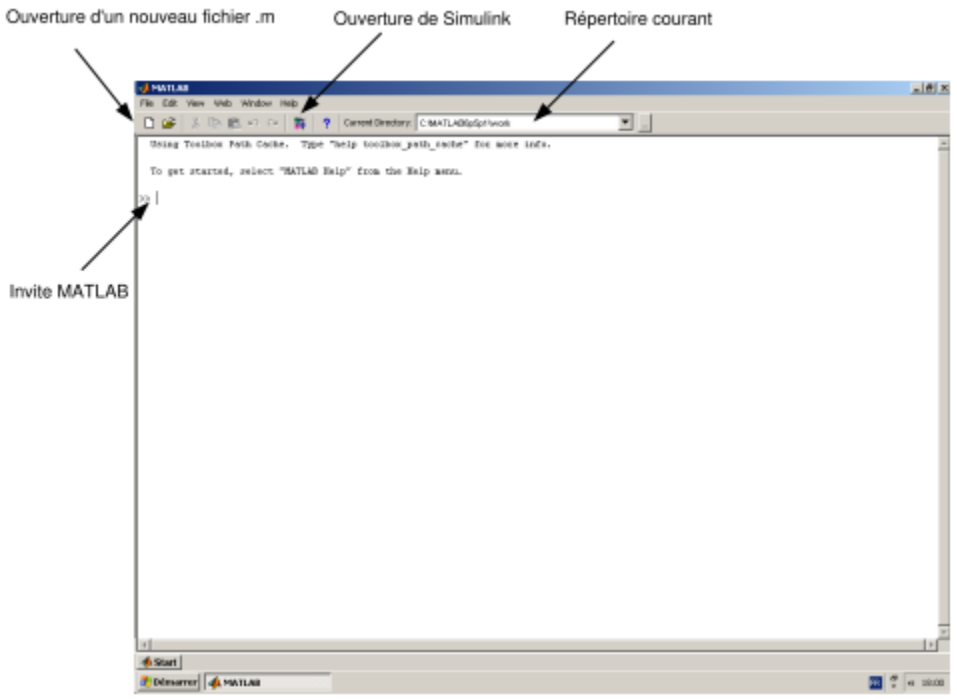

Figure 1 : Fenêtre de commandes de *Matlab*

Il est utile de noter que le langage *Matlab* n'est pas un langage compilé (contrairement au langage C++, par exemple). Le logiciel lit et exécute les programmes instruction par instruction et ligne par ligne.

Lorsque *Matlab* détecte une erreur, le logiciel s'arrête et un message d'erreur ainsi que la ligne où l'erreur est détectée s'affichent à l'écran. Apprendre à lire les messages d'erreur est donc important pour "déboguer" les programmes rapidement et efficacement.

Dans la fenêtre de commande, l'utilisateur peut affecter des valeurs à des variables et effectuer des opérations sur celles-ci. Par exemple :

```
>> x=4
x =
4
>> y=2
y =
2
>> x+y
ans =
6
>> x*y
ans =
8
>>
```

Ici, il faut noter que lorsque l'utilisateur ne fixe pas de variable de sortie, *Matlab* place par défaut le résultat d'une opération dans la variable *ans*. Il est toujours possible de connaître les variables utilisées et leur type à l'aide de la commande *who* ou bien *whos*. Par exemple, pour les manipulations précédentes:

```
>> whos
Name        Size        Bytes       Class
ans         1x1 8       double      array
x           1x1 8       double      array
y           1x1 8       double      array
Grand total is 3 elements using 24 bytes
>>
```

La solution de x+y a donc été perdue. Il est donc préférable de toujours donner des noms aux variables de sortie :

```
>> x=4;
>> y=2;
>> a=x+y
a =
6
```

```
>> b=x*y
b=
8
>> whos
Name        Size        Bytes       Class
a           1x1 8       double      array
b           1x1 8       double      array
x           1x1 8       double      array
y           1x1 8       double      array
Grand total is 4 elements using 32 bytes
>>
```

Notons au passage que le point-virgule permet de ne pas afficher la valeur à l'écran, ce qui permettra éventuellement des programmes plus rapides.

Le signe de pourcentage (%) permet de mettre ce qui suit sur une ligne en commentaire (*Matlab* n'en tiendra pas compte à l'exécution).

La fonction *clear* permet d'effacer des variables. Par exemple :

```
>> clear x  % on efface x de la mémoire
>> whos
Name        Size        Bytes       Class
a           1x1 8       double      array
b           1x1 8       double      array
y           1x1 8       double      array
Grand total is 3 elements using 24 bytes
>>
```

La sortie de la fonction *whos* donne, entre autre, la classe de la variable. Plusieurs classes de variables sont disponibles à l'utilisateur de *Matlab*. Les classes les plus utiles pour l'utilisateur débutant sont l'entier, le réel simple, le réel double et les variables caractère 'char'. Pour les variables char, la déclaration se fait entre apostrophes:

```
>> mot1 = 'bonjour'
mot1 = bonjour
```

Il est possible de concaténer des mots à l'aide des parenthèses carrées (crochets) (la fonction *strcat* de *Matlab* permet d'effectuer sensiblement la même tâche) :

>> mot1 = ’bonjour’;

>> mot2 = ’tout le monde’;

>> mot1_2 = [mot1 ’ ’ mot2] % l’emploi de ’ ’ permet d’introduire un espace

mot1_2 = bonjour tout le monde.

Chapitre 1: Opérations mathématiques de base avec *Matlab:* scalaires, vecteurs et matrices

Plusieurs types de données sont disponibles dans *Matlab*. Les types traditionnels que l'on retrouve dans tous les langages de programmation: les types numériques (single, double, int8, etc...), caractères char, les tableaux de réels, et les tableaux creux sparse, et les types composées cell, structure ainsi que les types définis par l'utilisateur, comme les fonctions *inline*. Le type de donnée privilégiée sous *Matlab* est les tableaux à une ou deux dimensions, qui correspondent aux vecteurs et matrices utilisés en mathématiques et qui sont aussi utilisés pour la représentation graphique. Nous allons donc nous attarder sur leur définition et leur maniement dans les paragraphes qui suivent.

L'élément de base de *Matlab* est la matrice. C'est-à-dire qu'un scalaire est une matrice de dimension 1x1, un vecteur colonne de dimension n est une matrice nx1, un vecteur ligne de dimension n, une matrice 1xn. Contrairement aux langages de programmation usuels (i.e. C++), il n'est pas obligatoire de déclarer les variables avant de les utiliser et, de ce fait, il faut prendre toutes les précautions dans la manipulation de ces objets.

Les scalaires se déclarent directement, par exemple :

```
>>  x = 0;
>>  a = x;
```

1.1 Vecteurs

Les vecteurs ligne se déclarent de la manière suivante :

```
>>    V_ligne = [0  1  2]
V_ligne =
0        1        2
```

Pour les vecteurs colonne, on sépare les éléments par des points-virgules :

```
>>     V_colonne = [0; 1; 2]
V_colonne  =
0
1
2
```

Il est possible de transposer un vecteur à l'aide de la fonction *transpose* ou avec le point apostrophe (.'). Ainsi,

```
>>    V_colonne=transpose(V_ligne)
V_colonne  =
0
1
2
>>    V_colonne=V_ligne.'
V_colonne  =
0
1
2
```

Le double point (:) est l'opérateur d'incrémentation dans *Matlab*. Ainsi, pour créer un vecteur ligne des valeurs de 0 à 1 par incrément de 0.2, il suffit d'utiliser:

```
>>  V= [0:0.2:1]
V  =
Columns  1  through  6
0       0.2000       0.4000       0.6000       0.8000       1.0000
```

Par défaut, l'incrément est de 1. Ainsi, pour créer un vecteur ligne des valeurs de 0 à 5 par incrément de 1, il suffit d'utiliser :

```
>>  V=[0:5]
V  =
0       1       2       3       4       5
```

On peut accéder à un élément d'un vecteur et même modifier celui-ci directement (Notez que contrairement au C++, **il n'y a pas d'indice 0** dans les vecteurs et matrices en *Matlab*) :

```
>>  a=V(2);
>>  V(3)=3*a
V  =
0       1       3       3       4       5
```

Les opérations usuelles d'addition, de soustraction et de multiplication par scalaire sur les vecteurs sont définies dans MATLAB :

```
>>  V1=[1  2];
>>  V2=[3  4];
```

```
>> V=V1+V2 % addition de vecteurs
V =
4       6
>> V=V2-V1 % soustraction de vecteurs
V =
2       2
>> V=2*V1   % multiplication par un scalaire
V =
2       4
```

Dans le cas de la multiplication et de la division, il faut faire attention aux dimensions des vecteurs en cause.

Pour la multiplication et la division élément par élément, on ajoute un point devant l'opérateur (.* et ./). Par exemple :

```
>> V=V1.*V2   % multiplication élément par élément
V =
3       8
>> V=V1./V2   % division élément par élément
V =
0.3333      0.5000
```

Cependant, *Matlab* lance une erreur lorsque les dimensions ne concordent pas. Les messages d'erreur sont utiles pour corriger les programmes (parenthèse oublié par exemple). Il faut cependant procéder à la vérification systématique de l'instruction ou du programme avant de lancer l'exécution (reflexe de base d'un programmeur):

```
>> V3=[1  2  3]
V3 =
1       2       3
>> V=V1.*V3
??? Error using ==> .* Matrix dimensions must agree.
```

La multiplication de deux vecteurs est donnée par (*). Ici, l'ordre a de l'importance :

```
>> V1=[1  2]; % vecteur 1x2
>> V2=V1';    % vecteur 2x1
```

```
>>  V=V1*V2
V  =
5
>>  V=V2*V1
V  =
1        2
2        4
```

Il est aussi possible de concaténer des vecteurs. Par exemple :

```
>>  V1=[1  2];
>>  V2=[3  4];
>>  V=[V1  V2]
V  =
1        2        3        4
```

De même, pour les vecteurs colonnes :

```
>>  V1=[1;2];
>>  V2=[3;4];
>>  V=[V1;V2]
V  =
1
2
3
4
```

1.2 Matrices

On peut aussi créer des matrices à partir de vecteurs, par exemple,

```
>>  V1=[1  2];
>>  V2=[3  4];
>>  V=[V1; V2]
V  =
1        2
3        4
```

qui n'est pas équivalent à :

```
>>  V1=[1; 2];
>>  V2=[3; 4];
```

```
>> V=[V1  V2]
V =
1       3
2       4
```

Il faut donc être très prudent dans la manipulation des vecteurs. Par exemple, une mauvaise concaténation :

```
>> V1=[1  2];
>> V2=[3; 4];
>> V=[V1; V2]
```

??? Error using ==> vertcat All rows in the bracketed expression must have the same number of columns.

Les matrices peuvent aussi être construites directement :

```
>> M=[1  2;  3  4]
M =
1       2
3       4
```

On peut évidemment avoir accès aux éléments de la matrice par :

```
>> m21=M(2,1)   %  2e  ligne,  1ere  colonne
m21 =
3
```

On peut aussi ”compter” les éléments. *Matlab* compte alors tous les éléments d'une colonne (de haut en bas) avant d'accéder à la colonne suivante. Ainsi, dans la matrice 3x3 suivante:

```
>> A=[1  2  3;  8  5  6;7  8  9]
A =
1       2       3
8       5       6
7       8       9
```

les valeurs des éléments ai,j sont données par leur rang affecté par *Matlab*. Le 4e élément est 2 :

```
>> a4=A(4)
a4 =
2
```

Il est aussi possible de stocker dans un vecteur une ou plusieurs lignes (ou colonnes). Ainsi, si l'on veut stocker la deuxième colonne de la matrice A :

```
>> V=A(:,2)  % ici, (:) signifie toutes les lignes
V =
2
5
8
```

De la même manière, si l'on veut stocker les lignes 2 et 3 :

```
>> M2=A(2:3,:)  % (2:3) signifie ligne 2 à 3
% et (:) signifie toutes les colonnes
M2 =
8     5     6
7     8     9
```

Il est possible d'inverser inv(), de transposer transpose() ou avec l'apostrophe (.') les matrices :

```
>> invM=inv(M)
invM =
-2.0000    1.0000
1.5000   -0.5000
>> transpM=M.'
transpM =
1     3
2     4
```

Un des intérêts de *Matlab* est la possibilité d'utiliser directement les opérations mathématiques prédéfinies pour les matrices. L'addition et la soustraction sont directes (attention aux dimensions) ainsi que la multiplication par un scalaire :

```
>> A=[1  2; 3  4];
>> B=[4  3; 2  1];
>> C=A+B  % addition
C=
5     5
5     5
>> D=A-B  % soustraction
D=
-3    -1
1     3
```

```
>> C=3*A % multiplication par un scalaire
C=
3     6
9    12
```

Pour la multiplication et la division, les opérateurs usuels (* et /) sont définis pour la multiplication et division matricielles :

```
>> C=A*B  % multiplication de matrices
C=
8     5
20    13
>> D=A/B  % division de matrices
D=
1.5000   -2.5000
2.5000   -3.5000
```

Afin de réaliser la multiplication et la division élément par élément, on précède les opérateurs par un point (.* et ./) :

```
>> C=A.*B  % multiplication élément par élément
C=
4     6
6     4
>> D=A./B  % division élément par élément
D=
0.2500    0.6667
1.5000    4.0000
```

D'autres opérations sur les matrices seront présentées dans les sections subséquentes.

Il faut noter certaines matrices spéciales qui peuvent être utilisées, par exemple la matrice identité :

```
>> I=eye(3) % matrice identité
I=
1     0     0
0     1     0
0     0     1
```

On peut aussi déclarer des vecteurs (et des matrices) ne contenant que des zéros ou des 1.

```
>> V_nul=zeros(1,2) % un vecteur de 1 ligne, 2 colonnes de 0
V_nul=
0       0
>> V_un=ones(1,2) % un vecteur de 1 ligne, 2 colonnes de 1
V_un=
1       1
>> M_un=ones(2,2) % une matrice 2x2 de 1
M_un=
1       1
1       1
```

Dans certaines applications, il est parfois utile de connaître les dimensions d'une matrice, et la longueur d'un vecteur (retournés, par exemple, par une fonction).

Dans ce cas, on utilise les fonctions *length* et *size*.

```
>> V=[0:0.1:10]; % utilisation de length - vecteur 1x101
>> n=length(V)
n=
101
>> M=[1 2 3; 4 5 6]; % utilisation de size - matrice 2x3
>> [n,m]=size(M)
n=
2
m=
3
>> dim=length(M)  % utilisation de length sur une matrice
dim=
3
```

Dans ce cas *length* donne la plus grande dimension, ici le nombre de colonnes.

1.3 Extraction d'une sous-matrice

On peut utiliser les deux points pour extraire une sous-matrice d'une matrice A.

A(:,j) extrait la jème colonne de A. On considère successivement toutes les lignes de A et on choisit le jème élément de chaque ligne.

A(i,:) extrait la ième ligne de A.

A(:) reforme le matrice A en un seul vecteur colonne en concaténant toutes les colonnes de A.

A(j:k) extrait les éléments j à k de A et les stocke dans un vecteur ligne

A(:,j:k) extrait la sous-matrice de A formée des colonnes j à k.

A(j:k,:) extrait la sous-matrice de A formée des lignes j à k.

A(j:k,q:r) extrait la sous-matrice de A formée des éléments situés dans les lignes j à k et dans les colonnes q à r.

Ces définitions peuvent s'étendre à des pas d'incrémentation des lignes et des colonnes différents de 1.

Chapitre 2: Fichiers *script* et *function*

Jusqu'à présent, l'utilisation que nous avons faite de *Matlab* s'apparente beaucoup à celle d'une calculatrice. Pour des tâches répétitives, il s'avère beaucoup plus pratique et judicieux d'écrire des programmes pour effectuer les calculs désirés.

Il existe deux types de fichiers qui peuvent être programmés avec *Matlab*: les fichiers *script* (M-file) et *function*. Dans les deux cas, il faut lancer l'éditeur de fichier et sauvegarder le fichier avec l'extension **.m**.

2.1 Fichiers *script*

Comme tout langage, *Matlab* possède aussi un certain nombre d'instructions syntaxiques (boucles simples, conditionnelles, etc...) et de commandes élémentaires (lecture, écriture, etc...). Ces instructions syntaxiques seront vues dans la deuxième partie du cours.

Dès que le calcul à effectuer implique un enchaînement de commandes un peu compliqué, il vaut mieux écrire ces dernières dans un fichier. Par convention un fichier contenant des commandes Matlab porte un nom avec le suffixe **.m** et s'appelle pour cette raison un ***M-file*** ou encore *script*. On utilisera **toujours l'éditeur intégré au logiciel** qui se lance à partir de la fenêtre de commande en cliquant sur les icônes new M-file ou open file dans la barre de menu.

Une fois le fichier enregistré sous un nom valide, on peut exécuter les commandes qu'il contient en tapant son nom - sans le suffixe .m - dans la fenêtre de commande. Si vous avez ouvert l'éditeur comme indiqué, à partir de la fenêtre de commande, les M-file seront créés dans le répertoire courant, accessible depuis cette fenêtre, et vous n'aurez pas de problème d'accès. Si vous voulez exécuter des scripts qui se trouvent ailleurs dans l'arborescence des fichiers, vous aurez éventuellement à modifier le *Path* en cliquant sur le menu **file − >SetPath** ou bien en changeant de répertoire de travail (cliquer sur l'onglet **current directory**).

Le fichier *script* permet de lancer les mêmes opérations que celles écrites directement à la fenêtre de commandes de *Matlab* après le symbole prompt (>>). Toutes les variables utilisées dans un *script* sont disponibles à l'invite *Matlab* (fenêtres de commandes) une fois le script exécuté.

Un script *Matlab* est composé d'une suite d'instructions, toutes séparées par une virgule (ou de manière équivalente, un passage à la ligne) ou un point virgule. La différence entre ces

deux types de séparation est liée à l'affichage ou non du résultat à l'écran (seulement effectué dans le premier cas).

Par exemple, créons à l'aide de l'éditeur intégré de *Matlab* dans le répertoire de travail choisi, déjà déclaré par SetPath, le fichier test.m. Supposons qu'il contient les instructions suivantes :

```
clear all
x=4;
y=2;
a=x +y
b=x*y
```

Ecrivons dans la fenêtre de commandes le nom du fichier

```
>>  test
a  =
6
b  =
8
```

En tapant whos ensuite, on produit la sortie suivante:

```
Name        Size        Bytes       Class
a           1x1 8       double      array
b           1x1 8       double      array
x           1x1 8       double      array
y           1x1 8       double      array
Grand total is 4 elements using 32 bytes
>>
```

Habituellement, on utilise les fichiers *script* afin de :

- Déclarer des variables ;
- Effectuer des opérations mathématiques ;
- Appeler des fonctions ;
- Tracer des figures ;
- Programmer des algorithmes.

2.2 Fichiers *function* (M-file function)

L'idée de base d'une fonction est d'effectuer des opérations sur une ou plusieurs entrées ou arguments pour obtenir un résultat qui sera appelé sortie. Il est important de noter que l'appel

de la fonction se fait en précisant ses variables entrées si ces dernières ne sont pas disponibles à l'invite *Matlab*.

Par exemple, la fonction suivante admet une seule sortie a qui constitue le résultat de l'addition des deux arguments d'entrée x et y:

```
function  a = ma_fonction(x,y)
a=x+y;
```

Lorsqu'on tape dans la fenêtre de commandes:

```
>>  a  =  ma_fonction(4,2)
```

on obtient

```
a  = 6
```

Ensuite, on vérifie que

```
>>  whos
Name          Size          Bytes          Class
a             1x1 8         double         array
Grand  total  is  1  element  using  8  bytes
>>
```

Modifions la fonction pour lui demander de calculer aussi le produit de x et y de la façon suivate :

```
function  [a,b] = ma_fonction2(x,y)
a=x+y;
b=x*y;
```

Dans ce cas, on vérifie que:

```
>>  [a,b]  =  ma_fonction2(4,2)
a  =
6
b  =
8
>>  whos
Name          Size          Bytes          Class
a             1x1 8         double         array
b             1x1 8         double         array
Grand  total  is  2  elements  using  16  bytes
>>
```

On peut éviter l'affichage des sorties en utilisant le point-virgule :

```
>> [a,b]=ma_fonction2(4,2);

>> whos
Name        Size        Bytes       Class
a           1x1 8       double      array
b           1x1 8       double      array
Grand total is 2 elements using 16 bytes
>>
```

Remarquons que nous pouvons appeler la fonction ma_fonction2 pour calculer uniquement la somme de x et y. On obtient alors

```
>> a = ma_fonction2(4,2)
a =
6
>> whos
Name        Size        Bytes       Class
a           1x1 8       double      array
Grand total is 1 element using 8 bytes
```

Remarques importantes :

* Le passage des arguments d'entrée dans les fonctions se fait par valeur. Aussi, même si elles sont modifiées dans la fonction les valeurs des paramètres ne sont pas modifiées dans le programme appelant.

* Si une des variables de la procédure n'est pas définie à l'intérieur de celle-ci elle doit obligatoirement être fournie en argument d'entrée.

* La récupération des valeurs calculées par la fonction se fait par les paramètres de sortie.

* Le nom du fichier contenant la fonction porte obligatoirement le nom de cette dernière. On peut mettre plusieurs fonctions dans le même M-file mais seule la fonction du même nom que le fichier peut être utilisée, appelée, à partir de la fenêtre de commandes ou d'une autre fonction ou d'un script. Les autres fonctions éventuellement stockées dans le fichier peuvent s'appeler entre elles mais ne sont pas visibles de l'extérieur.

Habituellement, on utilise les fichiers *function* afin de :

– Programmer des opérations répétitives ;

– Limiter le nombre de variables dans l'invite *Matlab ;*

– Diviser le programme (problème) de manière claire.

2.3 Définition d'une fonction par la commande « *inline* »

Une fonction ne comportant qu'un petit nombre d'instructions peut être définie directement dans la fenêtre de commandes de la manière suivante :

```
>>angle=inline('atan(y/x)')
angle =
Inline function:
angle(x,y) = atan(y/x)
>>angle(5,4)
ans =
0.6747
```

Les arguments de la fonction angle sont normalement fournis à l'appel dans l'ordre d'apparition dans la définition de la fonction. On peut aussi spécifier les arguments d'appel explicitement

```
>>f =inline('sin(alpha*(x+y))','x','y','alpha')
f =
Inline function:
f(x,y,alpha) =sin(alpha*(x+y))
>>f(0.2,0.3,pi)
ans =
1
```

2.4 Fonctions outils

Enfin, notez que certaines commandes spéciales ne peuvent s'utiliser qu'en relation à une fonction: nargin, donne le nombre d'arguments d'entrée passés à l'appel de la fonction.

```
function c=testarg1(a,b)
if (nargin = = 1)
c=2*a;
elseif (nargin = = 2)
c=a+b;
end
```

nargin peut aussi être utilisée pour connaître le nombre prévu d'arguments d'entrée

```
>> nargin('testarg1')
```

```
ans =
2
```

La commande nargout fonctionne de manière analogue pour les arguments de sortie.

Chapitre 3: Fonctions et représentation graphique sous *Matlab*

3.1 Graphiques simples

Cette section vise une initiation aux nombreuses facultés graphiques offertes par *Matlab*.

Dans toutes les représentations graphiques, le logiciel se base sur des données discrètes rangées dans des matrices ou des vecteurs colonnes. Par exemple, pour représenter des courbes du type y = f(x) ou des surfaces z = f(x, y), les données x,y,z doivent être des vecteurs colonnes (x et y) ou des matrices (z) aux dimensions compatibles. L'instruction de dessin correspondante (par exemple plot(x,y) pour tracer des courbes planes) est alors utilisée et éventuellement complétée par des arguments optionnels (couleur, type de trait, échelle sur les axes, etc...). La visualisation du résultat s'effectue dans une fenêtre graphique (avec possibilité de zoom, de rotation, d'impression).

La fonction plot permet de tracer des courbes en *Matlab*. Les arguments de cette fonction sont les vecteurs des variables indépendantes et dépendantes (en alternance), comme dans l'exemple qui suit :

```
>> x=[0:0.01:2*pi];
>> plot(x,cos(x),x,sin(x))
```

On aurait aussi pu simplifier par :

```
>> plot(x',[cos(x)' sin(x)'])
```

Ces graphiques manquent cependant de clarté. Il faut toujours nommer les axes (et mettre les unités si possible), proposer une légende, etc. Ceci est évidemment un peu long à écrire à l'invite *Matlab*. Un script est tout indiqué :

```
% graphique.m
clear all
close all % ferme les anciennes figures
x=[0:0.01:2*pi]; y1=cos(x); y2=sin(x);
figure(1)
plot(x,y1,'.',x,y2,'+') % cos(x) en points, sin(x) en +
title('sinus et cosinus') xlabel('x') ylabel('f(x)')
legend('cos(x)','sin(x)',0) % le 0 place la légende à côté des courbes
```

Pour en savoir plus, particulièrement sur les couleurs et types de courbes, tapez help *plot* à l'invite *Matlab*.

3.2 Fonctions mathématiques simples

Les opérateurs algébriques (+, -, *, /, .*,./) ont été définis précédemment pour les scalaires, vecteurs et matrices. On montrera ici (sans être exhaustif), les principales fonctions mathématiques fournies dans *Matlab* et leur utilisation. Pour les fonctions non présentées, l'utilisateur peut toujours utiliser l'aide des fonctions avec la fonction *help* qui prend pour argument le nom de la fonction. Par exemple, la fonction cosinus :

```
>> help cos
COS    Cosine of argument in radians.
   COS(X) is the cosine of the elements of X.
```

Dans la suite, on présente les fonctions mathématiques usuelles et leur appel dans *Matlab*. Ensuite, on présente les principales fonctions spécifiques aux matrices.

3.2.1 Fonctions mathématiques usuelles

Toutes les fonctions mathématiques de base sont déjà programmées dans *Matlab*.

Toutes les fonctions courantes et moins courantes existent. La plupart d'entre elles fonctionnent **en complexe**. On retiendra que pour appliquer une fonction à une valeur, il faut mettre cette dernière entre parenthèses. Exemple :

```
>> sin(pi/12)
ans =
0.16589613269342
```

Voici une liste non exhaustive :

– fonctions trigonométriques et inverses : sin, cos, tan, asin, acos, atan

– fonctions hyperboliques (on rajoute «h») : sinh, cosh, tanh, asinh, acosh, atanh

– racine, logarithmes et exponentielles : sqrt, **log**, log10, exp

– fonctions erreur : erf, erfc

– fonctions de Bessel et Hankel: besselj, bessely, besseli, besselk, besselh et hankel. Il faut deux paramètres : l'ordre de la fonction et l'argument lui-même. Ainsi J1(3) s'écrira besselj(1,3)

La notion de fonction est plus générale dans *Matlab*, et certaines fonctions peuvent avoir plusieurs entrées (comme besselj par exemple) mais aussi plusieurs sorties.

3.2.2 Fonctions matricielles

Toutes les fonctions matricielles de base sont déjà programmées dans *Matlab*.

Voici quelques exemples :

Size, length, diag, det, norm, rank, trace, sum, prod, mean, std, var, max, min, rand, null, inv, pinv, sort, reshape, fliplr, flipud, tril, triu,…

3.2.3 Fonctions avancées

Ce sont des fonctions qui interviennent en analyse numérique telles que: lu, chol, qr, cond, eig, fzero,…

3.3 Exemple de représentation graphique

En exécutant le script suivant :

```
x=linspace(0,pi,30);    % crée un tableau de 30 composantes uniformément
                        % réparties entre 0 et pi
y=sin(x);
plot(x,y)       %relie les points (xi,yi) par un trait continu noir
plot(x,y,'p-b')  %relie les points (xi,yi) par un trait continu de couleur et
                 %matérialise les points avec un symbole
plot(x,y,'pb')  %matérialise les points (xi,yi) avec un symbole de couleur
```

Les points peuvent être matérialisés par le symbole p prenant les valeurs suivants : o . ∗ + ×

Les couleurs sont repérées par leur initiale en anglais : r(ed), b(lue), blac(k), w(hite), y(ellow), m(agenta), g(reen)

On peut rajouter un titre à la figure avec la commande *title*

```
title('sin(x) sur l''intervalle [0,pi]')
```

Remarquer l'emploi d'une double apostrophe pour en faire figurer une dans une chaîne de caractères délimitée justement par deux apostrophes.

On peut représenter plusieurs courbes sur la même figure de plusieurs manières: d'abord par un seul appel à la fonction plot

```
plot(x,cos(x),x,sin(x),x,exp(-x))       % Matlab va automatiquement utiliser des couleurs
                                        % différentes pour chaque courbe
plot(x,cos(x),'o-r',x,sin(x),'x-b',x,exp(-x),'*-g')      % pour spécifier le type
                                % de symbole et la couleur à utiliser pour chaque courbe
legend('cos(x)','sin(x)','exp(-x)')  % pour rajouter une légende
```

Par défaut la fenêtre graphique est effacée avant chaque commande *plot*. Pour superposer des courbes par des appels successifs à cette fonction, il faut auparavant avoir utilisé la commande *hold on*.

3.4 Echelles logarithmiques

On peut tracer des échelles log en abscisse, en ordonnée ou bien les deux. Les fonctions correspondantes s'appellent respectivement *semilogx*, *semilogy* et *loglog*. Elles s'utilisent exactement de la même manière que *plot*.

Par exemple :

```
>> x=1:100;
>> semilogx(x,log(x))
```

3.5 Afficher plusieurs graphiques (subplot)

Voilà une fonctionnalité très utile pour présenter sur une même page graphique un grand nombre de résultats.

L'idée générale est de découper la fenêtre graphique en pavées de même taille, et d'afficher un graphe dans chaque pavé. On utilise l'instruction *subplot* en lui spécifiant le nombre de pavés sur la hauteur, le nombre de pavés sur la largeur, et le numéro du pavé dans lequel on va tracer:

subplot (Nbre pavés sur hauteur, Nbre pavés sur largeur, Numéro pavé)

La virgule peut être omise. Les pavés sont numérotés dans le sens de la lecture d'un texte : de gauche à droite et de haut en bas.

Une fois que l'on a tapé une commande *subplot*, toutes les commandes graphiques suivantes seront exécutées dans le pavé spécifié.

Comme exemple taper la suite d'instructions suivantes :

```
>> subplot(221)
>> plot(x,sin(x))
>> subplot(222)
>> plot(x,cos(x),x,sin(x),'-.')
>> subplot(223)
>> plot(cos(x),sin(x))
>> subplot(224)
>> plot(sin(2*x),sin(3*x))
```

3.6 Autres types de représentation

Outre la représentation cartésienne de courbes ou de surfaces, il existe d'autres possibilités pour illustrer graphiquement un résultat. On peut citer parmi les plus utiles, les instructions *contour*, *ezmesh* (pour tracer les courbes de niveau d'une surface paramétrique), *mesh*, *ezplot3*

(courbes paramétriques dans l'espace), *hist*, *rose* (histogramme d'un échantillon de données statistiques), etc...

Chapitre 4: Programmation avec *Matlab et structures de contrôle*

Cette section présente les différentes structures de programmation avec *Matlab*. Pour l'étudiant familier avec la programmation C++ ou Fortran, il ne s'agit que de se familiariser avec la syntaxe propre à *Matlab*. Pour l'étudiant moins familier avec les structures de base en programmation, les exercices sont indiqués pour devenir fonctionnel avec ces structures.
On va parler ensuite des tests et des boucles en commençant par introduire les opérateurs de comparaison et les opérateurs logiques.

4.1 Principe général

Le principe est simple: regrouper dans un fichier une série de commandes *Matlab* et les exécuter en bloc. Tout se passera comme si vous les tapiez au fur et à mesure dans une session *Matlab*. Il est fortement conseillé de procéder de cette façon en créant un fichier programme (M-file) car cela permet notamment de récupérer facilement le travail fait la veille.
Les fichiers de commandes peuvent porter un nom quelconque mais doivent finir par l'extension .m (attention toutefois à certains caractères qui sont interdits : le blanc, le symbole +,.., *Matlab* ne le dira pas tout de suite mais enverra à la première tentative d'exécution un message d'erreur qui dit que le fichier est introuvable).
A titre d'exemple de création d'un fichier de commandes, prenez l'éditeur de *Matlab* et ouvrez un fichier toto.m. Tapez des commandes *Matlab* à l'intérieur, puis sous *Matlab*, tapez :
>> toto
Toutes les commandes du fichier seront exécutées en bloc.

4.2 Où doit se trouver le fichier de commande?

Le plus simple, c'est qu'il se trouve dans le répertoire courant (c'est-à-dire celui où on a lancé *Matlab*). Il peut se trouver aussi dans un répertoire quelconque mais référencé dans la variable path *Matlab*. Tapez cette commande pour en voir le contenu, *Matlab* vous affichera tous les répertoires accessibles. Il est en général conseillé de se créer un répertoire propre ou d'utiliser le répertoire par défaut de *Matlab*. Pour connaître le répertoire actuel il suffit de taper la commande *pwd* dans l'invite de *Matlab*.
Le path peut être modifié avec la commande *addpath*. Cette commande permet de placer le chemin d'accès au fichier dans le fichier qui contient tous les chemins d'accès par défaut ou

déclarés, c'est-à-dire path, et qui est exécuté automatiquement au démarrage de *Matlab*. Voici un exemple:

addpath (genpath('C:\Documents and Settings\admin\Mes documents\MATLAB\Khamlichi'))

Cette commande permet de rajouter le nouveau répertoire Khamlichi, crée dans le répertoire par défaut de *Matlab* qui est nommé MATLAB, au path d'accès.

Ainsi tous les fichiers de commandes présents dans le nouveau répertoire Khamlichi seront accessibles de n'importe où.

4.3 Commentaires et autodocumentation

Tout ce qui se trouve après le symbole % sera considéré comme un commentaire. Il est également possible d'autodocumenter ses fichiers de commande. Ainsi, on peut définir une série de lignes de commentaires ininterrompues au début du fichier précédent toto.m. Lorsque vous taperez:

>> help toto

Ces lignes de commentaires apparaîtront à l'écran. C'est ainsi que fonctionne l'aide en ligne de *Matlab*. En général, un programme doit être suffisamment documenté afin d'améliorer sa lisibilité.

4.4 Suppression de l'affichage

L'affichage des résultats de toutes les commandes n'est pas nécessaire. Pour certaines commandes (création de gros tableaux), cela peut s'avérer fastidieux.

On peut donc placer le caractère ; à la fin d'une ligne de commande pour indiquer à *Matlab* qu'il ne doit pas afficher le résultat.

4.5 Pause dans l'exécution

Si l'on entre la commande pause dans un fichier de commandes, le programme s'arrêtera à cette ligne tant qu'on n'a pas tapé «Entrée» ou «Enter» en cas d'un clavier QWERTY.

4.6 Mode verbeux

Si l'on souhaite qu'au fur et à mesure de son exécution, *Matlab* affiche la séquence de commandes qu'il est en train d'exécuter, il suffit de taper :

>> echo on

Pour revenir au mode normal, on tapera simplement *echo off*. Ce mode peut-être utilisé en combinaison avec pause pour que le programme affiche un commentaire du style «Appuyez sur une touche pour continuer». Il suffit d'écrire le message dans un commentaire :

```
echo  on
pause  %  Appuyez sur une touche pour continuer !
echo  off
```

4.7 Opérateurs de comparaison et logiques

Matlab utilise le langage C. Notons tout d'abord le point important suivant, justement inspiré du langage C:

Matlab représente la constante logique «FAUX» par 0 et la constante «VRAIE» par 1.

Ceci est particulièrement utile par exemple pour définir des fonctions par morceaux.

Il est important de se familiariser avec les opérateurs logiques. Le premier type de ces opérateurs permet de comparer des valeurs entre elles.

Opérateur	Syntaxe Matlab
Egal à	= =
Différent de	~ =
Supérieur à	>
Supérieur ou égal à	>=
Inférieur à	<
Inférieur ou égal à	<=
Négation	~
Ou	|
Et	&

Par exemple, on veut comparer deux valeurs entre elles :

```
>>  a=sin(2*pi);
>>  b=cos(2*pi);
>>  bool=(a>b)
bool=
0
>>  a
```

a=

-2.4493e-016 % ici a devrait égaler 0, la précision est limitée!

>> b

b=

1

Les opérateurs logiques sont intéressants pour construire des fonctions par morceaux. Imaginons que l'on veuille définir la fonction suivante: $f(x)=\begin{cases}\sin(x) & si \quad x>0\\ \sin(2x) & sinon\end{cases}$.

Voilà comment écrire la fonction:

>> f = inline('sin(x).*(x>0) + sin(2*x).*(~(x>0))')

f =

Inline function:

f(x) = sin(x).*(x>0) + sin(2*x).*(~(x>0))

On ajoute les deux expressions sin x et sin 2x en les pondérant par la condition logique définissant leurs domaines de validité. On peut tester que ça marche en représentant la courbe:

>> x=-2*pi:2*pi/100:2*pi;

>> plot(x,f(x))

Il faut noter ici que l'emploie de l'opérateur '= =' est très risqué lorsque l'on compare des valeurs numériques. En effet, la précision de l'ordinateur étant limitée, il est préférable d'utiliser une condition sur la différence comme dans le code suivant :

```
if abs(a-b) < eps  % eps est la précision machine (2.2204e-016)
bool=1;
else
bool=0;
end
```

Il est aussi possible de lier entre elles des conditions par l'opérateur 'et' (&) et 'ou' (|).

Ces notions seront utiles pour la construction des conditions qui seront présentées dans les prochaines sections.

4.8 Boucles if-elseif-else

Dans un programme interviennent souvent des conditions. Les boucles *if-elseif-else* sont une structure de programmation qui est très utile pour rendre compte de cette situation.

En pseudo-code, on peut résumer la chose de la façon suivante:

si CONDITION1, FAIRE ACTION1. % condition 1 remplie

sinon et si CONDITION2, FAIRE ACTION2. % condition 1 non-remplie,

% mais condition 2 remplie

sinon, FAIRE ACTION3 % conditions 1 et 2 non-remplies

En *Matlab*, le pseudo-code précédent devient :

```
if CONDITION1
ACTION1;
elseif CONDITION2
ACTION2;
else
ACTION3;
```

Par exemple, on reçoit un entier a, s'il est impair négatif, on le rend positif. S'il est impair positif, on lui ajoute 1. S'il est pair, on ajoute 2 à sa valeur absolue.

La courte fonction suivante permet de réaliser cette transformation (notez ici, l'emploi du modulo pour déterminer si l'entier est divisible par 2).

```
function b=transf_entier(a)
if a<0 & mod(a,2) ~= 0 % mod permet de trouver le reste d'une division
b=-a;
elseif a>=0 & mod(a,2) ~= 0
b=a+1;
else
b=abs(a)+2;
end
```

4.9 Boucles for

Les boucles *for* sont très utiles dans la plupart des applications mathématiques (par exemple, pour effectuer un calcul sur tous les éléments d'un vecteur).

En *Matlab*, il est parfois beaucoup plus efficace d'utiliser les opérateurs algébriques usuels définis plus tôt (par exemple, le '.*'). Dans les cas où il est impossible de se soustraire à l'utilisation de ces boucles, voici le prototype en pseudo-code qui les trduit.

```
incrément = valeur initiale
Pour incrément=valeur_initiale jusqu'à valeur finale
```

ACTION1...N

AJOUTER 1 à incrément

En *Matlab*, ce pseudo-code devient :

```
for i = 0:valeur_finale
ACTION1;
ACTION2;
...
ACTIONN;
end
```

Remarquez que l'incrément peut être différent de 1, par exemple si l'on veut calculer les carrées des nombres pairs entre 0 et 10 :

```
for i=0:2:10
carre = i^2
end
```

4.10 Boucles while

Une boucle *while* permet de répéter une opération tant qu'une condition (critère) n'est pas remplie. En pseudo-code, elle peut être schématisée de la façon suivante :

Tant que CONDITION est VRAIE

ACTION1...N

En *Matlab*, on écrit ce type de boucle de la manière suivante:

```
while CONDITION
ACTION1;
ACTION2;
...
ACTIONN;
end
```

Ce type de boucle est très souvent utilisé pour faire converger une itération vers une valeur désirée dont la précision est fixée par un test de convergence.

Par exemple, on veut trouver le nombre d'entiers positifs nécessaires pour avoir une somme plus grande que 100. On pourrait réaliser cette tâche de la manière suivante:

```
function n=nombre_entier
n=0;  % initialisation des valeurs
somme=0;
while somme < 100
```

```
n=n+1; % itération de n
somme=somme+n; % nouvelle somme
end
```

4.11 Boucles switch

Les boucles *switch* permettent parfois de remplacer les boucles *if-elseif-else*, particulièrement dans le cas de menus. Le prototype de ce type de boucle en pseudo-code est le suivant :

```
Déterminer CAS
CAS choisi est CAS1
ACTION1
CAS choisi est CAS2
ACTION2
AUTREMENT
ACTION3
```

En *Matlab*, on obtient le code suivant :

```
switch (CAS)
case {CAS1}
ACTION1
case {CAS2}
ACTION2
otherwise
ACTION3
end
```

Par exemple, on veut faire une calculatrice simple en *Matlab*, pour déterminer l'exponentielle ou le logarithme en base e d'un nombre entré par l'utilisateur.

Une manière simple de rendre le programme interactif serait d'utiliser le script suivant :

```
operation=input('Opération: (1) exp ; (2) log ? ');
nombre=input('Valeur: ');
switch operation
case 1
b=exp(nombre)
case 2
b= log(nombre)
otherwise
```

```
disp('mauvais choix -- operation')
end
```

Avec la sortie (par exemple) suivante :

```
>> calcul_rapide
Opération: (1) exp ; (2) log ? 1
Valeur: 0.5
b =
1.6487
```

Chapitre 5: Echanges entre *Matlab* et l'extérieur

5.1 Sauvegarde de données

La commande *save* permet d'écrire toute ou une partie des variables dans un fichier binaire dont l'extension est .mat. La commande *load* permet de recharger les variables contenues dans le fichier. Syntaxe: save nom_fichier var1 var2 var3

Les variables sont simplement séparées par des blancs. Si aucune variable n'est spécifiée, toutes les variables sont sauvegardées dans nom_fichier.mat. La syntaxe de *load* est identique.

La commande *diary* permet d'activer l'écho des commandes et résultats dans un fichier :

diary nom_fichier

Pour désactiver, taper *diary off*.

5.2 Importer des tableaux

Matlab est souvent utilisé comme logiciel d'exploitation de résultats. Exemple classique: on a un programme C++ qui calcule des séries de valeurs, et on souhaite tracer une série en fonction de l'autre.

Le moyen le plus simple est que votre programme écrive un fichier texte organisé sous forme de n colonnes, séparées par des blancs, contenant chacune m lignes :

```
x1    y1    z1    . . .
x2    y2    z2    . . .
x3    y3    z3    . . .
. . .
xm    ym    zm    . . .
```

Cette structure est obligatoire pour que la méthode simple exposée ici fonctionne correctement.

Ce fichier doit porter une extension, qui peut être quelconque, mais différente de .m ou .mat. Admettons qu'il s'appelle toto.res, tapez dans *Matlab*:

```
>> load toto.res
```

Ce faisant, vous créez un nouveau tableau *Matlab*, qui porte le nom du fichier sans extension, soit toto. Vous pouvez maintenant extraire les colonnes de ce tableau, et les mettre dans des vecteurs, par exemple :

```
>> x = toto(:,1)
```

>> y = toto(:,2)

>> z = toto(:,3)

et vous n'avez plus qu'à tracer l'un de ces vecteurs en fonction d'un autre.

Il est aussi possible de sauvegarder les variables dans un fichier ASCII et lui donner un format désiré. Elles pourront ensuite être lues par d'autres logiciels (par exemple, Excel) et avec *Matlab* (en utilisant aussi la fonction *load*).

Dans le cas suivant, on sauvegarde les données dans le fichier ASCII ma_session.dat.

```
>>  save  -ascii  ma_session.dat
>>  clear
>>  load  ma_session.dat
```

La fonction xlsread permet de charger en *Matlab* des fichiers Excel.

```
>>  xlsread  classeur.xls
```

Pour plus d'information à ce sujet, vous pouvez consulter help fileformats.

Références

1. Nicolas Hudon. Initiation à Matlab, (nicolas.hudon@polymtl.ca), URCPC, Ecole Polytechnique de Montréal, 22 janvier 2004

2. Marie Postel. Introduction au logiciel Matlab, Version révisée septembre 2004. Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie.

3. Alfio Quarteroni, Fausto Saleri, Paola Gervasio. Calcul Scientifique; Cours, exercices corrigés et illustrations en MATLAB et Octave. Springer-Verlag, Italie 2010.