Chapitre 2 : Programmation en MATLAB

---

1. Fichiers de commandes

a. 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.

Les fichiers de commandes peuvent porter un nom quelconque, mais doivent finir par l'extension `.m`.

Pour définir un fichier de commandes, prenez votre éditeur préféré, et ouvrez par exemple 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.

b. Où doit se trouver mon fichier de commande ?

Le plus simple, c'est qu'il se trouve dans le directory courant (c'est-à-dire celui où vous avez lancé MATLAB). Il peut se trouver aussi dans un directory référencé dans la variable **`path`** MATLAB. Tapez cette commande pour en voir le contenu.

Le path peut être modifié avec les commandes **`path`** et **`addpath`**.

Par exemple, si on tape la commande :

```
>>path,'c:\USERS')
```

Tous les fichiers de commandes présents dans le répertoire 'c:\USERS' seront accessibles de n'importe où.

Il est d'autre part possible de se déplacer dans l'arborescence comme sous UNIX, avec la commande **`cd`**. La commande UNIX **`pwd`** permettant de voir dans quel directory l'on se trouve existe également sous MATLAB

c. Commentaires et auto-documentation

Les lignes blanches sont considérées comme des commentaires.

Tout ce qui se trouve après le symbole **%** sera également considéré comme un commentaire.

Il est également possible d'auto-documenter ses fichiers de commande. Ainsi, définissez une série de lignes de commentaires ininterrompues au début de votre fichier 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. C'est tellement simple que ça serait dommage de s'en priver.

d. Suppression de l'affichage

Jusqu'à présent, nous avons vu que toutes les commandes tapées sous MATLAB affichaient le résultat. 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.

e. Pause dans l'exécution

Si vous entrez la commande `pause` dans un fichier de commandes, le programme s'arrêtera à cette ligne tant que vous n'avez pas tapé « Entrée».

f. Mode verbeux

Si vous souhaitez qu'au fur et à mesure de son exécution, MATLAB vous affiche les commandes qu'il est en train d'exécuter, vous pouvez taper :

```
>> echo on
```

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

```
echo on
pause % Allez, appuyez un petit coup sur Entrée pour continuer !
echo off
```

---

2. Fonctions

Nous avons vu un certain nombre de fonctions prédéfinies. Il est possible de définir ses propres fonctions. La première méthode permet de définir des fonctions simples sur une ligne de commande. La seconde, beaucoup plus générale permet de définir des fonctions très évoluées en la définissant dans un fichier.

a. Fonctions inline

Admettons que je veuille définir une nouvelle fonction que j'appelle **sincos** définie mathématiquement par :

On écrira :

```
>> sincos = inline('sin(x)-x*cos(x)')

sincos =
     Inline function:
     sincos(x) = sin(x)-x*cos(x)
```

N'oubliez cependant pas les quotes, qui définissent dans MATLAB des chaînes de caractères. On peut maintenant utiliser cette nouvelle fonction :

```
>> sincos(pi/12)

ans =
    0.0059
```

Essayons maintenant d'appliquer cette fonction à un tableau de valeurs :

```
>> sincos(0:pi/3:pi)

??? Error using ==> inline/subsref
Error in inline expression ==> sin(x)-x*cos(x)
??? Error using ==> *
Inner matrix dimensions must agree.
```

Ça ne marche pas. Vous avez compris pourquoi ? (c'est le même problème que pour l'instruction **plot(x, x*sin(x))** vue au chapitre sur les Graphiques 2D). MATLAB essaye de multiplier **x** vecteur ligne par **cos(x)** aussi vecteur ligne au sens de la multiplication de matrice ! Par conséquent il faut bien utiliser une multiplication terme à terme **.*** dans la définition de la fonction :

```
>> sincos = inline('sin(x)-x.*cos(x)')

sincos =
   Inline function:
    sincos(x) = sin(x)-x.*cos(x)
```

et maintenant ça marche :

```
>> sincos(0:pi/3:pi)

ans =
    0    0.3424    1.9132    3.1416
```

On peut donc énoncer la règle approximative suivante :

Lorsque l'on définit une fonction, il est préférable d'utiliser systématiquement les opérateurs terme à terme .* ./ et .^ au lieu de * / et ^, si l'on veut que cette fonction puisse s'appliquer à des tableaux.

On aura remarqué que la commande **`inline`** reconnaît automatiquement que la variable est x. On peut de même définir des fonctions de plusieurs variables :

```
>> sincos = inline('sin(x)-y.*cos(x)')

sincos =
     Inline function:
     sincos(x,y) = sin(x)-y.*cos(x)
```

Ici encore, MATLAB a reconnu que les deux variables étaient `x` et `y`. L'ordre des variables (affiché à l'écran) est celui dans lequel elles apparaissent dans la définition de la fonction. On peut cependant spécifier explicitement l'ordre des variables (voir la doc).

Remarques :

- Avec en argument un vecteur, la fonction retourne un vecteur :
- Avec en argument une matrice, la fonction retourne une matrice :

b. Fonctions définies dans un fichier

C'est la méthode la plus généraliste, et elle permet notamment de réaliser des fonctions ayant plusieurs sorties. Commençons par reprendre l'exemple précédent (`sincos`).

Méthode :

L'ordre des opérations est le suivant :

1. Éditer un nouveau fichier appelé `sincos.m`
2. Taper les lignes suivantes :

   `function` s = sincos(x)

   s = sin(x)-x.*cos(x);

3. Sauvegardez

Le résultat est le même que précédemment :

```
>> sincos(pi/12)

ans =
     0.0059
```

On remarquera plusieurs choses :

- l'utilisation de **`.*`** pour que la fonction soit applicable à des tableaux.
- la variable **`s`** n'est là que pour spécifier la sortie de la fonction.
- l'emploi de **`;`** à la fin de la ligne de calcul, sans quoi MATLAB afficherait deux fois le résultat de la fonction.

Donc encore une règle :

Lorsque l'on définit une fonction dans un fichier, il est préférable de mettre un `;` à la fin de chaque commande constituant la fonction. Attention cependant à ne pas en mettre sur la première ligne.

Un autre point important :

Le nom du fichier doit porter l'extension `.m` et le nom du fichier sans suffixe doit être exactement le nom de la fonction (apparaissant après le mot-clé `function`)

En ce qui concerne l'endroit de l'arborescence où le fichier doit être placé, la règle est la même que pour les fichiers de commande (voir le paragraphe concernant la variable **`path`**).

Il est permis de comparer la concision de MATLAB pour la déclaration d'une fonction à la lourdeur du FORTRAN ou de tout autre langage, où les paramètres formels et toutes les variables locales doivent être déclarées.

Voyons maintenant comment définir une fonction comportant plusieurs sorties. On veut réaliser une fonction appelée `cart2pol` qui convertit des coordonnées cartésiennes (x, y) (entrées de la fonction) en coordonnées polaires (r, θ) (sorties de la fonction). Voilà le contenu du fichier `cart2pol.m` :

```
function [r, theta] = cart2pol (x, y)

r = sqrt(x.^2 + y.^2);
theta = atan (y./x);
```

On remarque que les deux variables de sortie sont mises entre crochets, et séparées par une virgule.

Pour utiliser cette fonction, on écrira par exemple :

```
>>[rr,tt] = cart2pol(1,1)

rr =
    1.4142
tt =
    0.7854
```

On affecte donc simultanément deux variables `rr` et `tt` avec les deux sorties de la fonction, en mettant ces deux variables dans des crochets, et séparés par une virgule. Les crochets ici n'ont bien sûr pas le même sens que pour les tableaux.

Il est possible de ne récupérer que la première sortie de la fonction. MATLAB utilise souvent ce principe pour définir des fonctions ayant une sortie «principale» et des sorties «optionnelles» . (C'est le cas par exemple de **`eig`** qui renvoie les valeurs propres d'une matrice et éventuellement la matrice de passage associée.)

Ainsi, pour notre fonction, si une seule variable de sortie est spécifiée, seule la valeur du rayon polaire est renvoyée. Si la fonction est appelée sans variable de sortie, c'est **ans** qui prend la valeur du rayon polaire :

```
>> cart2pol(1,1)

ans =

    1.4142
```

c. Portée des variables

À l'intérieur des fonctions comme celle que nous venons d'écrire, vous avez le droit de manipuler trois types de variables :

- *Les variables d'entrée de la fonction. Vous ne pouvez pas modifier leurs valeurs.*
- *Les variables de sortie de la fonction. Vous devez leur affecter une valeur.*
- *Les variables locales. Ce sont des variables temporaires pour découper des calculs par exemple. Elles n'ont de sens qu'à l'intérieur de la fonction et ne seront pas «vues» de l'extérieur*

Et C'EST TOUT ! Imaginons maintenant que vous écriviez un fichier de commande (l'équivalent du programme principal), et une fonction (dans deux fichiers différents bien sûr), le fichier de commande se servant de la fonction. Si vous voulez passer une valeur du fichier de commandes à la fonction, vous devez normalement passer par une entrée de la fonction. Cependant, on aimerait bien parfois que la variable **A** du fichier de commandes soit utilisable directement par la fonction.

Pour cela on utilise la directive **global**.

Prenons un exemple :

Vous disposez d'une corrélation donnant le C_p (chaleur massique à pression constante) d'un gaz en fonction de la température et vous voulez en faire une fonction. On pourrait faire une fonction à 5 entrées, pour T, A,B,C,D mais il est plus naturel que cette fonction ait seulement T comme entrée. Comment passer A,B,C,D qui sont des constantes ? Réponse : on déclare ces variables en global tant dans le fichier de commandes que dans la fonction, et on les affecte dans le fichier de commandes.

Fichier de commandes :

```
global A B C D

A = 9.9400e-8;
B = -4.02e-4;
C = 0.616;
D = -28.3;
 T = 300:100:1000 % Température en Kelvins
plot(T,cp(T))    % On trace le CP en fonction de
```

```
T
```

La fonction :

```
function y = cp(T)
global A B C D
y = A*T.^3 + B*T.^2 +C*T + D;
```

---

3. Structures de contrôle

a. Opérateurs de comparaison et logiques

Notons tout d'abord le point important suivant, 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.
Ensuite, les deux tableaux suivants comportent l'essentiel de ce qu'il faut savoir :

Opérateur	Syntaxe MATLAB
égal à	==
différent de	~=
supérieur à	>
supérieur ou égal à	>=
inférieur à	<
inférieur ou égal à	<=

Opérateurs de comparaison

Opérateur	Syntaxe MATLAB
Négation	~
Ou	|
Et	&

Opérateurs logiques

On peut se demander ce qui se passe quand on applique un opérateur de comparaison entre deux tableaux, ou entre un tableau et un scalaire. L'opérateur est appliqué terme à terme. Ainsi :

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

```
     3     2

>> A>2

ans =
     0     1
     1     0
```

Les termes de A supérieur à 2 donnent 1 (vrai), les autres 0 (faux). Il est sans intérêt d'effectuer cela dans une instruction `if` (voir section suivante), en revanche c'est intéressant pour construire des fonctions par morceaux. Imaginons que l'on veuille définir la fonction suivante :

```
function y = f (x)
y = 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 :

```
>> x=-2*pi:2*pi/100:2*pi;
>> plot(x,f(x))
>> grid
```

b. Instructions conditionnelles if

La syntaxe est la suivante :

`if` *condition logique*
  instructions
`elseif` *condition logique*
  instructions
...
`else`
  instructions
`end`

On remarquera qu'il n'y a pas besoin de parenthèses autour de la condition logique. Notons qu'il est souvent possible d'éviter les blocs de ce type en exploitant directement les opérateurs de comparaison sur les tableaux (voir la fonction par morceaux définie dans la section précédente), ainsi que la commande **`find`**.

Il existe aussi une structure du type **`switch...case`** (voir la doc)

c. Boucles for

La syntaxe est la suivante :

```
for variable = valeur début:pas:valeur fin
  instructions
end
```

L'originalité réside dans le fait que la variable de boucle peut être réelle.

d. Boucles while

La syntaxe est la suivante :

```
while condition logique
  instructions
end
```

Le fonctionnement est classique.