PROGRAMMATION AVEC MATLAB

Dans les précédentes sections, nous avons présenté des séries de commandes lancées depuis la `command window`. Pour des calculs complexes et répétitifs, il est préférablement (ou plutôt indispensable) de rassembler l'ensemble des commandes dans un fichier qui constituera le programme à exécuter. On distingue deux types de fichiers dans Matlab, également appelés `m-files` : les scripts et les fonctions. Bien que l'environnement de Matlab propose son propre éditeur (fenêtre `Editor`), ces fichiers sont de simples fichiers textes avec une extension .m. Vous pouvez donc utiliser votre éditeur de texte préféré pour créer vos programmes (sans oublier de modifier l'extension). A partir de Matlab, un `m-file` est créé ou ouvert, soit depuis le menu `Fichier` (`New > M-File`), soit depuis l'invite en tapant :

```
>> edit monfichier.m
```

Un `m-file` est reconnu, et donc exécutable, s'il se trouve dans le répertoire courant (`current directory`) ou si le répertoire contenant est spécifié dans le `PATH`.

Nous allons voir que Matlab offre la possibilité de réaliser de véritables applications très élaborées. Notons qu'il utilise un langage de programmation interprété, c'est-à-dire qu'il n'y a aucune phase de compilation et les instructions du code sont directement exécutées à leur lecture.

Fichiers SCRIPT

Un fichier `script` permet regrouper des séries de commandes Matlab. Cela évite d'avoir à saisir plusieurs fois de longues suites d'instructions. A son lancement, les instructions qu'il contient s'exécutent séquentiellement comme si elles étaient lancées depuis l'invite de commande. Un script stocke ses variables dans le `workspace`, lequel est partagé par tous les scripts. Ainsi, toutes les variables créées dans les scripts sont visibles depuis la `command window` et vice versa. Lorsque Matlab détecte une erreur, le programme s'arrête et un message d'erreur s'affiche à l'écran (avec le numéro de la ligne où l'erreur est détectée).

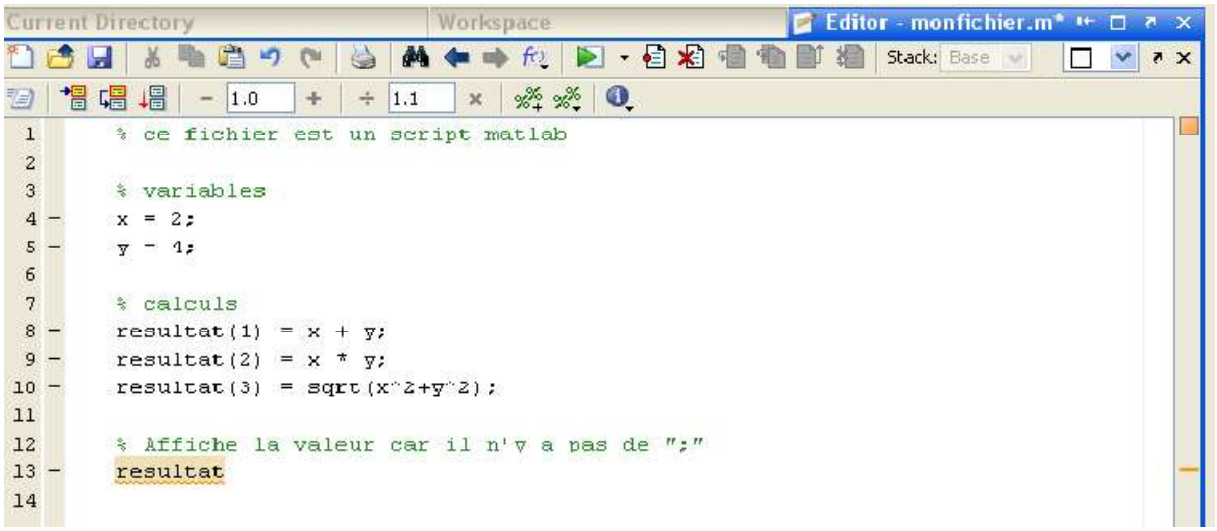
Editons notre script `monfichier.m`.

```
x = 2;
y = 4;
resultat(1) = x + y ;
resultat(2) = x * y ;
resultat(3) = sqrt(x^2+y^2);
resultat
```

Notre script peut ensuite être exécuté, soit en tapant son nom (sans l'extension) à l'invite de commande, soit en cliquant sur le bouton « run » de l'éditeur (icône avec un triangle vert).

```
>> monfichier
resultat =
6.0000 8.0000 4.4721
```

Des annotations peuvent être ajoutées dans le code afin de le commenter. Pour cela, chaque ligne de commentaires doit être précédée par le caractère %. Les mots suivant ce symbole ne seront pas interprétés.



Des annotations peuvent être ajoutées dans le code afin de le commenter. Pour cela, chaque ligne de commentaires doit être précédée par le caractère %. Les mots suivant ce symbole ne seront pas interprétés.
De manière générale, il est essentiel d'inclure dans le code un nombre conséquent de commentaires. Ils permettent de documenter un programme et facilitent la relecture, la maintenance de celui-ci.

Fichiers FUNCTION

Le principe d'une fonction est d'effectuer des opérations à partir d'une ou plusieurs entrées et fournir une ou plusieurs sorties (résultat). Les variables d'entrées sont des paramètres à spécifier en argument de la fonction, tandis que les variables de sorties sont des valeurs quelle renvoie. Un m-file function est tout à fait semblable aux fonctions intégrées de Matlab. Par exemple, la fonction `length` renvoie la taille du tableau entré en argument.

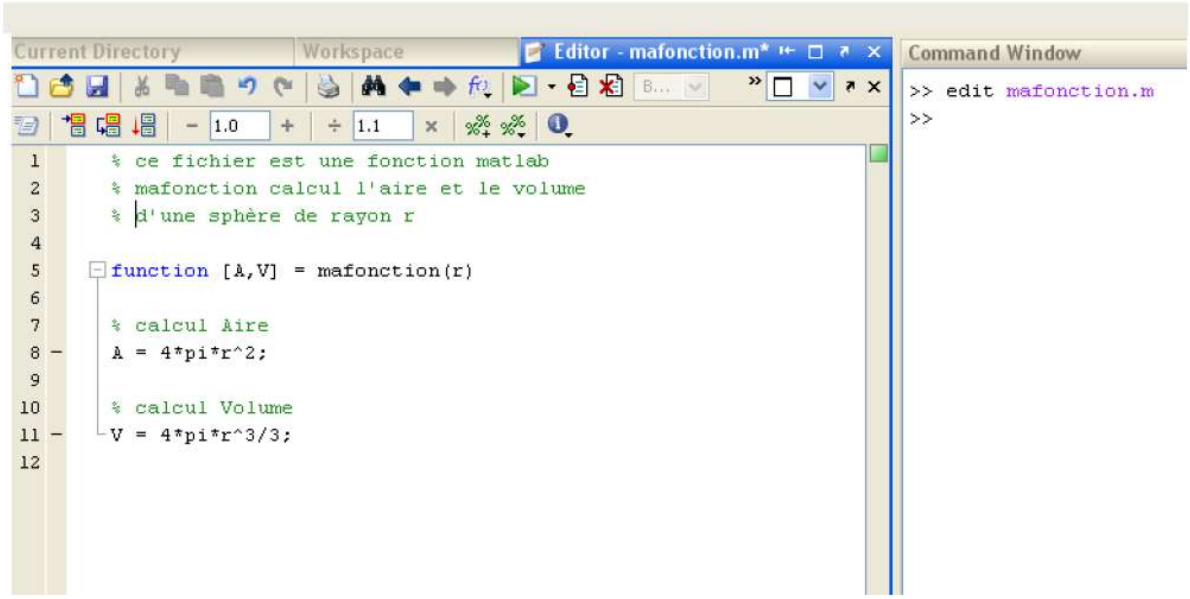
```
>> taille = length(tab);
```

Un m-file est définit comme une fonction en plaçant en tête du fichier le mot clé `function` suivit de son prototype. Un prototype est de la forme

```
[s1,s2,…] = nomfonction(e1,e2,…)
```

Le membre de gauche regroupe les sorties renvoyées par la fonction et les variables entres parenthèses sont les entrées. Le nom du fichier doit être identique au nom de la fonction. Ecrivons dans l'éditeur notre fonction qui permet de calculer l'aire et le volume d'une sphère pour un rayon donné.

```
% ce fichier est une fonction matlab
% mafonction calcul l'aire et le volume d'une sphère de rayon r
function [A,V] = mafonction(r)
% calcul Aire
A = 4*pi*r^2;
% calcul Volume
V = 4*pi*r^3/3;
```



L'appel de la fonction s'effectue de la même façon que pour les fonctions prédéfinies dans le logiciel

```
>> [aire,volume] = mafonction(2);
>> aire
aire =
50.2655
>> volume
volume =
33.5103
```

Le point fondamental qui différencie une fonction d'un script est le fait que les variables internes soient locales, c'est-à-dire que les variables définies dans une fonction n'existent que dans celle-ci. De plus, les variables du `workspace` ne sont pas visibles depuis une fonction. Ainsi, dans notre exemple, les paramètres `A`, `V` et `r` ne sont pas connues dans le `workspace`.

```
>> whos
Name          Size            Bytes  Class     Attributes
aire          1x1                 8  double
volume        1x1                 8  double
```

Pour pouvoir utiliser une variable partagée par le `workspace` et une (voire des) fonction(s), celle-ci doit être déclarée comme `global` à la fois dans la `command window` et dans la (les) fonction(s). Ajoutons à notre fonction

```
global x;
x = r;
```

et écrivons les lignes suivantes dans la `command window`

```
>> global x;
>> x = 10;
>> [aire,volume]=mafonction(3);
>> x
x =
3
```

constate On donc que la variable globale x, a été modifiée lors de l'appel de la fonction.

Notons toutefois que l'utilisation de variables globales est déconseillée, car souvent source d'erreurs d'exécution, et doit donc être minimisée.

Des instructions permettent de contrôler les arguments d'entrées et de sorties d'une fonction:

`nargin`	retourne le nombre d'arguments d'entrée
`nargout`	retourne le nombre d'arguments de sortie
`nargchk`	vérifie le nombre d'arguments d'entrée
`inputname`	retourne le nom d'un argument d'entrée

Opérateurs relationnels et logiques

Comme dans tout langage de programmation, Matlab possède des opérateurs qui permettent d'établir des expressions renvoyant en résultat une valeur logique, c'est-à-dire 0 ou 1. Ces expressions logiques sont généralement utilisées dans les structures de contrôle présentées dans la prochaine section.

Opérateurs relationnels

Ces opérateurs comparent deux opérandes de même dimension :

==	égal à
~=	différent de
>	strictement supérieur à
>=	supérieur ou égal à
<	strictement inférieur à
<=	inférieur ou égal à

Lorsque deux scalaires sont comparés, le résultat est un scalaire qui vaut 1 si la relation est vrai et 0 si elle est fausse. Si deux matrices sont comparées, le résultat est une matrice de même dimension constituée de 1 et 0, la relation étant testée élément par élément.

```
>> 10 > 9
ans =
1
>> 2 == 3
ans =
0
>> 4 ~= 7
ans =
1
>> [1 4 ; 7 3] <= [0 6 ; 7 2]
ans =
0 1
1 0
```

Opérateurs logiques

Ces opérateurs effectuent un test logique entre deux variables logiques de même dimension:

`&`	et
`|`	ou
`~`	non
`xor`	ou exclusif
`any(x)`	retourne 1 si un des éléments de x est non nul
`all(x)`	retourne 1 si tous les éléments de x sont nuls

Le résultat vaut 1 si le test est vrai et 0 s'il est faux. Pour des matrices, l'opération s'effectue aussi élément par élément. Concernant les opérandes, une valeur est considérée comme fausse (=0) si elle est nulle. Elle est considérée comme vrai (=1) si elle est non nulle.

```
>> x = [0 1 0 1];
>> y = [0 0 1 1];
>> x & y
ans =
0 0 0 1
>> x | y
ans =
0 1 1 1
>> ~x
ans =
1 0 1 0
>> xor(x,y)
ans =
0 1 1 0
>> 0 | 3
ans =
1
>> ~(-2.4)
ans =
0
```

Structures de contrôle

Dans sa forme la plus simple, le déroulement d'un programme est linéaire dans le sens où les instructions qui le composent s'exécutent successivement. Les structures de contrôle sont des mécanismes qui permettent de modifier la séquence d'exécution des instructions. Plus précisément, lors de l'exécution, en fonction des conditions réalisées certaines parties précises du code seront exécutées.

Branchement conditionnel **(`if` … `elseif` … `else`)**

Cette structure permet d'exécuter un bloc d'instructions en fonction de la valeur logique d'une expression. Sa syntaxe est :

```
if expression
instructions …
end
```

L'ensemble des instructions *instructions* est exécuté seulement si *expression* est vraie. Plusieurs tests exclusifs peuvent être combinés.

```
if expression1
instructions1 …
elseif expression2
instructions2 …
else
instructions3 …
end
```

Plusieurs `elseif` peuvent être concaténés. Leur bloc est exécuté si l'expression correspondante est vraie et si toutes les conditions précédentes n'ont pas été satisfaites. Le bloc *instruction3* associé au `else` est quant à lui exécuté si aucune des conditions précédentes n'a été réalisées.

```
if x > 0
disp('x est positif');
elseif x == 0
disp('x est nul');
else
x = 1;
end
```

Branchement multiple (`switch … case`)

Dans cette structure, une expression numérique est comparée successivement à différentes valeurs. Dès qu'il y a identité, le bloc d'instructions correspondant est exécuté. Sa syntaxe est :

```
switch expression
    case valeur1,
        instructions1 …
    case valeur2,
        instructions2 …
    case valeur3,
        instructions3 …
    ……..
   otherwise
        instructions …
end
```

L'expression testée, *expression*, doit être un scalaire ou une chaîne de caractère. Une fois qu'un bloc *instructionsi* est exécuté, le flux d'exécution sort de la structure et reprend après le `end`. Si

```
switch x
    case 0,
       resultat = a + b;
    case 1,
       resultat = a * b;
    case 2,
       resultat = a/b;
    case 3,
       resultat = a^b;
    otherwise
       resultat = 0;
end
```

aucun `case` vérifie l'égalité, le bloc qui suit `otherwise` est exécuté. En fonction de la valeur de `x` une opération particulière est effectuée. Par défaut, `resultat` prend la valeur 0.

Boucle conditionnelle (`while` …)

Ce mécanisme permet de répéter une série d'instructions tant qu'une condition est vérifiée. Sa syntaxe est :

```
while expression
    instructions …
end
```

Le terme *`expression`* est une expression logique. Si cette dernière est vraie, le bloc *`instructions`* est exécuté. Puis, *`expression`* est de nouveau testé. L'exécution du bloc est répétée tant que le test est vrai.

```
compteur = 0;
while compteur < 10
disp('toujours dans la boucle') ;
compteur = compteur + 1;
end
```

Cet exemple affiche 10 fois la chaîne de caractère `toujours dans la boucle`.

Boucle itérative (`for` …)

Cette boucle exécute le bloc interne autant de fois que spécifié par une variable jouant un rôle de compteur. Sa syntaxe est :

```
for variable = debut:increment:fin
instructions …
end
```

Le compteur *`variable`* est initialisé à la valeur *`debut`* et évolue jusqu'à la valeur *`fin`* par pas de *`increment`*. A chaque itération, le bloc *`instructions`* est exécuté. Généralement, *`variable`* est un scalaire, et souvent un entier.

```
N = 5 ;
for k = 1:N
      x(k)= 1/k;
end
```

Cet exemple construit élément par élément un vecteur x de dimension 5.