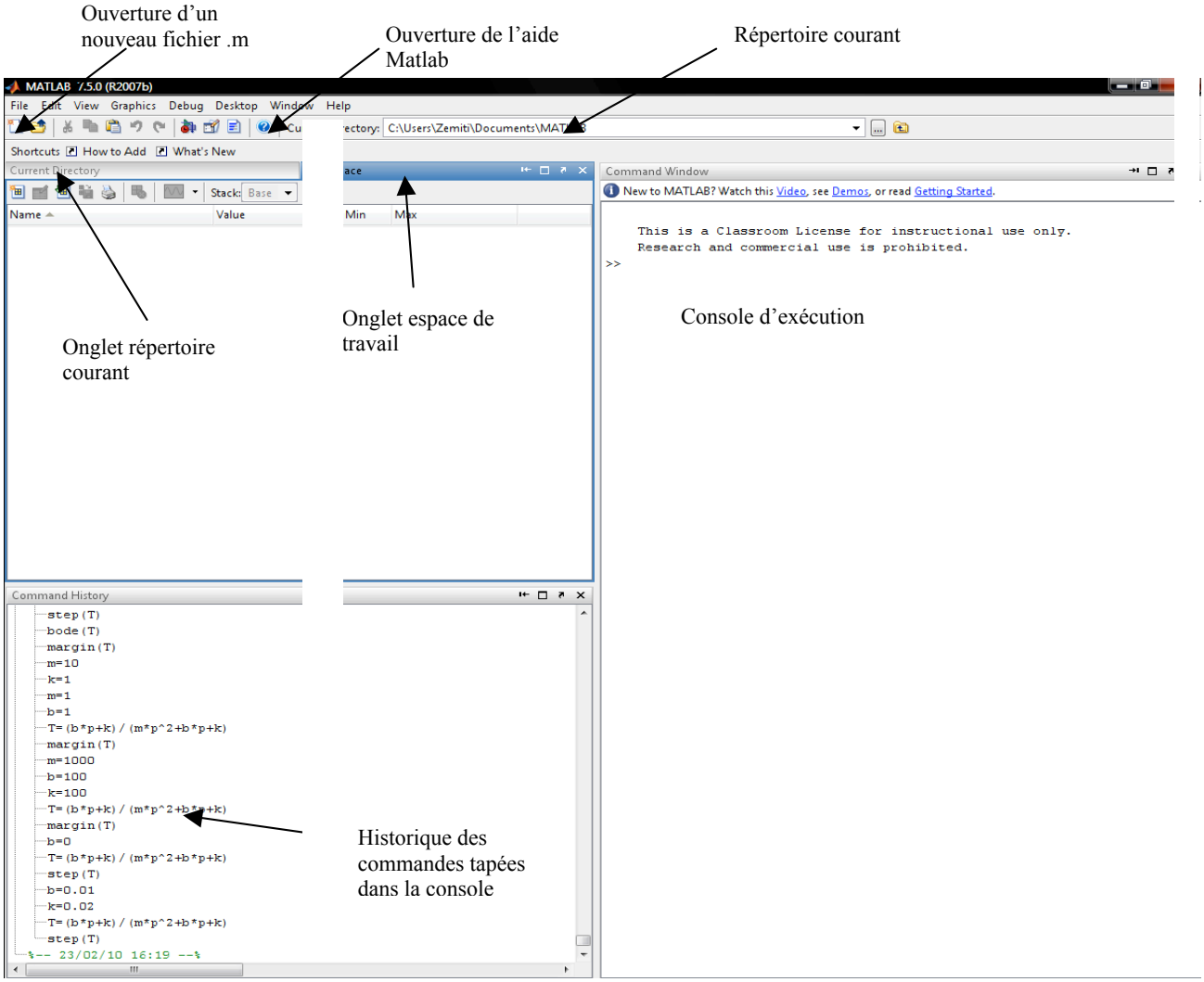
TP Informatique appliquée au génie électrique

TP N°I. Présentation générale

L'objectif de ce TP est de vous initier au logiciel MATLAB de la compagnie Mathworks et la programmation dans cet environnement. L'idée est de vous exposer les bases de cet outil c travail.



MATLAB est beaucoup plus qu'un langage de programmation. Il s'agit d'une console d'exécution (*shell*) au même titre que les consoles DOS ou UNIX. Comme toutes les consoles, MATLAB permet d'exécuter des fonctions, d'attribuer des valeurs à des variables, etc. Plus spécifiquement, la console MATLAB permet d'exécuter des opérations mathématiques, de manipuler des matrices, de tracer facilement des graphiques (cf. figure ci-dessus).

Le nom MATLAB vient de la contraction **MAT**rix **LAB**oratory, ce qui signifie que toutes les variables sont considérées comme des matrices. Une variable scalaire est vue par MATLAB comme une matrice 1x1 (une ligne, une colonne), comme le montre l'exemple suivant dans lequel on affecte à la variable `x` la valeur 5 et on demande ensuite ses dimensions par la commande `size`

```
>> x=5
x =
     5
>> size(x)
ans =
     1     1
>>
```

On remarque qu'après l'affectation de la variable, MATLAB affiche de suite la valeur affectée. Afin d'éviter ceci, il suffit de suivre la commande par un **point virgule**, ce qui permettra éventuellement des programmes plus rapides (pas de perte de temps du à l'affichage).

```
>> x=5;
>> size(x)
ans =
     1     1
>>
```

MATLAB, en tant que lan ;age scientifique, a prévu des constantes prédéfinies :

```
>> pi
ans =
    3.1416
>> eps
ans =
  2.2204e-016
>>
```

Tant qu'elles ne sont pas ıffectées à des vecteurs, les variables `i,j` représentent le nombre imaginaire $\sqrt{-1}$:

```
>> i
ans =
        0 + 1.0000i

>> j
ans =
        0 + 1.0000i
```

L'utilisateur peut affecter donc des valeurs à des variables et affecter des opérations à ces variables (+ - / * …).

```
>> x=4;
>> y=2;
>> x+y
ans =
     6
>> x*y
ans =
     8
>> ans + 2
ans =
    10
>>
```

Ici, il faut noter que lorsque l'utilisateur ne fixe pas de variable de sortie, MATLAB place le résultat d'une opération dans `ans`. Cette variable temporaire peut bien sûr être utilisée pour un calcul suivant comme le montre l'exemple précédent.

Il est toujours possible de connaître les variables utilisées et leur type à l'aide de la fonction `whos`. Par exemple, pour les manipulations précédentes :

Ces variables sont aussi présentes dans l'onglet **espace de travail** (**Workspace**).

```
>> 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 :

```
>> a= x+y
a =
     6
>> b= x*y
b =
     8
>> whos
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  ans       1x1                         8  double array
  b         1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array

Grand total is 5 elements using 40 bytes
```

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
  ans       1x1                         8  double array
  b         1x1                         8  double array
  y         1x1                         8  double array

Grand total is 4 elements using 32 bytes
```

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 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 le *double* tel que présenté plus haut et les variables *char*, pour le texte. Pour les variables *char*, la déclaration se fait entre apostrophe :

```
>> mot1='hello'
mot1 =
hello
```

Il est possible de concaténer les mots à l'aide des parenthèses carrées [] (la fonction `strcat` de MATLAB permet d'effectuer sensiblement la même tâche) :

```
>> mot1='hello';
>> mot2='les';
>> mot3='Mephys';
>> phrase=[mot1  ' '  mot2  ' '  mot3]  % l'emploi de ' ' permet d'introduire
                                        % un espace.
phrase =
hello les Mephys
```

Supposons que l'on veuille écrire un programme qui calcul la racine carrée d'un nombre entré par l'utilisateur et qui affiche le résultat dans une phrase. On peut convertir les nombres en chaîne de caractères en utilisant la fonction `num2str (numeric to String)`.

```
a=input('Entrez un nombre :'); % utilisation de input, l'utilisateur doit
                               % entrer un nombre
Entrez un nombre : 2

>> b=sqrt(a);
>> phrase_reponse = ['La racine carrée de ' num2str(a) ' est ' num2str(b) ];
                                    % Composition de la phrase de sortie
```

```
>> disp(phrase_reponse) % utilisation de display pour afficher le résultat à
l'écran.

La racine carrée de 2 est 1.4142
```

La prochaine section montre comment réaliser de courts programmes.

TP N°II. 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 de courts programmes pour effectuer les calculs désirés.

Il existe deux types de fichiers qui peuvent être programmés avec MATLAB : les fichiers **SCRIPT** et **FUNCTION**. Dans les deux cas, il faut lancer l'éditeur de fichier et sauvegarder le fichier avec l'extension *.m*.

1. Fichier SCRIPT

Le fichier SCRIPT permet de lancer les mêmes opérations que celles écrites directement à l'invite MATLAB. Toutes les variables utilisées dans un SCRIPT sont disponibles à l'invite MATLAB. Vous devez créer vous-même ce ficher en faisant créer un nouveau fichier MFile et le sauvegarder par exemple sous le nom test.m.

Habituellement, on utilise les fichiers SCRIPT afin de :

- Initialiser le système (fonctions *clear*)
- Déclarer les variables
- Effectuer les opérations algébriques
- Appeler les fonctions
- Tracer les figures…

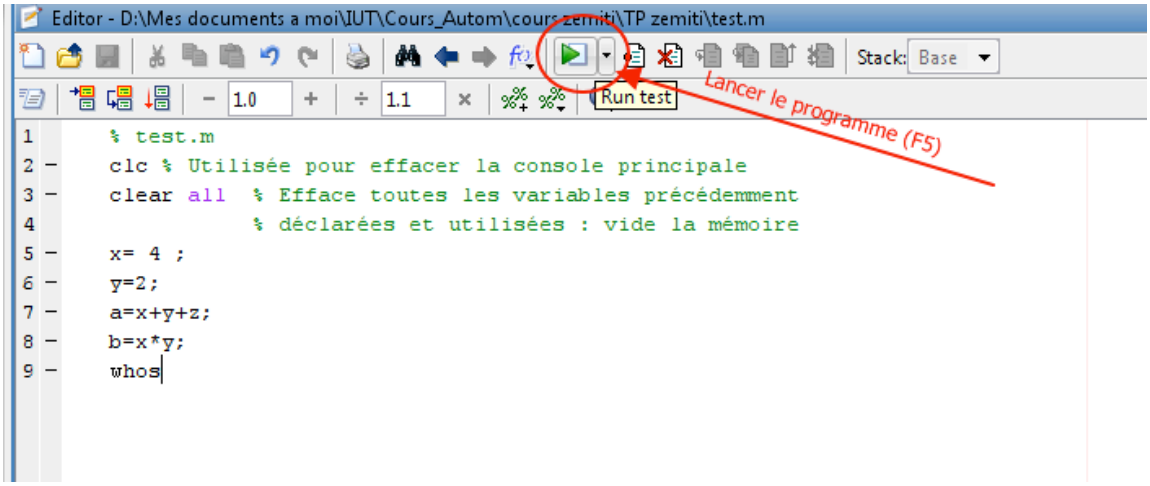
Il est utile ici de noter que le langage MATLAB n'est pas un langage compilé (contrairement au langage *C++*, par exemple). A chaque appel d'un SCRIPT (ou d'une FUNCTION), le logiciel lit et exécute les programmes ligne par ligne. Lorsque MATLAB détecte une erreur, le logiciel arrête et un message d'erreur 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" vos programmes rapidement et efficacement.**

Pour lancer le programme vous avez le choix :

- soit d'utiliser la fenêtre principale en tapant le nom du programme :

```
>> test
```

- soit de le lancer/exécuter (Run) directement depuis la fenêtre du fichier test.m (raccourci clavier F5)



Application :
Ouvrir un nouveau fichier .m le sauvegarder sous le nom test.m. Taper le texte ci-dessus dans le fichier. Exécuter/lancer le programme (méthode de votre choix).

2. Fichiers 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 les variables internes ne sont pas disponibles à l'invite MATLAB. Ces fonctions seront ensuite utilisées/appelées dans les programmes (SCRIPT). Autrement dit, **ON N'EXECUTE JAMAIS UN PROGRAMME DE TYPE FONCTION. ON L'APPELLE DEPUIS UN AUTRE PROGRAMME OU DEPUIS LA CONSOLE.**
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.

Par exemple, la fonction suivante (avec une seule sortie, le résultat de l'addition) :
Ouvrir un nouveau fichier que vous nommer `MaFonction.m`

```
function nom_de_la_sortie1 =  MaFonction(argument1, argument2)
% Le nom de la fonction doit être exactement le même que le nom de votre fichier.m
% ici il faut mettre les commentaires sur ce que fait votre fonction :
% c = MaFonction (a , b) : Calcule la somme de a et de b et met le
% résultat dans c

nom_de_la_sortie1 = argument1 + argument2 ;
```

Tous les commentaires (en vert) qui suivent (sans espace entre les lignes) la première ligne de définition de la fonction constituent du texte d'aide de la fonction. Si on exécute depuis la console MATLAB la commande `help MaFonction,` nous obtenons le texte que nous avons inséré sous forme de commentaires juste après l'en-tête de la fonction. Le texte d'aide s'arrête dès la rencontre d'une ligne vide.

Si on appelle la fonction créée depuis la console :

```
>> toto = MaFonction(1,2)
toto =
     3
>> whos
```

```
  Name       Size                    Bytes  Class
  toto       1x1                         8  double array

Grand total is 1 element using 8 bytes
```

Si on veut inclure le calcul de la multiplication dans `MaFonction`, on modifie les sorties de la manière suivante (modifier le fichier précédent) :

```
function [nom_de_la_sortie1, nom_de_la_sortie2]=  MaFonction(argument1, argument2)
% Le nom de la fonction doit être exactement le même que le nom de votre fichier .m
% ici il faut mettre les commentaires sur ce que fait votre fonction :
% [c,d] = MaFonction (a , b) : Calcule la somme de a et de b et met le
% résultat dans c, et, calcule le produit de a et de b et met le
% résultat dans d

nom_de_la_sortie1 = argument1 + argument2 ;
nom_de_la_sortie2 = argument1 * argument2 ;
```

pour obtenir (taper ce sui suite dans la console)

```
>> clear ; [somme, produit]  = MaFonction(1,2)
somme =
     3
produit =
     2
>> whos
  Name          Size                    Bytes  Class
  somme         1x1                         8  double array
  produit       1x1                         8  double array

Grand total is 2 elements using 16 bytes
```

TP N° III. Opérations mathématique avec MATLAB : Scalaires, vecteurs, matrices

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 est une matrice 1xn. Contrairement aux langages de programmation usuels (i.e. langage 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
a =
     0
```

Les **vecteurs lignes** se déclarent de la manière suivante :

```
>> V_ligne1=[0  1  2]
V_ligne1 =
     0     1     2

>> V_ligne2=[0,  1,  2]
V_ligne2 =
     0     1     2
>> size(V_ligne2)

ans =

     1     3
```

On sépare les éléments par des espaces ou des virgules.
Comme chaque élément de MATLAB, le vecteur est une matrice. Ici c'est une matrice 1x3.

Les **vecteurs colonnes** se déclarent de la manière suivante :

```
>> V_colonne1 = [1 ; 3 ; 5]
```

```
V_colonne1 =
     1
     3
     5

>> V_colonne2 = [1   % retour chariot
2                  % retour chariot
3]                 % retour chariot

V_colonne2 =

     1
     2
     3
>> size(V_colonne2)

ans =

     3     1
```

On sépare les éléments par des points-virgules ou on utilise le retour chariot.
Ici le vecteur V_colonne2 est une matrice 3x1

La plus grande dimension d'un vecteur constitue sa longueur :

```
>> length(V_colonne1)
ans =
     3
>> length(V_ligne2)
ans =
     3
```

Il est possible de transposer un vecteur à l'aide de la fonction `transpose` ou avec l'apostrophe (').

```
>> V_colonne = V_ligne2'
V_colonne =
     0
     1
     2

>> V_colonne = transpose(V_ligne2)
V_colonne =

     0
     1
     2
```

Le double point (:) est l'opérateur d'incrémentation dans MATLAB. Ainsi, pour créer un vecteur ligne de valeurs de 0 à 1 par incrément de 0,2, il suffit d'utiliser (notez le nombre d'éléments du vecteur) :

```
>> V=[0 : 0.2 : 1]

V =

         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

On peut éviter de mettre les crochets si les composants d'un vecteur varient d'un pas constant :

```
>> V= 0 : 0.2 : 1
V =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

Si l'incrément est de 1, le pas n'est pas noté. On met le (:) uniquement entre le premier et le dernier élément

```
V = -2 : 5
V =
    -2    -1     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 Langage C, il n'y a pas d'indice 0 dans les vecteurs et matrices en MATLAB) :

```
>> a=V(2)
a =
    -1

>> V(3)=20*a

V =

    -2    -1   -20     1     2     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 (remarquez les messages d'erreur, ils sont parfois utiles pour corriger vos programmes) :

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

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

```
>> V1 = [1    2]; % vecteur 1x2
>> V2 = V1' ;     % vecteur 2x1
>> V = V1*V2    %  (1x2) * (2x1) = (1x1)
V =
5
```

```
>> V = V2*V1     %  (2x1) * (1x2) = (2x2)
V =
1          2
2          4

>> V1*V3  %  (1x2) * (1x3) = impossible
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

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
```

On peut aussi créer des matrices, par exemple,

```
>> V1 = [1   2];  % Vecteur ligne 1x2
>> V2 = [3   4];  % Vecteur ligne 1x2
>> V = [V1 ; V2] % Concaténation de deux vecteurs ligne en colonne = matrice 2x2
V =
1    2
3    4
```

qui n'est pas équivalent à :

```
>> V1 = [1 ; 2] ;    % vecteur colonne 2x1
>> V2 = [3 ; 4] ;  % vecteur colonne 2x1
>> V = [V1    V2]  % Concaténation de deux vecteurs colonnes en ligne = matrice 2x2
V =
1          3
2          4
```

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

```
>> V1 = [1     2];  % 1x2
>> V2 = [3   ;   4]; % 2x1
>> V= [V1 ; V2]     % impossible !
??? Error using ==> vertcat
All rows in the bracketed expression must have the same number of columns.
```

Les matrices peuvent aussi être construites directement de deux façons différentes:

```
>> M = [1    2 ;  3     4]    % construction ligne et colonne en une fois
M =
1        2
3        4
>> M1 = [1    2         % construction ligne par ligne en utilisant le retour chariot
3  4]

M1 =

     1     2
     3     4
```

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

```
>> m21 = M(2,1) % 2ème ligne, 1ère  colonne
m21 =
3
```

On peut aussi "compter" les éléments. MATLAB compte alors tous les éléments d'une ligne (de gauche à droite) avant d'accéder à la ligne suivante. Ainsi, dans la matrice 3x3 suivante :

```
>> A = [1  2  3; 4  5  6; 7  8  9]
A =
1   2   3
4   5   6
7   8   9
```

La valeur des éléments $a_{i,j}$ sont données par leur rang affecté par MATLAB. Le 5ème élément est 5 :

```
>> a5 = A(5)
a5 =
5
```

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

```
>> V = A(:,2) % ici, (:) signifie toutes les lignes => mettre toutes les lignes de
              % la colonne 2 de A dans V
V =
2
5
8
>> V=A(1:2, 3) % ici, (1:2) signifie les lignes 1 à 2 => mettre les lignes 1 à 2 de
la colonne 3 de A dans V
V =

     3
     6
```

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

```
>> M2= A(2:3 ,:) % (2:3) signifie lignes 2 à 3
                 % et (:) signifie toutes les colonnes
M2 =
4    5    6
7    8    9
```

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

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

>> inverseM = inv(M)
inverseM =
-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
>> E=A*inv(B) % la division de matrices est une multiplication par l'inverse : E=D
E =
    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
```

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    % 2 lignes
m =
    3    % 3 colonnes

>> dimM = length(M) % utilisation de length sur une matrice
dimM =
       3 % donne la plus grande dimension, ici 3
```

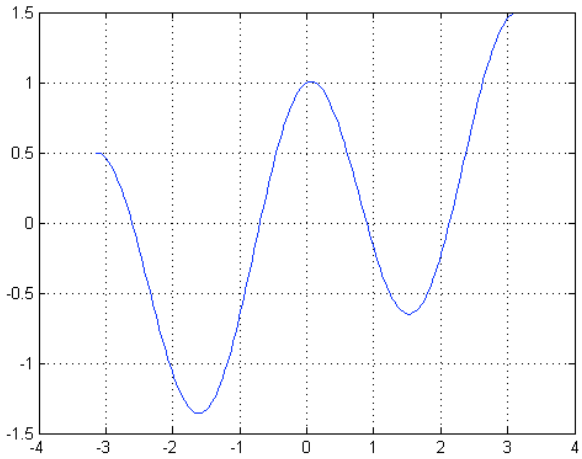
TP N° IV. Graphiques simples

MATLAB en plus de ses grandes possibilités de calcul numériques produit des graphiques en 2 ou 3 dimensions. On ne s'intéressera ici qu'au graphique 2D simple.
La fonction `plot(x,y)` permet de tracer une courbe liant un ensemble de valeurs (vecteur) `y` en fonction d'un une autre vecteur `x` (**bien entendu de même dimension).**
Dans l'exemple suivant on se propose de tracer la fonction suivante :

`y=cos(2x) +0.5 sin(0.5x).`
La variable x est un vecteur dont les valeurs vont de –π à +π avec un pas de π/100.

```
>> x = -pi: pi/100 : pi ; % Attention 'x' est en Radian
>> y = cos(2*x)+0.5*sin(0.5*x) ;  % Attention 'x' est en Radian
>> plot(x,y)
```

Ce qui produit la sortie graphique de la figure suivante :



Taper ensuite :

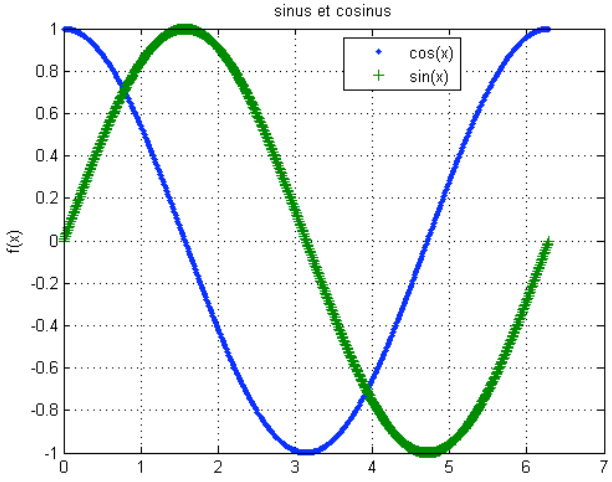
```
>> grid
>> xlabel('variable x')
>> ylabel('variable y')
>> title('y=cos(2x) +0.5 sin(0.5x)')
```

Que remarquez-vous ? Quelle est la fonction réalisée par chaque instruction ?

Pour plus de clarté, il faut nommer les axes, mettre les unités, proposer une légende… surtout quand il y a plusieurs courbes sur la même figure. Ceci est évidemment un peu long à écrire à l'invite MATLAB. Un SCRIPT (un fichier .m) est tout indiqué, comme le montre cet exemple (à taper dans un fichier .m) :

```
clear all
close all % ferme les anciennes figures
x = [0:0.01:2*pi];
y1 = cos(x); % Attention 'x' est en Radian
y2 = sin(x); % Attention 'x' est en Radian
figure(1) % ouvre une figure et la nomme figure(1) pour ensuite tracer la courbe
dessus.
plot(x, y1,'.',x, y2,'+') % cos(x) en points, sin(x) en +
grid;
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 plus de détail sur les fonctionnalités de chaque instruction, n'hésitez pas à consulter le



help (par exemple `help plot`)!

TP N° V. Fonction mathématiques simples

Les fonctions mathématiques usuelles sont listées dans le tableau ci-après :

Fonction	Description
sin(x)	sinus de x ; **x en radians**
cos(x)	cosinus de x ; **x en radians**
tan(x)	tangente de x ; **x en radians**
exp(x)	exponentielle de x
log(x)	logarithme (Ln) en base e de x
log10(x)	logarithme en base 10 de x
sqrt(x)	racine carr□e de x
power(x,a)	puissance a de x
abs(x)	valeur absolue de x
asin(x)	$\sin^{-1}$ de x ; **r□sultat en radians**
acos(x)	$\cos^{-1}$ de x ; **r□sultat en radians**
atan(x)	$\tan^{-1}$ de x ; **r□sultat en radians**
sinh(x)	sinus hyperbolique de x
cosh(x)	cosinus hyperbolique de x
tanh(x)	tangente hyperbolique de x
round(x)	arrondit un nombre □ l'entier le plus proche
floor(x)	arrondit vers l'entier imm□diatement au-dessous
ceil(x)	arrondit vers l'entier imm□diatement au-dessus

TP N° VI. Programmation avec MATLAB

Pour les étudiants familiers avec la programmation en langage C (que vous êtes !), il ne s'agit que de se familiariser avec la syntaxe propre à MATLAB.

I. Opérateurs logiques

Le type de variable retourné est de type Booléen `bool` (True = 1 ou False = 0)

Opérateur	Description
~a	NON logique sur a : retourne 1 si a = 0. Retourne 0 si a=1
a= =b	retourne 1 si a égal b, 0 autrement
a < b	retourne 1 si a est plus petit que b, 0 autrement
a > b	retourne 1 si a est plus grand que b, 0 autrement
a <= b	retourne 1 si a est plus petit ou égal à b, 0 autrement
a >= b	retourne 1 si a est plus grand ou égal à b, 0 autrement
a ~=b	retourne 1 si a est différent de b, 0 autrement

II. Boucle if-elseif-else :

```
si CONDITION1 vérifiée, FAIRE ACTION1.
sinon et si CONDITION2 vérifiée, 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;
END
```

Exemple à taper dans un script :

```
a= input('entrer un chiffre ' );
if a<0
phrase=['le chiffre '  num2str(a) ' est négatif'] ;
elseif a==0
phrase=['le chiffre ' num2str(a) ' est nul'];
else phrase= ['le chiffre ' num2str(a) ' est positif'] ;
end
disp (phrase)
```

III. Boucles for

```
incrément = valeur initiale  % initialisation
Pour incrément = valeur_initiale jusqu'à valeur finale  % faire :
ACTION1...N
AJOUTER 1 à l'incrément
```

En MATLAB, ce pseudo-code devient :

```
for i = 0 : valeur_finale  % valeur_finale doit être initialisée avant
ACTION1;
ACTION2;
…
ACTIONN;
end
```

Exemple : calcul d'un carré

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

```
clc , clear all ;
val_finale = 10 ;  % initialisation
for i=0 : 2 : val_finale
     i   % juste pour afficher i
    carre_i=i^2
end
```

IV. Boucle while

```
Tant que CONDITION est VRAIE  % faire
ACTION1..N
```

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

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

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 :

```
n = 0; % initialisation des valeurs
somme = 0;
while somme < 100
n=n+1; % itération de n
somme = somme + n; % nouvelle somme
end
n  % affiche n
somme  % affiche somme
```

V. Boucle Switch

```
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('Si vous voulez faire une Opération de type exp, tapez ''1'';\nSi
vous voulez faire une Opération de type log, tapez ''2'' : '); % /n pour revenir à
la ligne
% '' (double apostrophe) pour en afficher un seul
nombre = input('Valeur du nombre à opérer : ');
switch operation
case 1
b = exp(nombre)
case 2
b = log(nombre)
otherwise
disp('mauvais choix -- operation')
end
```

TP N° VII. MATLAB pour le traitement du signal

Un signal numérique est défini par un nombre d'échantillons N relevés à une fréquence d'échantillonnage Fe. Les signaux sont toujours captés de manière temporelle, mais on s'intéresse souvent à leur allure fréquentielle.

Taper le programme suivant dans un fichier .m.

```
clc, clear all, close all
%load exECG.mat
Fe = 1000;                      % Fréquence d'échantillonnage : 1000Hz
T = 1/Fe;                       % Période d'échantillonnage : 1ms
N = 1000;                       % longueur du signal = Nb de points du signal
numérique
t = (0:N-1)*T;                  % construction de vecteur temps (base de
temp)

% Construction du signal : Somme d'un sinus de 50Hz, d'un sinus de 120Hz et
d'une composante continue
y = 1 + 10*sin(2*pi*50*t) + 3*sin(2*pi*120*t)+ 2*sin(2*pi*200*t);

plot(1000*t(1:500),y(1:500))  % tracé d'une partie du signal pour voir son
allure. Le temps est multiplié par 1000 pour le mettre en millisecondes
```

```
title('Signal bruité avec un bruit aléatoire à moyenne nulle')
xlabel('Temps (millisecondes)')



%% Calcul de la FFT :
% Signal de sortie Y(f) = fft(signal y(t), Nb points N) : calcule la FFT
(en représentation bilatérale) d'un signal temporel y(t) sur
% un nombre de points N et met le résultat dans Y(f).
% Si le signal d'entrée comporte N échantillons, le Signal de sortie
comporte N valeurs complexes
% correspondant chacune à l'amplitude d'une composante de fréquence
déterminée. Ces fréquences sont
% espacées de delta_f = Fe/N. delta_f représente donc notre incrément (le
pas) fréquentiel.
% La première valeur du signal de sortie correspond à la fréquence f=0 (la
composante continue du
% signal d'entrée), la ieme valeur du signal de sortie correspond à la
fréquence (i-1)*delta_f.


%% Application 1 : Représentation bilatérale par rapport à la Fréquence de
%% Nyquist Fn=Fe/2

FFTy= fft(y,N)/N; % calcul de la fft (représentation bilatérale) du signal
y et placement du résultat dans FFTy.
% Observer le résultat dans la fenêtre de commande. FFTy est bien complexe.

figure ; plot (abs(FFTy)); % Tracé du module de FFTy (la FFT de y(t)) en
représentation bilatérale en fonction de N.
title('Spectre en amplitude de y(t) en représentation bilatérale')
xlabel('indice des points')
ylabel('|Y(f)|')

% Pour tracer Y(f)=FFTy, le spectre du signal y(t), en fonction des
fréquences, l'axe fréquentiel qui contient N points (les abscisses)
% doit être gradué de 0 à (N-1)*delta_f avec un pas de delta_f. Ce qui
permettra donc de tracer le spectre entre [0 et
% Fe[.

% Pour retrouver l'amplitude des composantes fréquentielles  il faut
diviser le
% module de FFTy par N (voir formule de la FFT).
% Les amplitudes des composantes fréquentielle en représentation bilatérale
% (hors composante continue), comparées aux amplitudes temporelles des
signaux,
% sont divisées par 2 car la FFT répartit l'énergie du signal de part et
d'autre du spectre.

% Création de l'axe fréquentiel (les abscisses) pour la représentation
bilatérale en symétrie par rapport la Fe/2 :
delta_f= Fe/N ;
axe_freq= 0 : delta_f : (N-1)*delta_f ;


figure ; plot (axe_freq, abs(FFTy)); % Tracé du module de FFTy (la FFT de
y) en représentation bilatérale en fonction des fréquences.
```

```
title('Spectre en amplitude de y(t) en représentation bilatérale : symétrie
par rapport à Fe/2')
xlabel('Fréquence (Hz)')
ylabel('|Y(f)|')

% Remarque : En représentation bilatérale, il y a symétrie horizontale par
% rapport à la fréquence de Nyquist Fe/2 (indice N/2). Les valeurs au-delà
de cet indice N/2 correspondent aux fréquences négatives du
% spectre de fréquence.


%% Application 2 : la FFT inverse avec ifft()
%La fonction ifft() permet de calculer le signal temporel (Réel) à partir
du spectre du signal complexe en représentation
%bilatérale.
% Y(f)= FFT(y(t)) = FFTy
% y(t)= FFT inv (Y(f)) = FFTinv_y
FFTinv_y = N* ifft(FFTy) ; % comme on a divisé précédemment par N, on doit
multiplier ici aussi par N
figure,  plot (FFTinv_y,'b') % tracé du signal résultat
figure,  plot ( y, 'r') % comparaison avec le signal originel y(t)


%% Application 3 : Représentation bilatérale par rapport à f=0

% Pour placer la composante correspondant à f=0 (c-à-d, la composante
% continue) au milieu du spectre de fréquence, on utilise la fonction
% fftshift :
figure ; plot (fftshift(abs(FFTy)));
title('Spectre en amplitude de y(t) en représentation bilatérale : symétrie
par rapport à la composante continue')
xlabel('indice des points')
ylabel('|Y(f)|')

% On obtient donc une représentation bilatérale avec
% symétrie par rapport à la composante continue.
% Dans ce cas, contrairement à l'application 1, l'axe des fréquences (qui
contient N
% points) correspondant à cette représentation doit être gradué de
(N/2)*delta_f à ((N/2)-1)*delta_f. Ce qui permet de tracer le spectre entre
[-Fe/2 et Fe/2[

axe_freq2= (-N/2)*delta_f : delta_f : ((N/2)-(1))* delta_f ;
% ou axe_freq2=(-Fe/2):(delta_f):(Fe/2)-(delta_f);
figure ; plot (axe_freq2,fftshift(abs(FFTy)));
title('Spectre en amplitude de y(t) en représentation bilatérale : symétrie
par rapport à la composante continue')
xlabel('Fréquence (Hz)')
ylabel('|Y(f)|')

% Nous obtenons bien une représentation bilatérale avec symétrie par
% rapport à f=0.

%% Application 4: passage de la représentation bilatérale à la
%% représentation monolatérale
% En représentation monolatérale, on doit tracer le spectre su signal
% uniquement sur (N/2)+1 points de fréquence, avec  f = [0  Fe/2]
% Se pose le problème ici, si N est paire ou impaire ! pour résoudre ce
% problème on utilise la fonction d'arrondi vers le bas (arrondi par
défaut) : "floor"
```

```
axe_freq3= 0 : delta_f : ((floor(N/2)))* delta_f ;


% La fonction fft() permet de calculer le spectre en représentation
bilatérale. Comme dit précédemment, les amplitudes des composantes
% en représentation bilatérale sont ( hors composante continue) divisées
par 2.
% La représentation monolatérale serait plus correcte et attribut à chaque
% fréquences l'énergie du signal lui correspondant.
% Pour passer de la représentation bilatérale à la représentation
monolatérale, il faut donc multiplier par
% 2 les amplitudes fréquentielles de toutes les composantes de la FFT
% bilatérale SAUF celle de la composante continue. Il faut également
% garder seulement les (N/2)+1 premières composantes de la FFT.
FFT_y_monolaterale= [ FFTy(1)   2*FFTy(2:(floor(N/2))+1)  ] ; % On garde la
composante continue telle qu'elle et on x2 le reste.

% Plot du spectre du signal d'entrée en représentation monolatérale :
figure;
plot(axe_freq3,abs(FFT_y_monolaterale))

title('Spectre en amplitude de y(t) en représentation monolatérale')
xlabel('Fréquence (Hz)')
ylabel('|Y(f)|')


%% Application 5: Filtrage de fréquence passe bas par fenêtrage
% objectif : garder la CC et le signal de plus basse fréquence (50Hz). Pour
% ce faire repérer les indices des pics correspondants à ces composantes.
% Construire une fenêtre dont l'amplitude vaut 0 pour toutes les fréquences
% qu'on veut supprimer et 1 pour les autres.
% En pratique, réaliser la fftshift sur le spectre de votre signal et
multiplier le par une fenêtre qui doit valoir 1 entre les indices 400 et
600 et 0
% ailleurs.

% Création de la fenêtre
fenetre_carree = zeros(1, N); % initialisation de la fenêtre à 0
fenetre_carree(400:600) = 1;
figure, plot(fenetre_carree);

FFT_y = fftshift(fft(y));
FFT_y_filtre= FFT_y .* fenetre_carree ;

y_filtre= ifft(ifftshift(FFT_y_filtre)) ;
%
figure , clf, hold on,  plot(y, 'r') %% tracé du signal original et le
signal filtré.
plot(real(y_filtre), 'b')
% L'utilisation de REAL (partie réelle) est des fois nécessaire car la
% la combinaison du fenêtrage et de la fonction FFT inverse peut donner un
signal résultant complexe.
% Remarque : dans les application de filtrage, il est généralement pas
% nécessaire de /N la FFT du signal (comme vu dans l'application 1) et donc
% de x N la FFT inverse.

% On peut vérifier le résulat en fréquence (n'oublier pas de diviser par N
pour avoir les bonnes amplitudes.
figure , plot (abs(fftshift(fft(y_filtre)))/N)
```