Ministry of Higher Education and Scientific Research
Badji Mokhtar Annaba University
Faculty of Technology

وزارة التعليم العالي والبحث العلمي
جـامعة باجـي مختـار- عنابـة
كليـة الــتكنـولوجيا

Département : **Electromécanique**

Polycopié pédagogique

Dossier numéro (à remplir par l'administration) **:** ……

Titre

PROGRAMMATION MATLAB

Cours destiné aux étudiants de

Master 1 : Hygiène et sécurité industrielle

Année : Janvier 2023

Table Des Matières

PARTIE 1 : Débuter avec Matlab

1) Introduction :

MATLAB dont le nom provient de **MAT**RIX **LAB**ORATORY est un langage de calcul scientifique basé sur le type de variables matricielles à syntaxe simple. Avec ses fonctions spécialisées, et ses grandes capacités de calcul numérique MATLAB peut être aussi considéré comme un langage de programmation adapté aux problèmes scientifiques.

MATLAB est constitue d'un noyau relativement réduit, capable d'interpréter puis d'évaluer les expressions numériques matricielles qui lui sont adressées :

- soit directement au clavier depuis la fenêtre de commande : c'est le mode interactif ;

- soit sous forme de séquences d'expressions ou scripts enregistrées dans des fichiers textes appelés m-fiches (programme en langage MATLAB) et exécutées depuis la fenêtre de commande : c'est le mode exécutif. Les programmes peuvent être simples ou comporter des fonctions avec des paramètres d'appel et de retour. Les fonctions sont très utiles dans la mesure où chaque utilisateur peut étendre les possibilités de MATLAB à son domaine d'application.

- soit plus rarement sous forme de fichiers binaires appelés mex.files (fichiers.mex) générés à partir d'un compilateur C ou de fortran.

2) La structure de MATLAB :

Le noyau de MATLAB est complété par une bibliothèque de fonctions prédéfinies, très souvent sous forme de fichiers « m-files », et regroupés en paquetages ou toolboxes. A côté des toolboxes requises local et matlab, il est possible d'ajouter des toolboxes spécifiques à tel ou tel problème mathématique, Optimization Toolbox, Signal Processing Toolbox par exemple ou encore des toolboxes crées par l'utilisateur lui-même. Un système de chemin d'accès ou path permet de préciser la liste des répertoires dans lesquels MATLAB trouvera les différents fichiers m-files.

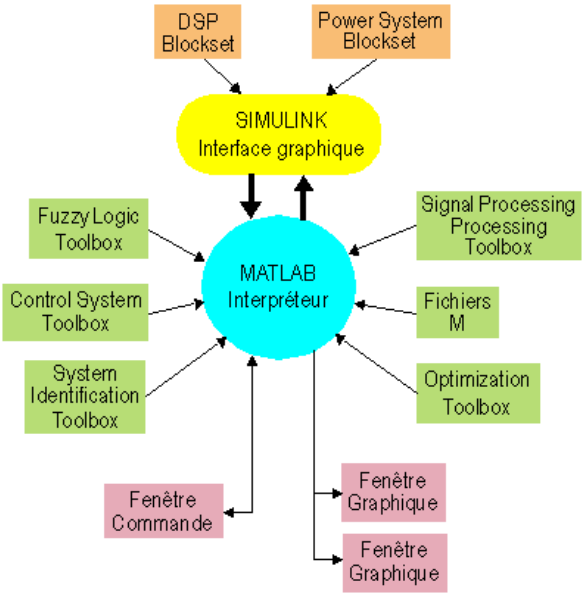


Figure 1. Environnement MATLAB

Fenêtre Commande : Dans cette fenêtre, l'utilisateur donne les instructions et MATLAB retourne les résultats.

Fenêtres Graphique : MATLAB trace les graphiques dans ces fenêtres.

Fichiers.m : Ce sont des programmes en langage MATLAB (écrits par l'utimisateur).

Toolboxes : Ce sont des boites à outils composées des collections de fichiers m-files développés pour des domaines d'application spécifiques (Signal Processing Toolbox, System Identification Toolbox, Control System Toolbox, Robust Control Toolbox, Optimization Toolbox, Neural Network Toolbox, Fuzzy Logic Toolbox, etc.)

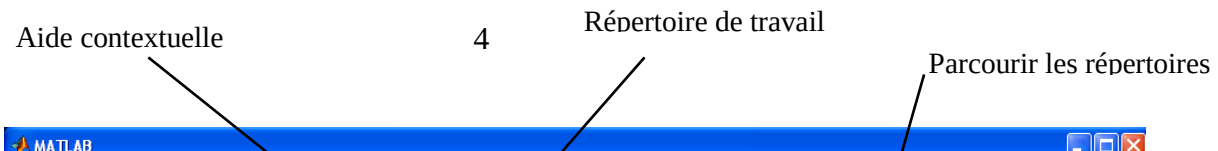
Simulink : C'est l'extension graphique de MATLAB permettant de travailler avec des diagrammes en blocs.

Blocksets : Ce sont des collections de blocs Simulink développés pour des domaines d'application spécifiques.

3) Une session de travail MATLAB

3.1) DÉMARRER MATLAB

- En exécutant le programme matlab, celui-ci répondra par le symbole suivant >> dans l'espace de travail.

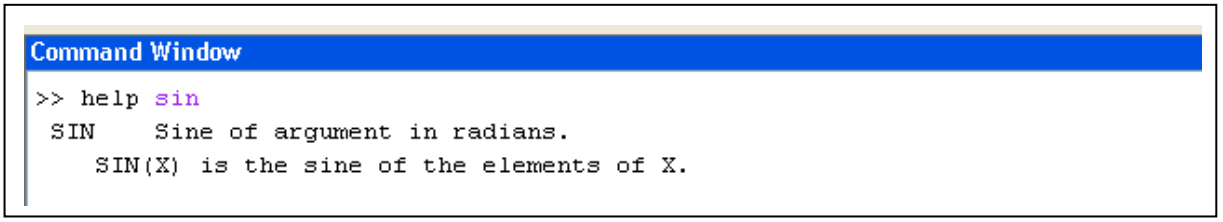


- Dans cette fenêtre Commande, on tape les instructions une ligne à la fois.
- Chaque ligne est exécutée immédiatement après la touche "Entrée".
- Une ligne peut contenir plusieurs instruction séparées par des virgules (,).
- Des boucles FOR, WHILE, IF ... ELSE peuvent être sur plusieurs lignes.

3.2) FONCTION "HELP"

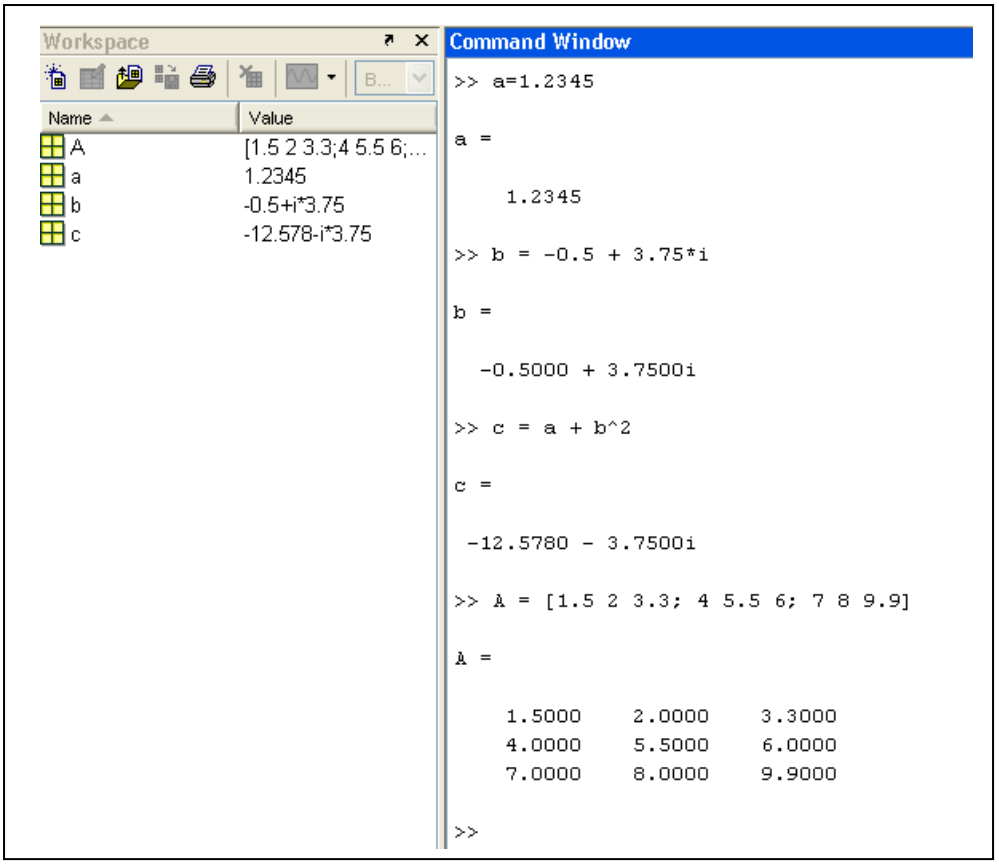
Pour obtenir de l'aide sur un sujet, une instruction ou une fonction, on tape help suivi par le sujet, l'instruction ou la fonction désirée.

Exemple 1 :



3.3) ESPACE DE TRAVAIL (Workspace)

Les variables sont définies au fur et à mesure que l'on donne leurs noms et leurs valeurs numériques ou leurs expressions mathématiques. Les variables ainsi définies sont stockées dans l'espace de travail et peuvent être utilisées dans les calculs subséquents.

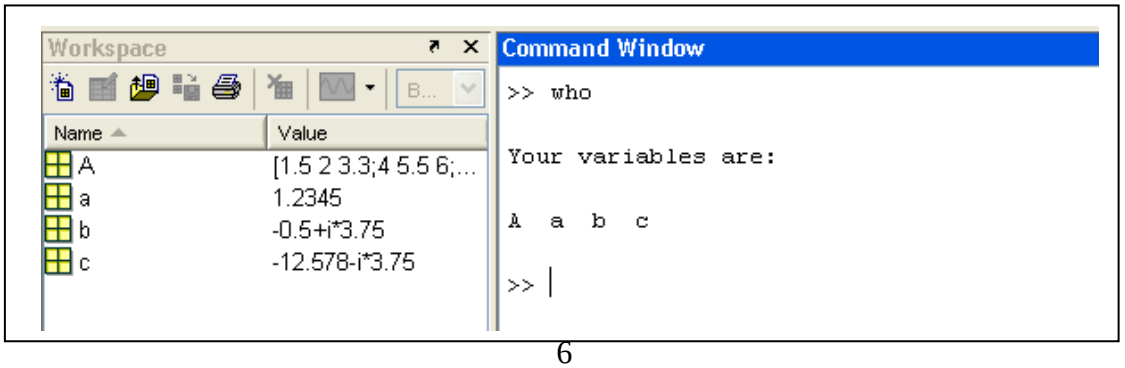


3.4) INFORMATION SUR L'ESPACE DE TRAVAIL

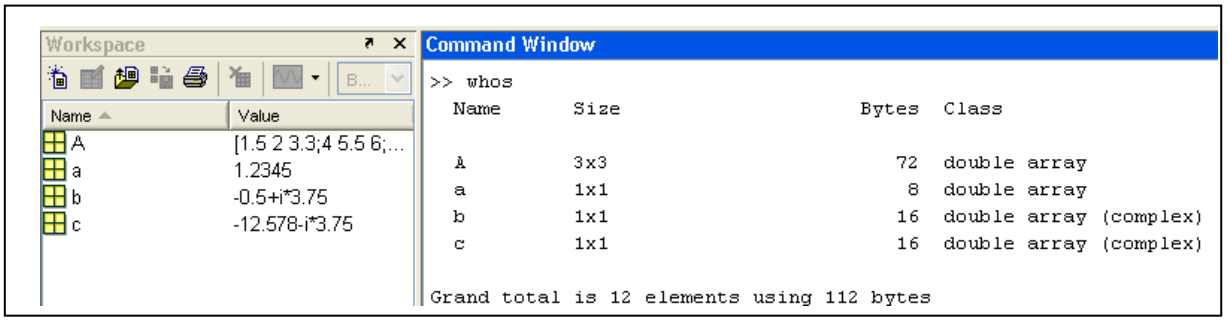
Pour obtenir une liste des variables dans l'espace de travail, on utilise les instructions suivantes

- who : permet l'affichage des variables dans l'espace de travail.
- whos : permet l'affichage **détaillé** des variables dans l'espace de travail.

Instruction who :



Instruction whos :



3.5) ENREGISTRER LES VARIABLES DE L'ESPACE DE TRAVAIL DANS UN FICHIER

Pour enregistrer les variables de l'espace de travail dans un fichier, on utilise les instructions suivantes :

- **save** permet l'enregistrement de toutes les variables dans un fichier matlab.mat.

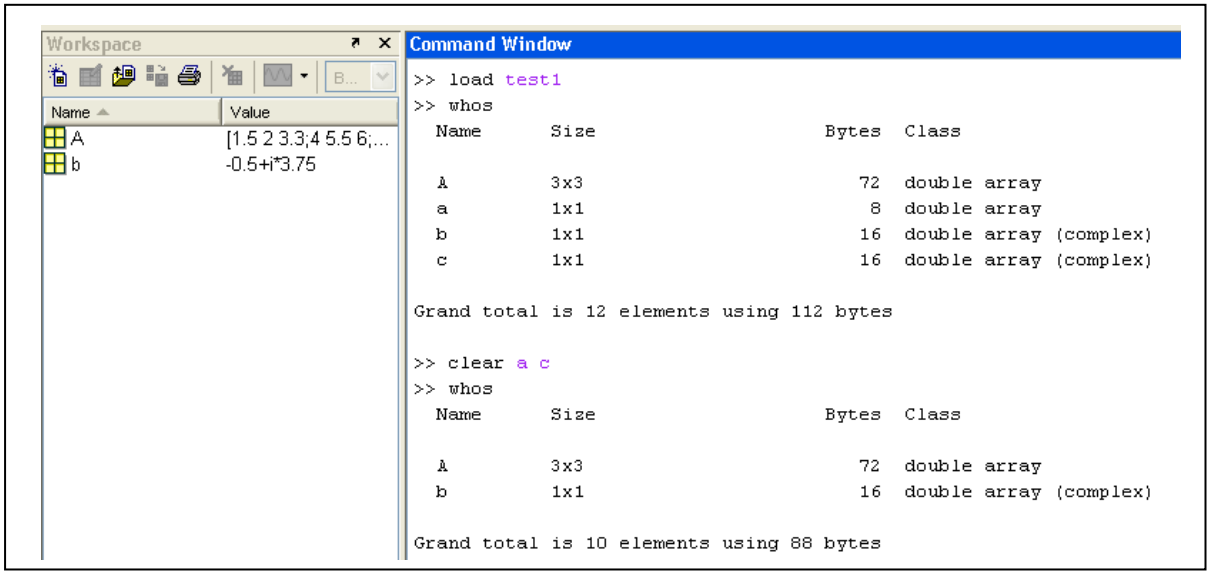
  Dans une session ultérieure, taper load pour ramener l'espace de travail enregistrée.

- **save fichier1.mat A a b c** permet l'enregistrement des variables A, a, b, c dans le fichier fichier1.mat.

  Dans une session ultérieure, taper **load fichier1** pour ramener les variables A, a, b, c dans l'espace de travail.

3.6) Quelques commandes et constantes souvent utilisées :

- La commande CLEAR permet la suppression d'une variable dans l'espace de travail.



- La commande CLC : permet d'effacer la fenêtre de commande.
- La commande CLOSE : permet la fermeture d'une fenêtre « figure ».
- La constante Pi, le plus grand réel et le plus petit réel représentés sous MATLAB, la constante INF et EPS.

```
>> pi
ans =
      3.1416

>> realmax
ans =
      1.7977e+308

>> realmin
ans =
      2.2251e-308

>> inf
ans =
       Inf

>> eps
ans =
      2.2204e-016

>> NaN   'Not-a-Number' est généré en tentant d'évaluer
des expressions comme 0/0 ou bien Inf/Inf
```

4) Calculer avec Matlab

4.1) Les nombres et tableaux de nombres

Les nombres réels (ou entiers) sont entrés sous les formes décimales ou scientifiques usuelles, les nombres complexes sont écrits sous la forme *a* + *bi*, comme dans 1+2i. Les tableaux de nombres réels ou complexes de dimension un ou deux suivent la syntaxe suivante :

- un tableau est délimité par des crochets ;
- les éléments sont entrés ligne par ligne ;
- les éléments appartenant à la même ligne sont séparés par des espaces (ou par des virgules) ;
- les différentes lignes doivent comporter le même nombre d'éléments et sont séparées par des points-virgules.

Exemple : 1 2 3 4 s'écrit sous la forme : [1 2 3 4]

Et 1 s'écrit sous la forme : [1 **;** 2 **;** 3 **;** 4]

2

3

4

Matlab utilise par convention le point « **.** » comme notation décimale. On peut définir aussi les nombres à l'aide de la notation scientifique. Elle utilise la lettre e pour spécifier le facteur de puissance de 10. Voici quelques exemples de nombres

3 -99 0.0001

1.60210e-20 -3.14159j

Tous les nombres sont stockés de manière interne en utilisant le format long. La précision est donc de 16 chiffres après la virgule.

4.2) Les caractères et chaînes de caractères

On écrit les caractères et les chaînes de caractères entre apostrophe **′** a **′**, **′**licence**′,** matlab considère les caractères comme des chaînes de caractères de longueur un (1) et identifie chaînes de caractères et liste de caractères.

Exemple : Le tableau de caractères [′a′ ′b′ ′c′ ′d′] est identique à la chaîne de caractères [′a b c d′].

Le tableau de caractères [′a b c′ ′d e′] est identique à la chaîne de caractères [′a b c d e′]

4.3) Cellules et tableaux de cellules

Une cellule est un conteneur dans lequel on peut placer toute sorte d'objets : nombre, chaîne de caractères, tableau et même tableau de cellules. Les tableaux de cellules permettent d'associer dans une même structure des éléments de nature très différente. La syntaxe des tableaux de cellules est voisine de celle des tableaux usuels, les crochets étant remplacés par des accolades.

Exemple :

```
>> { ′alpha′, 6 ; ′beta′, 4 }
ans =
        ′alpha′ [6]
        ′beta′  [4]
```

La manipulation des ces objets (sauf lorsqu'on on se limite à des composants qui sont des nombres ou des chaînes de caractères) est un peu plus délicate que celle des tableaux usuels et sera examinée dans un prochain chapitre.

4.4) Les variables

Une caractéristique de matlab est qu'il n'est pas nécessaire de déclarer les variables ou bien encore leurs dimensions. Quand Matlab rencontre un nouveau nom de variable, il crée automatiquement cette variable et alloue la place mémoire nécessaire à sons stockage. Si la variable existe déjà, Matlab change son contenu et si nécessaire réalloue une nouvelle place pour le stockage. Le symbole d'affectation de valeur à une variable est le caractère =

Exemple : alpha = 25 crée une matrice 1 × 1 de nom alpha et stocke la valeur 25 dans cet élément.

Les règles de dénomination des variables sont très classiques :

- un identificateur débute par une lettre, suivie d'un nombre quelconque de lettres ou de chiffres ;
- sa longueur est inférieure ou égale à 31 caractères;
- les majuscules sont distinctes des minuscules. Ainsi B et b ne désignent pas la même variable.

Pour connaître le contenu d'une variable, il suffit simplement de taper son nom suivi d'un retour chariot.

Remarque :

Le nom des constantes n'est pas réservé, c'est-à-dire qu'il est possible de définir des variables de même nom. Dans ce cas, l'identificateur fera référence à la variable définie par l'utilisateur et non plus à la constante matlab. On fera attention par exemple, si l'on utilise le type complexe, à ne pas écrire de boucles ayant i ou j comme indices. Pour que l'identificateur fasse à nouveau référence à la constante matlab, il suffit de supprimer la variable de même nom de la mémoire par la commande clear.

Ex **i** : $\sqrt{-1}$ **j** : $\sqrt{-1}$

4.5) Les opérateurs

Les expressions utilisent les opérateurs arithmétiques classiques ainsi que leurs règles usuelles

+	Addition
-	Soustraction
*	Multiplication
/	Division
\	division à gauche (pour les matrices)
^	Puissance
'	transposition et conjugaison complexe
( )	spécifie l'ordre d'évaluation
~	Opérateur de négation

4.6) Opérations et fonctions portant sur les scalaires :

Il est bien entendu possible d'utiliser matlab pour faire de simples additions. (Si x et y sont deux variables scalaires de type réel, x + y, x - y, x * y et x / y désignent les 4 opérations usuelles entre les valeurs de x et y dans l'ensemble R. Si x et y sont deux variables scalaires de type complexe, x + y, x - y, x * y et x / y désignent les 4 opérations usuelles entre

les valeurs de x et y dans l'ensemble C. L'exponentiation s'obtient grâce au symbole ^ (la syntaxe est x ^ y).

Les fonctions mathématiques incorporées sont :

log(x) :	logarithme népérien de x,
log10(x) :	logarithme en base 10 de x,
exp(x) :	exponentielle de x,
sqrt(x) :	racine carrée de x (s'obtient aussi par x.^0.5),
abs(x) :	valeur absolue de x,
sign(x):	fonction valant 1 si x est strictement positif, 0 si x est nul et -1 si x est strictement négatif.

Lorsque la fonction est définie sur C, l'argument peut être de type complexe. On dispose également de fonctions spécifiques aux complexes :

conj(z) :	le conjugué de z,
abs(z) :	le module de z,
angle(z):	argument de z,
real(z) :	partie réelle de z,
imag(z) :	partie imaginaire de z.

Les fonctions d'arrondis sont :

round(x) :	entier le plus proche de x,
floor(x) :	arrondi par défaut,
ceil(x) :	arrondi par excès,
fix(x) :	arrondi par défaut un réel positif et par excès un réel négatif.

Les fonctions trigonométriques et hyperboliques sont :

cos :	cosinus,
acos :	cosinus inverse (arccos),
sin :	sinus,
asin :	sinus inverse (arcsin),
tan :	tangente,
atan :	tangente inverse (arctan),
cosh :	cosinus hyperbolique (ch),
acosh :	cosinus hyperbolique inverse (argch),

sinh : sinus hyperbolique (sh),

asinh : sinus hyperbolique inverse (argsh),

tanh : tangente hyperbolique (th),

atanh : tangente hyperbolique inverse (argth).

4.7) Opérateurs booléens

- Les opérateurs de comparaison :
  - Supérieur >
  - Inférieur <
  - Egalité = =
  - Non égalité ~=
- Les opérateurs logiques
  - et logique &
  - ou logique |

4.8) Résumé

Voici un résumé des opérations possibles

Notation mathématique	Commande Matlab
a + b	a + b
a - b	a - b
ab	a * b
a/b	a / b ou bien b \ a
x^b	x^b
$\sqrt{x}$	Sqrt(x) ou bien x^0.5
$\|x\|$	Abs(x)
π	Pi
$4{\cdot}10^3$	4e3 ou bien 4 * 10^3
3-4i	3 - 4* i ou bien 3 – 4 * j
e, e^x	exp(1), exp(x)
ln x, log x	log(x), log10(x)
sin x, arctan x	sin(x), atan(x)

5) Manipulation des Vecteurs

Un vecteur est un tableau particulier qui ne comporte qu'une seule ligne ou bien une seule colonne. MATLAB propose un certain nombre de fonctions qui simplifient l'usage des vecteurs

5.1) Construction des vecteurs lignes

Ainsi que nous l'avons déjà vu, on peut définir la valeur d'un vecteur en donnant la suite de ses éléments séparés par des espaces, la liste étant délimitée par des crochets :

Exemple :

```
>> v1 = [1  2  3  4  5]

v1 =

     1    2   3  4  5
```

5.2) L'opérateur deux-points « : »

L'opérateur deux-points est l'un des plus important en Matlab. Il est utilisé sous différentes formes. L'expression **1 : 10** est un vecteur ligne contenant les entiers entre 1 et 10

```
>> 1 : 10
ans =
     1     2     3     4     5     6     7     8     9     10
```

Pour obtenir un espacement (un pas) autre que un (1), il suffit de spécifier l'incrément.

Exemple :

```
>> 100 : -7 : 50
ans =
     100    93    86    79    72    65    58    51
```

L'expression $v_i : p : v_f$ crée un vecteur dont les éléments constituent une progression arithmétique de valeur initiale v_i, de pas p et dont tous les termes sont inférieurs ou égaux à la valeur finale v_f. Lorsque la valeur du pas est omise $v_i : v_f$ la valeur du pas est fixée par défaut à un (1).

Exemples :

```
>> v2 = 1 : 5
v2 =
     1   2   3   4  5

>> v3 = 1 : 5.5
v3 =
     1   2   3   4  5
```

Comme le pas n'est pas spécifié l'incrémentation s'arrête à 5 (puisque 5.0000 + 1 est strictement supérieur à 5.5)

```
>> v4 = 1.5 : 0.3 : 2.5
v3 =
     1.5000   1.8000   2.1000   2.4000          puisque  (2.4000 + 0.3 > 2.5000)
```

5.3) Fonction linspace

La fonction linspace(v_i, v_f, n) crée une liste de n éléments uniformément répartis entre v_i et v_f, linspace(v_i, v_f, n) est équivalent à $v_i : \frac{v_f - v_i}{n-1} : v_f$

Exemple :

```
>> v5 = linspace(0.5 , 2 , 4)
v5 =
     0.5   1.0   1.5   2.0
```

5.4) Construction des vecteurs colonnes

On peut définir la valeur d'un vecteur colonne en tapant entre deux crochets la suite de ces éléments séparés par des points virgules.

Exemple

```
>> v6 = [1 ; 2 ; 3]
v6=
     1
     2
     3
```

5.5) Transposition

La transposée d'un vecteur ligne est un vecteur colonne, on peut donc utiliser les expressions et les fonctions pour les vecteurs lignes en les composant avec une transposition (opérateurs ′ ou .′ pour des vecteurs de nombres complexes)

Exemples

```
>> v7 = [1  2  3]′
v7 =
        1
        2
        3
>> v8 = [1+i    2+2i     3+3i].′
v8=
        1 + 1.000 i
        2 + 2.000 i
        3+  3.000 i
```

5.6) La fonction size

La fonction size retourne le couple (nl , nc) formé du nombre de lignes nl, et du nombre de colonnes nc du vecteur.

Exemple

```
>>  a = [1  2  3  4]
>>  size(a)
ans =
        1   4
```

Pour accéder plus facilement au nombre de lignes et au nombre de colonnes, on peut affecter la valeur retournée par size à un tableau à deux éléments [nl, nc] :

Exemple

```
>>  a = [1  2  3  4]
>>  [nl , nc] = size(a)
nl =
        1
```

```
nc =
     4
```

5.7) Nombre d'éléments d'un vecteur (length)

Le nombre de ligne d'un vecteur ligne est bien évidemment un, et le nombre de colonne pour un vecteur colonne est un. Pour les vecteurs on utilise de préférence la fonction length qui retourne le nombre d'éléments ou la longueur du vecteur.

Exemple

```
>> b = [2  4  6  8  10] ; length (b)
ans =
     5
>> c = [5  10  15   20]' ; length (c)
ans =
     4
```

5.8) Accès aux éléments d'un vecteur

Soient v un vecteur non-vide, et *k* un entier compris entre 1 et la longueur du vecteur considéré ($1 \leq k \leq$ *length*(v)). On accède à l'élément d'indice *k* du vecteur v par v(k), le premier élément du vecteur étant indicé par 1.

Exemple

```
>> v = [4  11  9  7  12] ; v(1)
ans =
     4
>> v(4)
ans =
     7
```

Remarque 1: L'accès en lecture à un élément d'indice négatif ou dont la valeur est strictement supérieure à la longueur du vecteur, conduit à une erreur.

Pour l'exemple précédent

```
>> v = [4  5  6  7  2] ;
>> v(8)
??? Index exceeds matrix dimensions.
```

Remarque 2:

Il est possible d'affecter une valeur à un élément d'un vecteur dont l'indice dépasse la longueur du vecteur. Comme le montre l'exemple suivant, les éléments dont l'indice est compris entre la longueur du vecteur et l'indice donné sont affectés de la valeur 0. La longueur du vecteur est alors modifiée en conséquence.

Exemple toujours avec l'exemple précédent

```
>>  v = [4  11  9   7  12] ;  length(v)
ans =
        5
>>  v(8) = 3 ; v
v =
        4   11   9   7   12   0   0   3
>>  length(v)
ans =
        8
```

6) Manipulation des Matrices

6.1) Saisie de matrices

La meilleure manière de débuter avec Matlab est d'apprendre comment manipuler les matrices. Il est possible de saisir des matrices de différentes manières

- entrer une liste explicite d'arguments
- charger une matrice depuis un fichier externe
- générer des matrices avec des fonctions Matlab
- créer une matrice avec des M-fichiers

Nous commençons par saisir la matrice comme une liste de ses éléments. Il faut suivre les règles suivantes :

- séparer les éléments d'une même ligne par des espaces ou des virgules
- utiliser le point virgule pour indiquer la fin d'une ligne
- encadrer toute la liste des éléments par des crochets [ ].

En suivant les principes précédents, la saisie de la matrice carrée se fait en tapant dans la ligne de commande :

```
A = [1   2   3   4 ;  5   6   7   8 ;  9  10   11  12 ;  13   14   15   16]
```

Matlab affiche alors la matrice

A =

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Une fois que la matrice a été saisie, elle est automatiquement stockée dans l'espace de travail sous forme de variable A. On peut donc maintenant se référer à cette matrice par A.

6.2) Somme, diagonale et transposée

- **Somme :** Si on veut obtenir la somme des lignes, on utilise la syntaxe sum(A). Reprenons la matrice A et vérifions cela avec Matlab .

  sum(A)

  ans =

  28 32 36 40

  Quand on ne spécifie pas de variable pour le résultat, Matlab utilise la variable par défaut ans (answer = résultat). On vient de calculer un vecteur ligne qui contient la somme des colonnes de A. Pour faire la même opération sur les lignes, il suffit de transposer la matrice.

- **Diagonale :**

  diag(A)' donne avec matlab

  >> 1 6 11 16

  La somme des éléments de la diagonale principale avec les commandes sum et diag

  sum(diag(A)') donne

  >> 34

- **Transposée :** si on veut obtenir la transposée de la matrice A on utilise la syntaxe **A′**

  >> **A′**

  ans =

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

6.3) Les indices

L'élément en ligne i et en colonne j de A est désigné par A(i , j). Par exemple l'élément A (4 , 2) correspond au nombre situé en quatrième ligne et deuxième colonne. Pour la matrice précédente, A(4 , 2) est 14.

Il est donc possible de calculer la somme des éléments dans la quatrième colonne par la commande

```
>> A(1 , 4) + A(2 , 4) + A(3 , 4) + A(4 , 4)
   ans =
          40
```

Cette méthode d'identification des éléments d'une matrice est moins élégante. Il est aussi possible de désigner des éléments d'une matrice à l'aide d'un seul indice A(k). Dans le cas d'un vecteur, cette manière de noter n'a rien de choquant. Dans le cas d'une matrice, elle peut paraître plus étrange. Dans ce cas, la matrice est considérée comme un long vecteur colonne formé à partir des autres colonnes de la matrice originale.

Pour la matrice citée précédemment l'élément A(8) est une autre manière de référencer la valeur 14 stockée dans A(4,2).

Remarque 1 : Si l'on tente d'utiliser un élément en dehors de la matrice, on obtient une erreur

```
>>  t = A(4,5)
    ??? Index exceeds matrix dimensions.
```

D'un autre coté, si l'on stocke un élément en dehors de la matrice, sa taille est automatiquement augmentée pour accommoder la nouvelle matrice

```
>>  B = A;
>>  B (4 , 5) = 17

    B =
          1     2     3     4     0
          5     6     7     8     0
          9    10    11    12     0
         13    14    15    16    17
```

Remarque 2 : Les indices peuvent aussi utiliser l'opérateur « **:** ». Il devient alors très simple de désigner des portions de matrices. Par exemple, B(1 **:** k , j) désigne les k premiers éléments de la jème colonne de X.

Ainsi, sum(B(1 : 4 , 4)) calcule la somme de la 4ème colonne.

Mais, il y a encore mieux. Les **:** par eux mêmes désignent tous les éléments d'une ligne ou d'une colonne d'une matrice. Le mot clé end désigne la dernière ligne ou dernière colonne. Par exemple, sum(A(**:** , end)) calcule la somme des éléments dans la dernière colonne de A. Si les entiers de 1 à 16 sont rangés dans quatre groupes de sommes égales, alors, cette somme doit être

```
>> sum(B( : , 4)
    ans =
          40
```

6.4) Concaténation de matrices

L'opérateur [ ] permet la concaténation de matrices (tableaux) :

- Si les matrices m_k possèdent le même nombre de lignes l'expression [m_1 **,** m_2 **,** m_3 **,** …**,** m_p] crée une matrice :
  - qui a le même nombre de lignes que les matrices composantes
  - dont le nombre de colonnes est la somme des nombres des colonnes de chacun des matrices composantes
  - qui est obtenu en concaténant « **à droite** » les matrices composantes

  On peut dans l'expression ci-dessus remplacer les virgules par des espaces

- Si les matrices m_k possèdent le même nombre de colonnes l'expression [m_1 **;** m_2 **;** m_3 **;** …**;** m_p] crée une matrice :
  - qui a le même nombre de colonnes que les matrices composantes
  - dont le nombre de lignes est la somme des nombres des lignes de chacun des matrices composantes
  - qui est obtenu en concaténant « les uns sous les autres » les matrices composantes

Exemple :

```
>> m1 = [1  2 ; 5  6]
```

```
m1 =
     1     2
     5     6

>> m2 = [3  4 ; 7  8]

m2 =
     3     4
     7     8

>> m3 = [ m1  , m2]      % ou bien  m3 = [ m1   m2]

m3 =
     1     2     3     4
     5     6     7     8

>> m4 = [ m1 ; m2]
m =
     1     2
     5     6
     3     4
     7     8
```

6.5) Matrices particulières

Les fonctions ci-dessous permettent de construire des matrices correspondants aux matrices usuelles : identité, matrice nulle, …

- **eye (n) :** La fonction eye (n) construit une matrice carrée de diagonale égale à 1

  Exemple :

```
>> A= eye (3)
   A =
        1     0     0
        0     1     0
        0     0     1
```

- **ones(n,m) :** La fonction ones (n , m) construit une matrice formée de n lignes et m colonnes dont tous ces éléments sont égaux à un.

  Exemple :

  ```
  >> B= ones (3 , 4)
       1     1     1     1
       1     1     1     1
       1     1     1     1
  ```

- **zeros(n,m) :** La fonction zeros (n , m) construit une matrice formée de n lignes et m colonnes dont tous ces éléments sont égaux à zéro.

  Exemple :

  ```
  >> C= zeros (3 , 5)
       0     0     0     0     0
       0     0     0     0     0
       0     0     0     0     0
  ```

- **magic (n) :** la fonction magic (n) construit une matrice magique carrée (n x n) formée des entiers de 1 à (n^2) dont la somme des lignes, des colonnes et des diagonales est la même. Cette condition est valable pour tout n > 0 sauf pour n = 2.

  Exemple :

  ```
  >> D= magic (3)
       8     1     6
       3     5     7
       4     9     2
  ```

- **rand (n) :** la fonction rand (n) construit une matrice (n x n) aléatoire dont les éléments sont uniformément distribués dans l'intervalle unitaire.

  Exemple :

  ```
  >> E= rand (3)
     E =
         0.9334    0.8392    0.2071
         0.6833    0.6288    0.6072
         0.2126    0.1338    0.6299
  ```

On peut également construire une matrice aléatoire dont les éléments seront uniformément distribués en spécifiant le nombre de colonnes et des lignes. On écrit la fonction avec les arguments n et m comme suit : **rand (n , m)** ou bien **rand (m , n)**.

Exemple :

```
>> E1= rand (2 , 4)
   E1 =
      0.4565    0.8214    0.6154    0.9218
      0.0185    0.4447    0.7919    0.7382
```

6.6) Fonctions Fliplr et Flipud

- la fonction **fliplr** effectue un retournement de la matrice de gauche vers la droite comme l'indique son nom 'flip left right'.

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

>> B = fliplr (A)     % donne avec matlab
   B =
      3     2     1
      6     5     4
      9     8     7
```

- la fonction **flipud** (up down) effectue un retournement de haut vers le bas .

```
>> C = flipud (A)     donne avec matlab
   C =
      7     8     9
      4     5     6
      1     2     3
```

6.6) Fonctions max et min

- Appliquée à un vecteur, la fonction max (respectivement min) détermine le plus grand élément (respectivement le plus petit élément) du vecteur et éventuellement la position de cet élément dans le vecteur.
- Appliquée à une matrice, la fonction max (respectivement min) retourne la liste des plus grands éléments (respectivement plus petit élément) de chaque colonne.

Exemple :

```
>> s = [5  2  3  1  7] ; [ma , ind] = max (s)
        ma =
                7

        ind =
                5
>> [mi , ind] = min (s)
        mi =
                1

        ind =
                4
>> t = magic (3) ,   [ma , ind] = max (t)
        t =
           8      1      6
           3      5      7
           4      9      2

        ma =
           8      9      7

        ind =
           1      3      2
```

Pour obtenir la valeur de l'élément maximale du tableau on écrit :

```
>> m = max (max (t))
    m =
        9
```

6.7) Fonction mean

- Appliquée à un vecteur, la fonction mean détermine la moyenne des éléments du vecteur.
- Appliquée à une matrice la fonction mean retourne la liste des moyennes des éléments de chaque colonne.

Exemple :

```
>> s = [5  2  3  1  7] ;   m = mean (s)

        m =
            3.6000
>> t = pascal (3) ,   m = mean (t)

    t =
        1     1     1
        1     2     3
        1     3     6

    m =
        1.0000        2.0000        3.3333
```

Pour obtenir la moyenne des éléments du tableau on écrit :

```
>> m = mean (mean (t))
    m =
        2.1111
```

6.8) Opérations sur les matrices

Lorsque des opérations sont appliquées à des nombres (ou à des expressions booléennes) dont la valeur est représenté par une matrice de dimensions (1 x 1), le résultat fourni par ces opérations est le résultat usuel. Lorsque ces opérations sont appliquées à des vecteurs ou plus généralement des matrices, le résultat est bien sûr quelque peu différent. Dans le tableau suivant, A et B sont des matrices et c un nombre :

Opérateur	**Résultat**	**Conditions**
$A + B$	Matrice dont les éléments sont définis par $a_{ij} + b_{ij}$	A et B même format
$A + c = c + A$	Matrice dont les éléments sont définis par $a_{ij} + c$	
$A - B$	Matrice dont les éléments sont définis par $a_{ij} - b_{ij}$	A et B même format
$A - c$	Matrice dont les éléments sont définis par $a_{ij} - c$	
$c - A$	Matrice dont les éléments sont définis par $c - a_{ij}$	
$A * B$	Matrice résultant du produit matriciel de A par B	nb col. A = nb lign. B
$A * c = c * A$	Matrice dont les éléments ont pour valeur $c * a_{ij}$	
$A.* B$	Matrice dont les éléments ont pour valeur $a_{ij} * b_{ij}$	A et B même format
$A\hat{}n\ (n > 0)$	$A * A * \ldots\ldots * A$ (n fois)	A carrée
$A\hat{}n\ (n < 0)$	$A^{-1} * A^{-1} * \ldots\ldots * A^{-1}$ ($\|n\|$ fois)	A inversible
$A.\hat{} B$	Matrice dont les éléments ont pour valeur $(a_{ij})^{b_{ij}}$	A et B même format
A'	Transposée conjuguée de la matrice A, $a'_{ij} = \overline{a_{ij}}$	
$A.'$	Transposée de la matrice A, $a'_{ij} = a_{ij}$ si tous les éléments de A sont réels , $A.' = A'$	
B / A	Matrice X solution de l'équation matricielle $XA = B$ si A est inversible $X = BA^{-1}$	nb col. A = nb col. B
$A \setminus B$	Matrice X solution de l'équation matricielle $AX = B$ si A est inversible $X = A^{-1}B$	nb lign. A = nb lign. B
$A./ B$	Matrice dont les éléments ont pour valeur a_{ij} / b_{ij}	A et B même format
$A.\setminus B$	Matrice dont les éléments ont pour valeur $b_{ij} \setminus a_{ij}$ $A.\setminus B = B./ A$	A et B même format
A /c	Matrice dont les éléments ont pour valeur a_{ij} / c	

On notera que certains opérateurs sont associés à un opérateur pointé (* et .* par exemple). De façon générale, l'opérateur pointé correspond à une opération semblable à celle représentée par l'opérateur non pointé, mais appliquée non pas « matriciellement » mais terme à terme : * est l'opérateur matriciel alors que .* est l'opérateur du produit « terme à terme ».

Exemple :

On crée deux matrices a et b. La fonction ones (n) crée une matrice carée d'ordre 2 dont tous les éléments sont égaux à 1 ; la fonction eye (2) crée la matrice d'identité d'ordre 2

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

    a =
         1       2
         1       0
>> b = ones (2) +eye (2)

    b =
         2       1
         1       2
>> c = a * b

    c =
         4       5
         2       1

>> d = a .* b

    d =
         2       2
         1       0
```

6.9) Opérations booléennes

Dans la suite, « matrice booléenne » désignera une matrice dont les éléments ont pour valeur 0 (représentant faux) ou 1 (représentant vrai). Les opérateurs booléens sont peut être ceux dont le comportement peut apparaître le plus déroutant parce qu'ils fonctionnent « terme à terme ».

Exemple :

Avec les variables a et b définis dans l'exemple précédent on obtient :

```
>> a , b
   a =
        1     2
        1     0
  b =
        2     1
        1     2

>> a = = b
      ans =
        0     0
        1     0
```

Dans l'expression a = = b la comparaison porte sur les éléments homologues de a et b :

$$1 == 2 \rightarrow faux \rightarrow 0, \quad 2 == 1 \rightarrow faux \rightarrow 0$$
$$1 == 1 \rightarrow vrai \rightarrow 1, \quad 0 == 2 \rightarrow faux \rightarrow 0$$

Il en est de même pour l'expression a > b :

```
>> a > b
   ans =
        0     1
        0     0
```

$$1 > 2 \rightarrow faux \rightarrow 0, \quad 2 > 1 \rightarrow vrai \rightarrow 1$$
$$1 > 1 \rightarrow \text{faux} \rightarrow 0, \quad 0 > 2 \rightarrow faux \rightarrow 0$$

Dans l'exemple suivant, on évalue a & b et a | b. Les valeurs des éléments des matrices a et b sont converties avant l'évaluation en valeurs booléennes avec la convention habituelle : 0 vaut faux, toute valeur non nulle vaut vrai :

```
>> a & b

   ans =
        1     1
        1     0

>> a | b

   ans =
        1     1
        1     1
```

En résumé, si a et b sont des matrices de même format et si op désigne un des opérateurs booléens <, >, <=, >=, &, |, le résultat de « a op b » est défini par :

Opérateur	Résultat	Condition
a op b	« Matrice booléenne » dont les éléments sont définis par $a_{ij}\ op\ b_{ij}$	même format

6.10) Déterminant, inverse et valeur propre d'une matrice

- **Déterminant :** det (X) est le déterminant de la matrice carrée X

Exemple : Nous avons déjà vu que la transposition s'obtient à l'aide de l'apostrophe « **'** ». Ajouter une matrice à sa transposée donne une matrice symétrique, multiplier aussi une matrice par sa transposée donne aussi une matrice symétrique, vérifions tout cela :

```
>> A = magic(4)
   A =
       16     2     3    13
        5    11    10     8
        9     7     6    12
        4    14    15     1
```

```
>> B = A + A'
    B =
        32     7    12    17
         7    22    17    22
        12    17    12    27
        17    22    27     2

>> det (B)
    ans =
         0

>> C = A * A'
    C =
       438   236   332   150
       236   310   278   332
       332   278   310   236
       150   332   236   438
>> det (C)
    ans =
         0
```

- **Inverse :** inv(X) donne l'inverse de la matrice carrée X. Il faut noter qu'un message d'alerte apparaît lors du calcul de l'inverse d'une matrice à déterminant nul.

  Exemple : Reprenons l'exemple de la matrice C précédente, son déterminant est nul donc elle n'est donc pas inversible.

  ```
  >> inv (C)
  Warning: Matrix is close to singular or badly scaled.
           Results may be inaccurate. RCOND = 1.459809e-017.
  ```

```
ans =

   1.0e+013 *

       -0.2469   -0.7407    0.7407    0.2469
       -0.7407   -2.2222    2.2222    0.7407
        0.7407    2.2222   -2.2222   -0.7407
        0.2469    0.7407   -0.7407   -0.2469
```

Matlab utilise le calcul flottant pour calculer les inverses des matrices, il est ainsi capable d'inverser la matrice mais indique que le résultat risque d'être erroné car le conditionnement réciproque et proche de la précision machine

- **Valeurs propres :** eig (X) est un vecteur qui donne les valeurs propres de la matrice carrée X.

  Exemple :

```
>> A= magic(4)
   A =
        16     2     3    13
         5    11    10     8
         9     7     6    12
         4    14    15     1


>> B = eig (A) '
    B =
        34.0000    8.9443   -8.9443    0.0000
```

Une des valeurs propres est nulle ce qui confirme que la matrice n'est pas inversible

6.11) Résumé des manipulations matricielles :

Commande	Résultat
A (i , j)	matrice formée des éléments $a_{i\,j}$
A (**:** , j)	$j^{ème}$ colonne de la matrice A

A (i , :)	$i^{ème}$ ligne de la matrice A
A (k:l , m:n)	sous matrice de A composée des lignes k à l et des colonnes m à n
n = rank(A)	n égale au rang de la matrice A
x = det (A)	*x* égale au déterminant de la matrice A
z = size (A)	z est le vecteur ligne contenant les dimensions de la matrice A
C = A+ B	somme de deux matrices
C = A- B	différence de deux matrices
C = A* B	produit de deux matrices
C = A .* B	produit termes à termes des matrices A et B
C = A ^ k	puissance de matrices
C = A . ^ k	matrice dont les termes sont les puissances des termes de A
C = A ′	transposée de A
C = A ./ B	matrice dont les termes sont les divisions termes à termes de A et B
X = A \ B	solution du système AX = B
X = B / A	solution du système XA = B
C = inv (A)	calcul de l'inverse de A
L = eig (A)	calcul des valeurs propres de A
[X,D] = eig(A)	produit une matrice diagonale D formée des valeurs propres de A et d'une matrice X dont les colonnes sont les vecteurs propres associés.
A = eye(n)	matrice d'identité de taille n
A = ones(n,m)	matrice formée de uns
A = zeros(n,m)	matrice formée de zeros
A = diag(v)	matrice diagonale dont les éléments diagonaux sont les éléments du vecteur v
X = tril(A)	partie triangulaire inférieure de la matrice A
X = triu(A)	partie triangulaire supérieure de la matrice A
A = rand(n,m)	matrice aléatoire à éléments uniformément distribués
A =magic(n)	matrice carrée magique de taille n
v = max(A)	vecteur contenant les valeurs maximales de chaque colonne de A
v = min(A)	vecteur contenant les valeurs minimales de chaque colonne de A
v = mean(A)	vecteur contenant la moyenne des valeurs de chaque colonne de A
v = sum(A)	vecteur contenant la somme des valeurs de chaque colonne de A

PARTIE 2 : Programmer avec Matlab

1) Introduction

Un script matlab est composé d'une suite d'instructions toutes séparées par une virgule ou un point virgule. La différence entre ces deux types de séparation est liée à l'affichage ou non du résultat à l'écran. Comme tout langage, matlab possède aussi un certain nombre d'instructions syntaxiques (boucles simples, conditionnelles, etc.…) et de commandes élémentaires (lecture, écriture, etc.…).

Dès que le calcul à effectuer implique un enchaînement de commandes un peu compliqué, il vaut mieux écrire ces derniers 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 l'icône **new 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.

2) Premier script

Créer dans le répertoire courant un **M-file** en cliquant sur l'icône **new M-file** et taper les instructions suivantes :

```
% premier script
a = 1 ;
b = 2 ;
c = a + b ;
```

Sauver (en cliquant sur l'icône save) sous le nom premierscript**.**m et exécuter la commande

```
>> pemierscript
```

Dans la fenêtre matlab, soit en le tapant au clavier soit en cliquant sur l'icône Run à partir de la fenêtre d'édition. Taper maintenant

```
>> c
```

Matlab renvoie la valeur de c calculée par le script.

Remarque :

Plusieurs types de données sont disponibles dans matlab. Les types traditionnels que l'on retrouve dans tous les langages de programmation : les types (single, double, etc.…) caractères « char », les tableaux de réels ainsi que les types définis par l'utilisateur comme les fonctions. Le type de données privilégié sous matlab est les tableaux à une ou deux dimensions correspondant aux vecteurs et matrices utilisés en mathématiques et qui sont aussi utilisés pour la représentation graphique. Ayant déjà vu les vecteurs et matrices ; nous allons définir les caractères, les entrées sorties et les structures algorithmiques.

3) Chaînes de caractères

3.1) Les caractères

En matlab, les caractères constituent un type de données au même titre que les nombres réels. Ainsi **'1'** représente le caractère « 1 » qui n'est pas de même nature que le nombre 1. Une chaîne de caractères est un vecteur dont les entrées sont des caractères. Une présentation de la gestion des chaînes de caractères sous matlab est donnée en faisant **help string**.

Exemple :

```
>> S = 'Elément positif'
```

Dans matlab, ceci définit une variable S de type chaîne de caractères qui n'est rien d'autre qu'un vecteur de caractères. En tant que vecteur :

S(1) vaut **' E '** et
S(4) vaut **' m '** **t**andis que :
length(S) vaut **15**

La commande « **ischar »**permet de tester si une expression est du type chaîne de caractères.

Exemple : Considérons T un vecteur composé d'une chaîne de caractères

```
>> T = ' 1.23 '     matlab répond
>> T =
        1.23
```

Effectuons le test si T est composé d'une chaîne de caractères :

>> ischar (T)

>> ans =

1

Considérons maintenant T mais composé d'un chiffre

>> T = **1.23** matlab répond

>> T =

1.2300

Effectuons le teste si T est une chaîne de caractères :

>> ischar (T)

>> ans =

0

3.2) Concaténation de chaînes de caractères

Les chaînes de caractères étant des vecteurs, elles sont donc concaténables comme pour les vecteurs.

Exemple : Considérons T un vecteur composé d'une chaîne de caractères

>> T = ['Ceci est' , ' ␣une chaîne ', ' ␣de caractères ']

T =

Ceci est une chaîne de caractères

Remarque : On peut également utiliser les commandes « strcat » ou « strvcat » pour une concaténation horizontale ou verticale

3.3) Conversion d'un réel en chaîne de caractères

Il est possible de convertir un réel en une chaîne de caractères qui représente son écriture en base 10. Ainsi la commande « num2str » convertit un réel en chaîne de caractère par contre l'opération inverse est obtenue par la commande « str2num ».

Exemple : Considérons le même exemple que précédemment

```
>> x =  num2str(1.23)
>> x =
        1.23
>> ischar (x)
>> ans =
        1
```

Faisons l'opération inverse c'est à dire passons de la chaîne de caractères au numérique

```
>> y =  str2num(x)
>> y =
        1.2300
>> ischar (y)
>> ans =
        0
```

4) Entrées sorties

4.1) Commande input

La commande « **input** » permet de demander à l'utilisateur d'un programme de fournir des données en les saisissant au clavier. Elle permet d'afficher une question et de récupérer la réponse dans une variable. La syntaxe est **var = input(' une phrase ')**.

La phrase est affichée et MATLAB attend que l'utilisateur saisisse une donnée au clavier. Cette donnée peut être une valeur numérique ou une instruction MATLAB. Un retour chariot provoque la fin de la saisie. Une valeur numérique est directement affectée à la variable **var** tandis qu'une instruction MATLAB est évaluée et le résultat est affecté à la variable **var**. Il est possible de provoquer des sauts de ligne pour aérer le présentation en utilisant le symbole **\n** de la manière suivante: **var = input('\n une phrase : \n ').**

Si l'on souhaite saisir une réponse de type chaîne de caractères on utilise la syntaxe :

var = input(' une phrase ','s').

Exemple :

```
>> n = input ('Donnez la valeur de n : ')        % matlab renvoie
Donnez la valeur de n : 134                      % On saisit au clavier 134
n =
   134
```

4.2) Commande sprintf

La commande « **sprintf** » permet de construire des chaînes de caractères à partir de variables de différents types. Un modèle d'édition se présente sous la forme du symbole pourcent (%) suivi d'indications permettant de composer le contenu du champ à écrire, en particulier sa longueur en nombre de caractères. Réciproquement, la commande « **sscanf** » permet de décomposer une chaîne de caractères en plusieurs variables de différents types. Ces deux commandes généralisent les commandes **num2str** et **str2num** et sont très pratiques.

4.2..1) Modèle d'édition des réels

Un modèle d'édition de réel est de la forme **% +- L.D t**, où **%** est le symbole de début de format, **L** est un entier donnant la longueur total du champ (en nombre de caractères, point virgule compris), **D** est le nombre de décimales à afficher et **t** spécifie le type de notation utilisée. Le symbole **- (moins)** permet de justifier à gauche. Le symbole **+ (plus)** provoque l'affichage systématique d'un signe **+** devant les réels positifs. Les principales valeurs possibles pour **t** sont les suivantes:

d : pour les entiers

e : pour une notation à virgule flottante où la partie exposant est délimitée par un e minuscule (ex: 3.1415e+00)

E : même notation mais E remplace e (ex: 3.1415E+00)

f : pour une notation à virgule fixe (ex: 3.1415)

g : la notation la plus compacte entre la notation à virgule flottante et la notation à virgule fixe est utilisée

Exemple 1 : Illustration de l'utilisation de sprintf

```
>> clear all, clc
>> r = 1.23;
>> n = 5;
>> chaine = sprintf (′r vaut  %f  et n vaut %d ′,r, n) ;
>> disp (chaine)
```

Exemple 2 : Illustration du modèle d'édition des réels

```
>> x = pi/3;
>> y = sin(x);
>> sprintf('sin(%11.6f) = %4.2f', x,y)
```

```
ans =
   sin(1.047198) = 0.87
```

Exemple 3 : Illustration de l'utilisation de sscan

```
>> clear all, clc
>> chaine =  ′1.2 : 5.5 : 3.8 : 51′
>> X = sscanf (chaine, ′ %f : %f : %f : %f ′)
```

4.2.2) Utilisations particulières de sprintf

La commande **sprintf** est « vectorielle » : si la variable n'est pas scalaire le format d'impression est réutilisé pour tous les éléments du vecteur, colonne par colonne.

Exemple 1 :

```
>> x = [1:10];
>> sprintf (' %d ,',x)
ans =
 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 ,
```

Exemple 2 : Edition de chaînes de caractères

```
>> sprintf ('%s', 'il fait beau')
ans =
   il fait beau

>> temps = 'il fait beau';
>> sprintf('%s',temps)
ans =
il fait beau
>> sprintf('%15s', temps)   % affichage avec un déplacement du texte à droite (texte justifié)
ans =
        il fait beau
```

4.3) Commande pause

Elle permet de stopper l'exécution de matlab, la reprise de l'exécution sera relancée après l'appui par l'utilisateur de n'importe quelle touche, il est aussi possible de stopper l'exécution de matlab pendant un temps déterminé en utilisant « **pause(n)** » où n représente le

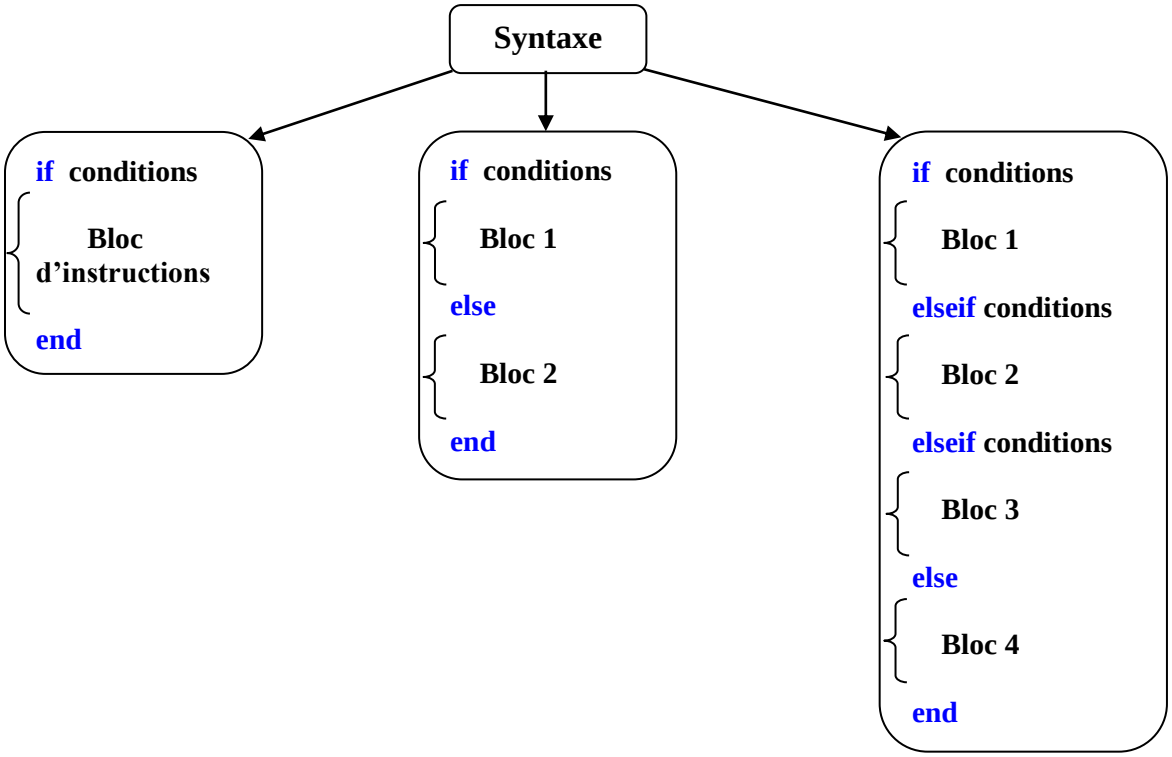
temps d'arrêt en secondes. La reprise de l'exécution s'effectuera automatiquement après épuisement du temps souhaité.

5) Structures algorithmiques :

Comme tous les langages, MATLAB dispose des commandes du type FOR, WHILE, IF . . .

5.1) Commande if

La commande **if** évalue une expression logique et exécute un groupe d'instructions quand l'expression est vraie. Les mots clés optionnels **elseif** et **else** donnent la possibilité d'exécuter d'autres groupes d'instructions si l'expression vraie est fausse. Il faut rajouter à la fin le mot **end** pour indiquer la fin du dernier groupe d'instructions. La syntaxe est alors la suivante :



Exemple : test sur le type d'une variable

```
>>  x = 35 ;
>>  if  length(x) ~= 1
        ' non scalaire '
    elseif  isnan(x)
        ' NaN '
    elseif  x > 0
        ' positif '
    elseif  x < 0
        ' negatif '
    else
        ' nul '
    end
```

Remarque :

Il est important de comprendre le fonctionnement de if. En effet le test **if A = = B** est légal en matlab. Si A et B sont des scalaires, ceci est le test d'égalité entre A et B.

Par contre si A et B sont des matrices **A = = B** ne teste pas si A et B sont égales mais où elles le sont. Le résultat est une matrice de 0 et de 1.

En fait, si A et B ne sont pas de même taille, alors **A = = B** est une erreur. La façon propre de test d'égalité de deux matrices est **if isequal(A , B)**

5.2) Commande switch et case

La commande **switch** exécute un groupe d'instructions suivant la valeur d'une variable a ou d'une expression. Les mots clés **case** et **otherwise** délimitent le groupe. Seulement le premier cas vrai est exécuté. Il doit obligatoirement y avoir un **end** à la fin. La syntaxe est :

```
switch  (expression)
        case exp1
                Bloc 1
        case exp2
                Bloc 2
        case {exp3, exp4, exp5}
                Bloc 3
        otherwise
                Bloc 4
end
```

Exemple : Affichage du type de solution d'une équation du second degré

```
>>   a = 4;
>>   b = 5;
>>   c = 2;
>>   delta = b^2 – 4*a*c;
>>   switch  sign(delta)
            case 1
                disp(' 2 solutions réelles ' );
            case 0
                disp(' solution double ' );
            otherwise
                disp(' 2 solutions complexes ' );
       end
```

5.3) Commande for

La commande de boucle **for** répète un groupe d'instructions un nombre prédéterminé de fois. Un **end** délimite la fin. La syntaxe est :

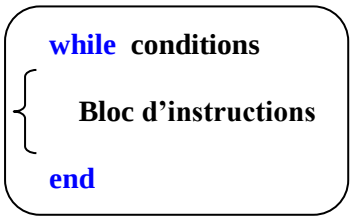
```
for  k = valeur initiale : pas : valeur finale
    Bloc d'instructions
end
```

Exemple : Calcul de la somme des 10 premiers éléments de l'ensemble des entiers naturels

```
>>   N = 10;
>>   s = 0;
>>   for  k = 1 : 10
          s = s + k ;
       end
>> s
 s =
     55
```

5.4) Commande while

La boucle **while** répète un groupe d'instructions un nombre de fois sous le contrôle d'une expression logique. Un **end** termine l'instruction. La syntaxe est :



Exemple : Calcul du produit des 4 premiers éléments de l'ensemble des entiers naturels

```
>>   p = 1;
>>   N = 4;
>>   k = 1;
>>   while  k <= N
          p = p * k ;
          k = k+1 ;
       end
>> p
  p =
       24
```

5.5) Commande break

La commande break permet de quitter une boucle avant la fin de celle-ci. Dans les boucles imbriquées, break ne fait que sortir de la boucle interne.

Exemple : Recherche et affichage du premier élément qui accepte d'être le diviseur de 9 dans l'intervalle [2,6]

```
>>   for  k = 2 : 1 : 6
          if rem (9 , k) = = 0
          sprintf (' le premier élément est %d ', k)
          break ;
          end
       end
```

```
ans =
Le premier élément est 3
```

6) Les Polynômes

Le calcul polynomial est à la base de nombreux domaines scientifiques : traitement du signal, contrôle de procédés, approximation des fonctions et interpolation des courbes. Matlab représente les polynômes sous forme de vecteurs lignes dont les composantes sont ordonnées par ordre des puissances décroissantes. Un polynôme de degré « n » est représenté par un vecteur de taille « n+1 ».

Exemple : Soit la fonction f suivante : $f(x) = 3x^5 + 2x^3 - x + 5$

La représentation de f dans matlab est la suivante :

```
>> f = [3   0   2    0  -1   5]
f =
     3    0    2    0   -1    5
```

6.1) Racines d'un polynôme

Les racines d'un polynôme sont déterminées à l'aide de la commande « roots ».

Exemple : considérons la fonction f précédente. Les racines de f sont alors :

```
>> roots(f)
ans =
      - 1.0460
      - 0.2586 + 1.2072 i
      - 0.2586 - 1.2072 i
        0.7817 + 0.6591 i
        0.7817 - 0.6591 i
```

6.2) Reconstruction d'un polynôme par ces racines

Connaissant les racines d'un polynôme on peut le reconstruire à l'aide de la commande « poly ».

Exemple : considérons la fonction f précédente. Les racines de f sont alors :

```
>> poly( [1   2   3] )
ans =
          1   -6    11    -6
```

Le polynôme est alors : $f(x) = x^3 - 6x^2 + 11x - 6$

6.3) Produit polynomiale

C = conv(A , B) est le produit de convolution des vecteurs A et B. La longueur du vecteur C est égale à la somme des longueurs des vecteurs A+B-1

Exemple : produit des polynômes f1 et f2 :

```
>> f1 = [1   3    5]                % f1(x) = x^2 + 3x + 5
f1 =
       1    3    5
>> f2 = [2   4    6]                % f2(x) = 2x^2 + 4x + 6
f1 =
       2    4    6
>> conv (f1 , f2)
ans =
       2   10   28   38   30
```

Le produit de f1 et f2 est donc : $F(x) = 2x^4 + 10x^3 + 28x^2 + 38x + 30$

6.4) Division polynomiale

« [Q , R] = deconv(B , A) » donne la division du vecteur B par le vecteur A. Le résultat est affecté au vecteur Q (Quotient) et le reste est affecté au vecteur R (Remainder) tel que : « B = conv(A , Q) + R »

Exemple Reprenons les fonctions f1 et f2 et faisons la division de f1 par f2.

```
>> [q , r] = deconv(f1 , f2)
q =
       0.5000

r =
       0    1    2
```

6.5) Evaluation d'un polynôme

Pour déterminer la valeur d'une fonction polynomiale pour un point ou dans un intervalle quelconque on utilise la commande dont la syntaxe est : « polyval(f , x) »

Exemple : Evaluons la fonction $f(x) = 2x^2 + x - 2$ pour $x = 4$

```
>> x = 4 ;
>> f = [2   1  -2] ;
>> polyval (f , x)            %  f (x = 4)
ans =
       34
```

Evaluons maintenant la fonction $f(x) = 2x^2 + x - 2$ dans l'intervalle [0 2]

```
>> x = [0 : 0.5 :  2]
x =
    0    0.5000    1.0000    1.5000    2.0000
>> f = [2   1  -2] ;
>> polyval (f , x)
ans =
      -2     -1      1      4      8
```

6.6) Calcul de la dérivée d'une fonction polynomiale

La dérivée d'une fonction polynomiale est calculée à partir de la commande « polyder »

Exemple : Calculons la dérivée de la fonction $f(x) = 2x^2 + x - 2$

```
>> f = [2   1  -2]
f =
    2     1    -2
>> polyder (f)
ans =
      4     1
```

La dérivée de f est : $f'(x) = 4x + 1$

7) Le Graphisme

Matlab possède un vaste ensemble de fonctionnalités graphiques. Nous ne présenterons que quelques principes de base qui serviront à visualiser les courbes et les surfaces. Que ce soit pour tracer le graphe d'une fonction dans le plan ou dans l'espace, la technique peut se décomposer en trois temps.

1. Discrétiser le domaine de représentation.
2. Evaluer la fonction en chaque point de ce domaine discrétisé.
3. Exécuter l'instruction graphique avec les données précédentes.

7.1) Visualisation des courbes en 2 dimensions

❖ Matlab permet de tracer facilement le graphe de fonctions scalaires. Cela se fait à l'aide de la fonction « **plot** ». Soient deux vecteurs *x* et *y* de même longueur, la fonction **plot(x,y)** trace dans une fenêtre le graphe de y en fonction de x.

En fait le graphe est obtenu en joignant par de petits segments de droite de coordonnées $(x(k), y(k))$ pour $(1 \le k \le \text{length}(x))$, et

- x contient les abscisses
- y contient les ordonnées

Exemple 1 :

Soit la fonction $y = \cos(3x)$ sur l'intervalle [0, 2π], le tracé de y s'obtient comme suit :

```
>> x = [0 : 0.01 : 2*pi];
>> y = cos(3*x);
>> plot(x,y)
```

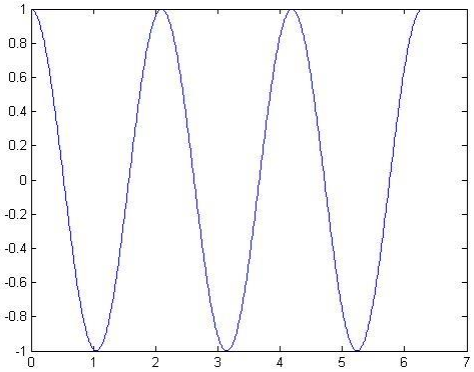


Fig. Tracé de y =cos(3x)

❖ Matlab définit automatiquement un système d'axes. La qualité de la courbe dépend du nombre de points construits comme le montre la courbe suivante pour laquelle on a choisit un pas plus petit pour décrire y dans l'intervalle [0,2π]. On remarquera la qualité du tracé dans l'exemple 2.

Exemple 2 :

Reprenons la fonction $y = \cos(3x)$ sur l'intervalle [0, 2π] et prenons un pas plu petit :

```
>> x = [0 : 0.3 : 2*pi];
>> y = cos(3*x);
>> plot(x,y)
```

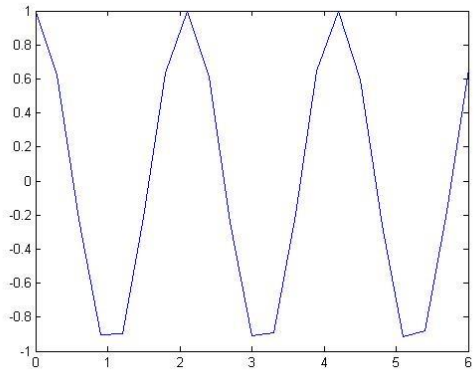


Fig. Tracé de y =cos(3x) avec un pas plus petit

❖ Les types de tracé de fonctions sont des tracés continus avec une interpolation entre les points fournis. On peut modifier la courbe en modifiant ces propriétés à l'aide de la commande suivante : **plot(x,y,′propriété′,'valeur'**). Les propriétés peuvent être la couleur, le tracé, l'épaisseur du tracé…etc. On peut même utiliser une combinaison de 2 propriétés à l'aide de la commande suivante :

plot(x,y,′propriété1′,'valeur', ′propriété2′,′valeur′)

Couleur		**Tracé**	
bleu	b	trait continu	-
vert	g	trait discontinu	--
rouge	r	pointillés	**:**
cyan	c	tiret	**-.**
magenta	m	point	**.**

jaune	y	cercle	**o**
noir	k	croix	**x**
		plus	**+**
		étoile	*****
		carré	**s**
		triangle	**v , ^ , > , <**

Exemple 3 : Modifions la couleur et l'épaisseur du tracé de la fonction $y = \cos(3x)$:

```
>> x = [0 : 0.05 : 2*pi];
>> y = cos(3*x);
>> plot(x,y,'color','r','linewidth',2.5)
```

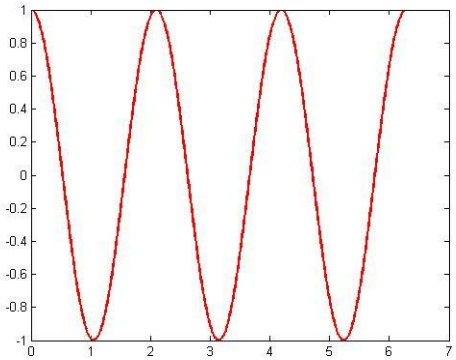


Fig. Modification des propriétés du graphe de y=cos(3x)

Remarque 1 : On peut choisir une combinaison de couleur en précisant dans un vecteur les proportions des trois couleurs principales à savoir [RougeVert Bleu], dans matlab on écrit :

```
>> plot(x,y,'color',[0.7 0.4 0.9])
```

Remarque 2 : Dans l'exemple 3, on peut simplifier l'écriture en spécifiant ce qui suit :

```
>> plot(x,y,'r-.','linewidth',1.5)
```

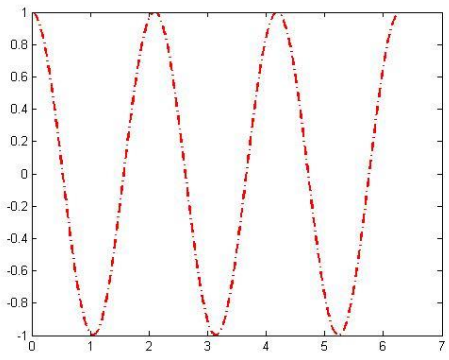


Fig. Modification des propriétés du graphe de y =cos(3x)

Remarque 3 : On peut utiliser la combinaison de 2 types de tracés, on écrit dans matlab

```
plot(x,y,'--rs','MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',5)
```

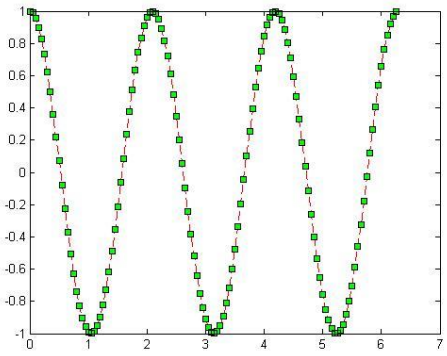


Fig. Modification des propriétés du graphe de y =cos(3x)

❖ On peut tracer les graphes de plusieurs fonctions simultanément. Suivant le contexte, on peut souhaiter que :

1. Les tracés apparaissent dans des fenêtres différentes : on crée autant de fenêtre que de tracés en utilisant la fonction « **figure »**

Exemple 4 :

Traçons les graphes des fonctions $y = \cos(3x)$ et $z = \sin(3x)$ sur l'intervalle $[0, 2\pi]$

```
>> x = [0 : 0.01 : 2*pi];
>> y = cos(3*x);
>> plot(x,y);
>> z = sin(3*x);
>> figure(2);
>> plot(x,z)
```

Ces deux séquences construisent deux fenêtres, la première contenant le graphe de cos(3x), la seconde le graphe de sin(3x). On peut remarquer que les tracés ont tous la même couleur

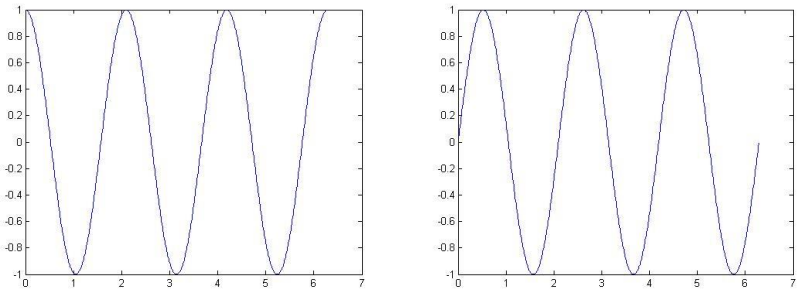


Fig. Graphes des fonctions y =cos(3x) et z =sin(3x)

2. Tous les tracés apparaissent simultanément dans la fenêtre active : on peut procéder de deux façons :

- Soit en utilisant les commandes « **hold on »** et « **hold off »** : après « hold on » tous les tracés ont lieu dans la fenêtre active, « hold off » fait revenir au mode de tracé normal.

Exemple 5 :

Reprenons le tracé des fonctions $y = \cos(3x)$ et $z = \sin(3x)$ sur l'intervalle [0, 2π] dans une seule fenêtre en utilisant les fonctions hold on et hold off

```
>> x = [0 : 0.01 : 2*pi];
>> y = cos(3*x);
>> plot(x,y);
>> hold on
>> z = sin(3*x);
>> plot(x,z,'r'); hold off
```

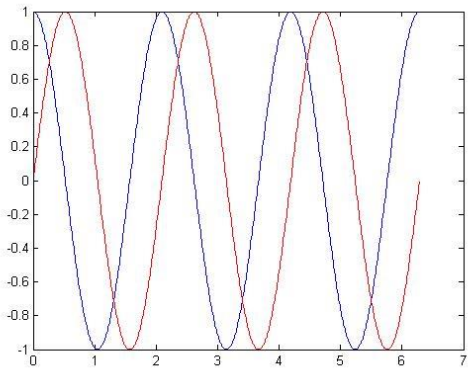


Fig. Tracés des fonctions y et z en utilisant hold on et hold off

- Soit en donnant comme argument à **plot** la **liste** de tous les couples de vecteurs correspondants aux courbes à tracer : **plot(x_1 , y_1 , x_2 , y_2 , …)**.

  Lorsque plusieurs tracés ont lieu dans la même fenêtre, il peut être intéressant d'utiliser un style différent pour distinguer les différents tracés. On ajoute un troisième argument à la définition de chaque tracé **plot(x_1, y_1 ,'st_1', x_2, y_2, 'st_2',…)**.

Exemple 6 :

Traçons les fonctions $y = \sin(3x)$ et $z = \sin(4x) + 2\cos(5x)$z sur l'intervalle [0, 2π] dans une seule fenêtre avec le plot d'une liste de vecteurs et spécifions des tracés différents pour y et z

```
>> x = [0 : 0.01 : 2*pi];
>> y = sin(3*x);
>> z = sin(4*x)+2*cos(5*x);
>> plot(x,y,'r+',x,z,'*','markersize',1.8)
```

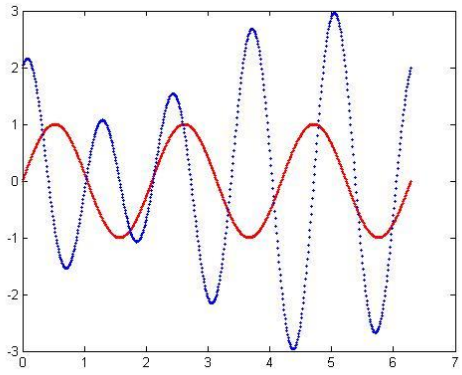
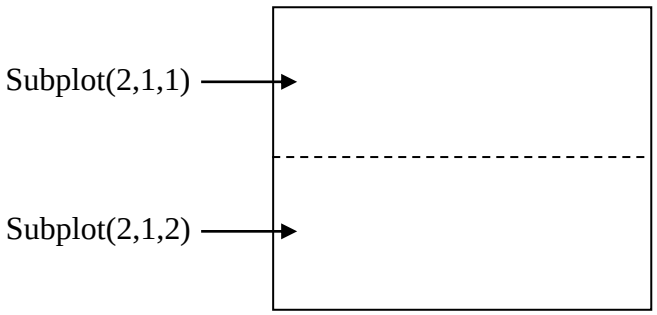


Fig. Utilisation de 2 couples de vecteurs dans la commande plot

- Soit en en utilisant la fonction subplot pour diviser la fenêtre en plusieurs parties.
  - Diviser la fenêtre en 2 parties (2x1)



Exemple 7 :

Traçons les fonctions $y = \sin(3x)$ et $z = \sin(4x) + 2\cos(5x)$ en utilisant subplot

```
>> x = [0 : 0.01 : 2*pi];
>> y = sin(3*x);
>> z = sin(4*x)+2*cos(5*x);
>> subplot(2,1,1); plot(x,y);
>> subplot(2,1,2); plot(x,z);
```

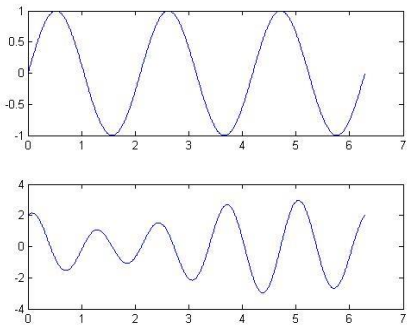
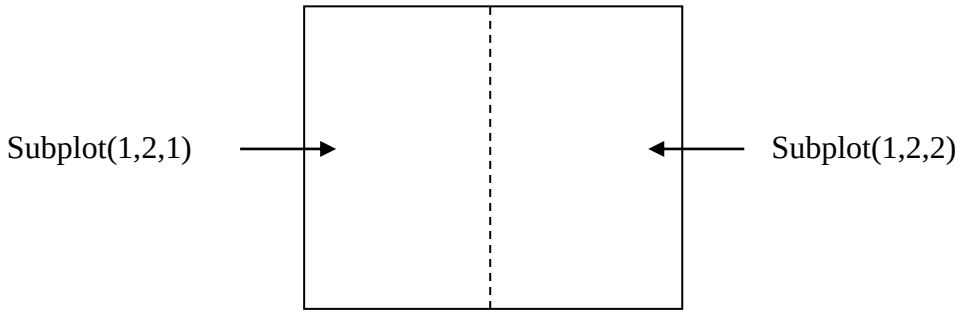
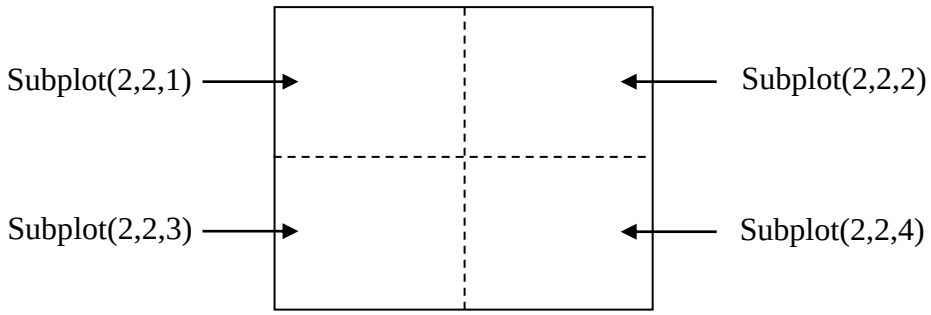


Fig. Utilisation de subplot dans un tracé

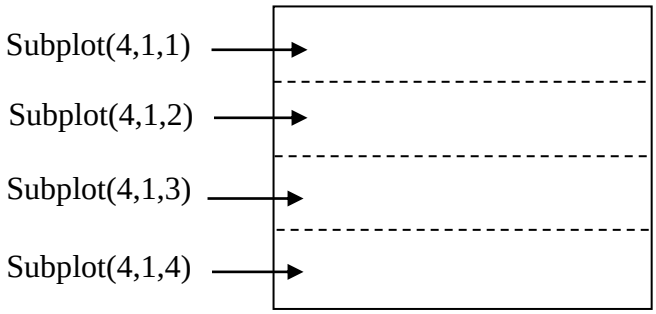
- Diviser la fenêtre en 2 parties (1x2)



- Diviser la fenêtre en 4 parties (2x2)



- Diviser la fenêtre en 4 parties (4x1)



❖ Il est préférable de légender les graphiques pour qu'ils soient explicites, on utilise pour cela la fonction « **legend** », on rajoute ensuite la légende qu'on souhaite afficher :

Exemple 8 : Reprenons les fonctions $y = \cos(3x)$ et $z = \sin(3x)$ traçons les courbes et rajoutons la ligne comme légende du tracé :

```
>> x = [0 : 0.01 : 2*pi]; y = cos(3*x); z = sin(3*x);
>> plot(x,y,'--',x,z,'*','markersize',1.7);
>> legend('cos(3x)','sin(3x)')
```

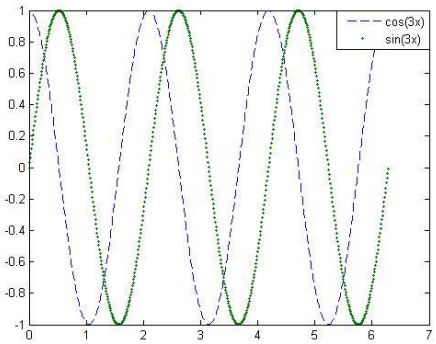


Fig. Utilisation de la légende des fonctions

❖ Dans matlab, les axes sont définis automatiquement, on peut choisir les bornes des coordonnées du graphe à l'aide de la fonction « **axis** ». On peut aussi donner un titre à la courbe et étiqueter les axes des abscisses et des ordonnées.

Exemple 9 : les instructions suivantes vont réduire l'échelle du tracé, donner un nom, et faire apparaître une grille sur le fond de la figure pour la fonction $y = \cos(3x)$

```
>> x = [0 : 0.01 : 2*pi] ; y =cos(3*x) ; plot(x,y)
>> axis ([-1.1   1.1    -1.1   1.1]) ;
>> title('Courbe de la fonction y = cos3x')
>> xlabel('axe des abscisses')
>> ylabel('axe des ordonnées'); grid
```

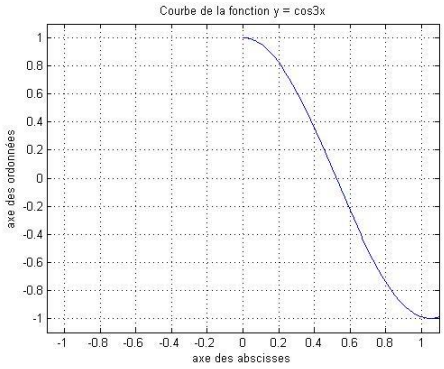


Fig. Tracé réduit de la fonctions y =cos(3x)

7.2) Autres fonctions de tracé de courbes planes

Les fonctions dérivées de plot sont nombreuses, on peut se référer à l'aide pour en avoir une liste exhaustive

- **La Fonction plotyy** : **plotyy(x_1 , y_1 , 'st$_1$', x_2 , y_2 , 'st$_2$', …)** trace y_1 en fonction de x_1 et y_2 en fonction de x_2 avec 2 axes des y l'un à gauche adapté à y_1 et l'autre à droite adapté à y_2

- **La Fonction stem:** est utilisée pour tracer des courbes échantillonnées. La syntaxe est :
  stem(x,y)
  stem(x,y,'fill')

Exemple 10 :

```
>> x = 0:0.1:2*pi; y = sin(x);
>> subplot (1,2,1),stem(x, y,'k')
>> subplot (1,2,2),stem(x, y,'fill','vr',...
     MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',5)
```

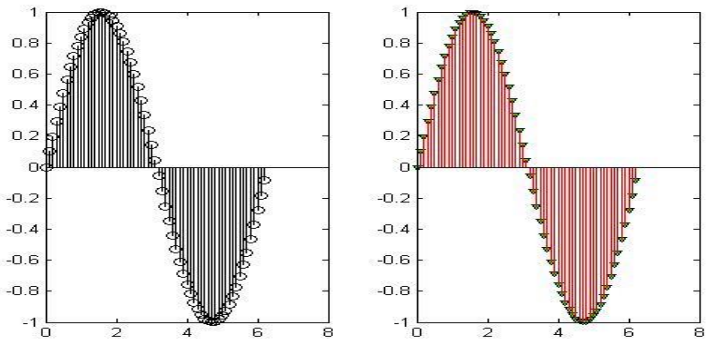


Fig. Utilisation de stem et 'fill' dans un tracé

❖ **La fonction stairs :** Une fonction qui est spécialement adaptée à l'affichage en escalier des signaux discrets est « **stairs** » ceci permet de représenter le signal sous la forme d'une fonction escalier.

Exemple 11 : soit la fonction $y = \cos(t)$ utilisons stairs pour tracer y.

```
>> x = 0:0.2:4*pi;
>> y = 3*cos(x);
>> stairs(x, y,'k');
>> xlabel('x'); ylabel('cos(x)','rotation',0)
```

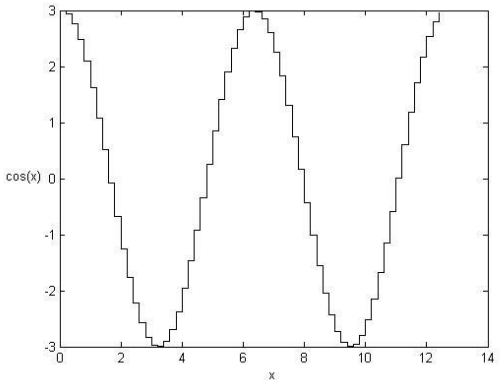


Fig. Tracé de cos(3x) avec stairs

❖ **La fonction bar :** Elle permet de tracer de histogrammes d'un tableau ou d'une matrice. La syntaxe est :

bar(x)

bar(x,y)

bar(x,y,style) % style prend les valeurs ′group′ ou ′stack′

Exemple 12 :

```
>> x = [-5:0.3:5];
>> y = exp(-x.^2);
>> bar(x, y)
```

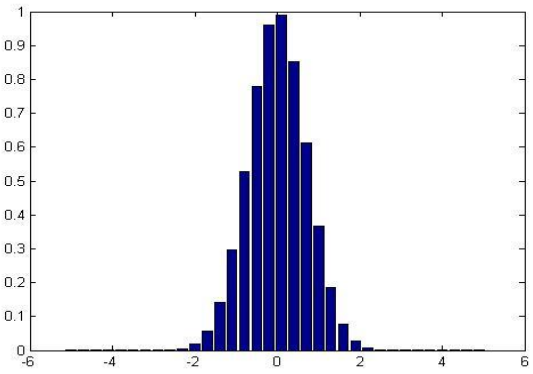


Fig. Tracé d'un histogramme

7.3) Visualisation des courbes en 3 dimensions

Le plus important en trois dimensions des fonctions graphiques sont certainement « **mesh** » et « **surf** ». Là encore, nous ne discuterons pas toutes les fonctionnalités de MATLAB en détail, on se limitera aux fonctions essentielles. Voyons d'abord la fonction qui permet de représenter une courbe dans l'espace $\Re^3$ « **plot3** » dont la syntaxe est : **plot3(x,y,z,s)** où : x, y et z sont les vecteurs des trois coordonnées.

Exemple 13 : Traçons une hélice circulaire définie par une équation paramétrique

```
>> t = linspace(0,10*pi,500) ;
>> x = cos(t) ; y = sin(t) ;
>> plot3(x,y,t,'r*') ;
>> title('Helice') ;
>> xlabel('x'),ylabel('y'),zlabel('t','Rotation',0); grid
```

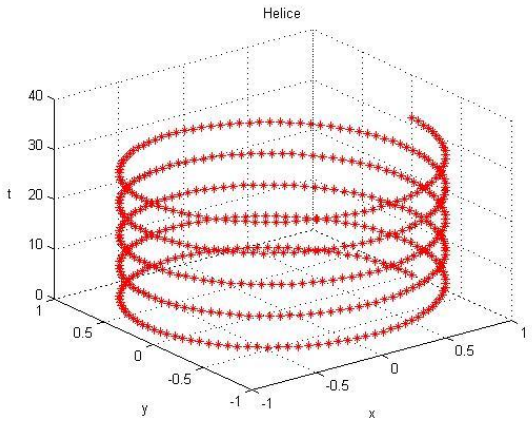


Fig. Tracé d'une hélice circulaire en 3D

Remarques :

1. Matlab donne une vue perspective du graphe de la fonction inclus dans un parallélépipède à l'aide de la fonction « box on », la fonction « box off » permet de revenir en mode normal.
2. La fonction « rotate3d » permet à l'aide de la souris le déplacement de la figure en 3D.

❖ Ces deux fonctions sont à rajouter après le plot.

```
>> plot3(x,y,t,'r*'); rotate3d on; box on;
```

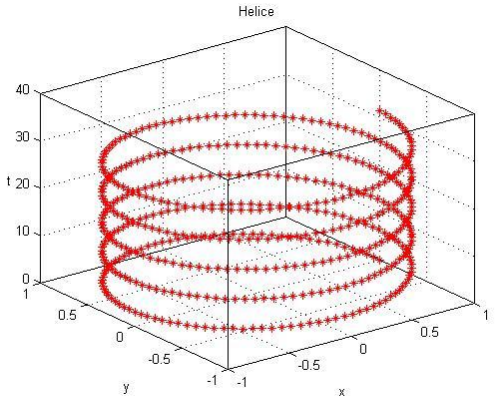


Fig. Tracé d'une hélice dans une boite 3D

7.3) Visualisation des surfaces

Matlab permet de représenter les surfaces données en coordonnées cartésiennes ou sous forme paramétrique qui peut s'écrire :

$$x = x(s,t)$$
$$y = y(s,t)$$
$$z = z(s,t)$$

Où les paramètres *s* et *t* parcourent un certain domaine.

Plusieurs fonctions permettent cette représentation, parmi lesquelles « **mesh** » représente un treillis, et « **surf** » une surface pleine. Elles ont le même format d'appel :

surf(X,Y,Z,C)

X, Y et Z sont des matrices de mêmes dimensions contenant les coordonnées de points de la surface. La matrice C permet de définir les couleurs et peut aussi être omis (voir la suite remarque 2).

Exemple 14 : Faisons la représentation du cône d'équation $x^2 + y^2 - 2xz = 0$ en utilisant la représentation paramétrique donnée pour $(\theta, \rho) \in]-\pi, \pi[\times]0, 12[$ par :

$$\begin{cases} x = \rho(1 + \cos\theta) \\ y = \rho \sin\theta \\ z = \rho \end{cases}$$

On procède de la façon suivante :

```
>> n = 31;
>> theta = pi*[-n:2:n]/n ;
>> r = linspace(0,12,n);
>> X = r'*(1+cos(theta));
>> Y = r'*sin(theta);
>> Z = r'*ones(size(theta));
>> surf(X,Y,Z);
>> Xlabel('X'),Ylabel('Y'),zlabel('Z','rotation',0)
```

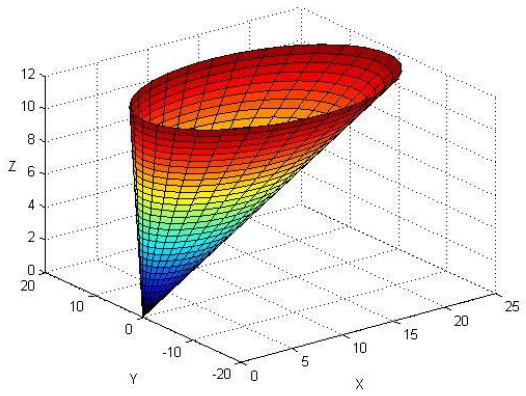


Fig. Courbe du cône d'équation $x^2 + y^2 - 2xz = 0$

Remarque 1 : Il est important de remarquer que X, Y et Z sont des matrices de même taille. La surface représentée laisse apparaître les traces d'une grille qui la recouvre. L'aspect de la surface est contrôlé par la commande « shading » dont les valeurs possibles sont « faceted » (valeur par défaut), « flat » (supprime le treillis), et « interp » (adoucit les transistions).

- Essayons la commande :

```
>> shading interp
```

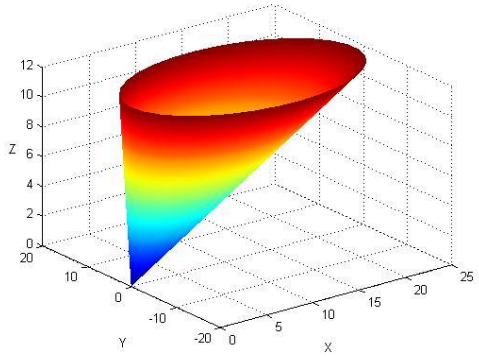


Fig. Courbe du cône d'équation $x^2 + y^2 - 2xz = 0$

Remarque 2 :

Les couleurs sont définies par une matrice C de même taille que celle des coordonnées, et par défaut cette matrice est C = Z de sorte que d'une certaine façon la couleur est « proportionnelle » à l'altitude de point de la surface. Tout cela dépend du choix d'une palette graphique.

- Essayons :

```
>> C = rand(size(Z));
>> surf(X,Y,Z,C)
```

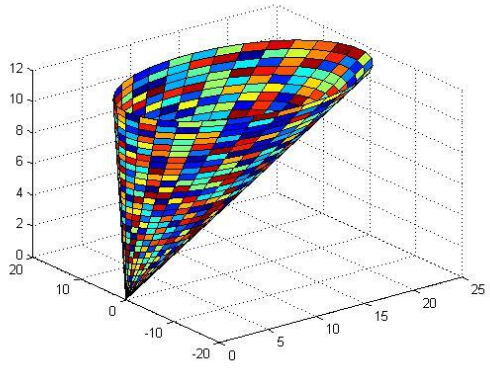


Fig. Courbe du cône d'équation $x^2 + y^2 - 2xz = 0$

Remarque 3 :

La palette de couleurs de la fenêtre de visualisation est définie par la variable « colormap ». Sa valeur par défaut est « jet »

- Essayons :

```
>> colormap('cool')
```

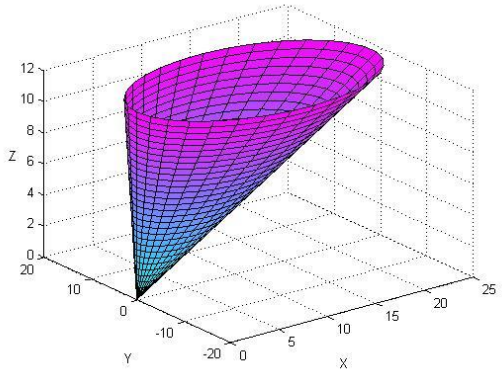


Fig. Courbe du cône d'équation $x^2 + y^2 - 2xz = 0$

Remarque 4 :

La liste des palettes disponibles est fournie avec celle de toutes les « functions » ou variables de contrôle d'un graphique 3D par la commande :

```
>> more on,
>> help graph3d,
>> more off
```

Exemple 15 : Examinons à présent le tracé d'une surface par son équation cartésienne $z = f(x, y)$. En général, on dispose d'une famille de valeurs de x et d'une famille de valeurs de y stockés dans 2 vecteurs respectivement x et y. On va devoir construire deux matrices X et Y telles que les colonnes de X et les lignes de Y soient constantes : ces matrices seront utilisés pour le calcul des valeurs de f. On peut utiliser la fonction « meshgrid ». Ainsi pour représenter la surface d'équation $z = \exp(-x^2 - y^2) \quad \forall \; -2 < x, y < 2$.

On procède comme suit :

```
>> [X,Y] = meshgrid(-2:0.2:2, -2:0.2:2);
>> Z = X.*exp(-X.^2 - Y.^2);
>> surf(Z);
```

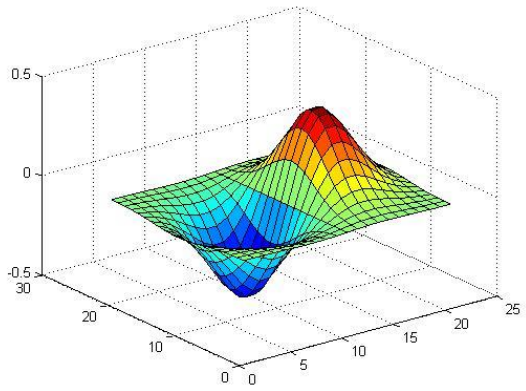


Fig. Tracé de surface avec meshgrid

8) Programmation des fonctions sous MATLAB.

8.1) Introduction

Lorsque nous écrivons un grand nombre de programmes, nous trouverons qu'il y a certains types de calculs que nous effectuons et qui se répètent souvent. Dans ces situations, il serait intéressant d'écrire une fonction MATLAB qui peut être utilisé par de nombreux programmes. Rappelons qu'une fonction en mathématiques est une règle selon laquelle une variable d'entrée est transformée dans une variable de sortie.

Soit la fonction suivante : $f(x) = x^3 + 4x^2 + 2x + 3$

Alors : $y = f(5)$ équivaut à dire prendre la valeur 5 et l'évaluer dans f.

Vue de cette manière, nous pouvons dire que la notation fonction est vraiment un un raccourci pour le calcul d'une valeur de sortie connaissant une valeur d'entrée quelconque et un ensemble de règles. L'utilisation de la notation « function » pourrait donc remplacer la partie dépendant du problème du programme par une syntaxe simple :

Calculé = Nom_de_la_fonction (arguments d'entrée)

Où les arguments d'entrée sont les valeurs nécessaires pour obtenir le calcul souhaité. Dans le cas de la fonction précédente $f(x)$, la valeur d'entrée étant *x*.

Supposons que nous avons écrit un programme pour calculer la valeur maximale de $f(x)$. Il y a différents algorithmes pour trouver les maxima. La structure de l'algorithme lui-même est généralement indépendante de la forme particulière de $f(x)$. Par conséquent, nous

pourrions écrire le programme de telle manière que tous les calculs sont effectués en utilisant la fonction MATLAB.

On pourrait alors utiliser ce programme pour trouver les maxima d'une variété de fonctions et à chaque fois qu'on change le problème nous pouvons changer le nom de la fonction pour faire la correspondance avec le problème à traiter.

8.2) Règles pour l'écriture de fonctions

Selon les règles de MATLAB la syntaxe utilisée pour l'écriture d'une fonction est la suivante :

function [arguments de sorite] = Nom_de_la_fonction (arguments d'entrée)

{ **Bloc d'instructions**

- ✓ Les arguments d'entrée sont les données nécessaires.
- ✓ Les arguments de sorite sont le résultat attendu
- ✓ On peut avoir plusieurs arguments de sortie
- ✓ Pour faire appel à cette fonction on écrit le nom avec laquelle est sauvegardée la fonction

Exemple 1 :

Ecrivons une fonction qui nous donne les racines réelles d'une équation du second degré $ax^2 + bx + c$, si elles existent. Ainsi,

- ✓ les variables d'entrée de la fonction seraient a, b, c.
- ✓ Les variables de sortie seraient les deux racines.
- ✓ Une autre variable qui informe l'utilisateur s'il n 'y a pas de racines réelles.

Une mise en œuvre possible de cette fonction se présente comme suit:

```
function [racine1,racine2,condition] = equation_2deg(a,b,c);
% Arguments d'entrée : a, b, c
% Arguments de sortie: racine1 , racine2, condition
delta = b^2 - 4*a*c;
      if delta < 0    % Test si l'équation a des racines réelles
```

```
        condition = -1
        disp('pas de racines réelles')
    else
        condition = 0
        racine1 = (-b + sqrt(delta))/(2*a);
        racine2 = (-b - sqrt(delta))/(2*a);
    end
```

✓ Comme le nom de la fonction est « **equation_2deg »**, il doit être stocké dans un fichier appelé « **equation_2deg.m** »

✓ Pour utiliser cette fonction à partir de MATLAB en vue d'obtenir les racines de l'équation $2x^2 + 4x + 1$ nous tapons ce qui suit :

>> **[r1,r2,c] = equation_2deg (2 , 4 , 1)**

✓ La réponse de matlab sera :

condition =

0

r1 =

-0.2929

r2 =

-1.7071

Remarque :

Si nous pouvons écrire nos propres fonctions MATLAB, nous devons être conscients des différentes fonctions existantes intégrées à MATLAB.

8.3) Programmation avancée des fonctions

Matlab offre plusieurs moyens de vérifier les arguments d'entrée et de sortie d'une fonction :

- ✓ La commande « nargin » renvoie le nombre d'arguments d'entrée d'une fonction.
- ✓ La commande « nargout » renvoie le nombre d'arguments de sorite d'une fonction.
- ✓ La commande « nargchk » vérifie le nombre d'arguments d'entrée.
- ✓ La commande « error » affiche un message d'erreur.
- ✓ La commande « input name » renvoie le nom d'argument d'entrée

9) Quelques applications de MATLAB.

9.1) Introduction

Les méthodes numériques sont couramment utilisées pour résoudre des problèmes mathématiques où il est difficile, voire impossible, d'obtenir des solutions exactes.

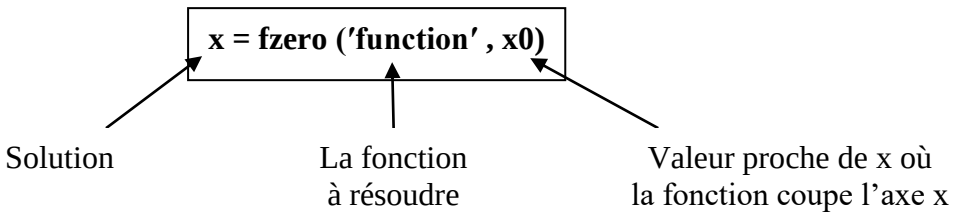
MATLAB possède une grande bibliothèque de fonctions et d'outils qui permet de résoudre une grande variété de problèmes mathématiques. Cette partie du cours explique comment utiliser un certain nombre de ces fonctions les plus fréquemment utilisées. Il est à rappeler que seulement certaines notions sur les méthodes numériques sont données.

9.2) Résolution d'une équation à une variable

Une équation à une variable peut être écrite sous la forme $f(x)=0$. La solution est la valeur de *x* où la fonction traverse l'axe des x, ce qui signifie que la fonction change de signe au point *x*.

Une solution exacte est une valeur de *x* pour laquelle la valeur de la fonction est exactement zéro. Si une telle valeur ne n'existent pas ou est difficile à déterminer, une solution numérique peut être déterminée par trouver un *x* qui est très proche de l'endroit où le fonction change de signe (traverse l'axe des x). Cela se fait par le processus itératif où à chaque itération l'ordinateur détermine une valeur de x qui est plus proche de la solution. Les itérations s'arrêtent lorsque la différence de *x* entre deux itérations est inférieure à une certaine mesure. En général, une fonction peut avoir zéro, un, plusieurs, ou un nombre infini de solutions.

Dans Matlab le zéro d'une fonction peut être déterminé à l'aide de la commande suivante :



❖ Détails supplémentaires sur les arguments de « **fzero** »:

- La fonction à résoudre **:** Elle peut être introduite de 2 façon différentes :
  1. Le moyen le plus simple est d'entrer l'expression mathématique comme une chaîne.

2. La fonction peut aussi être crée comme « function » et définie par l'utilisateur dans un fichier fonction (voir création function), puis le nom de la fonction est entré comme une chaîne.

- La fonction doit être écrit sous une forme standard. Exemple : si la fonction à résoudre est $xe^{-x} = 0.2$, elle doit être écrite comme $f(x) = xe^{-x} - 0.2 = 0$.

  Si cette fonction est entré dans la commande fzero comme une chaîne, elle est entrée comme suit : **' x * exp(-x) – 0.2'**

- Quand une fonction est inscrite comme une chaîne, elle ne peut pas inclure les variables prédéfinies.

  Exemple : si la fonction qui doit être entrée est $f(x) = xe^{-x} - 0.2 = 0$, il n'est pas possible de définir **b = 0.2**, puis entrer **' x * exp(-x) – b'**

- x0 peut être un scalaire ou un vecteur à deux éléments. Si x0 est inscrit comme un scalaire, il doit être un élément au voisinage du point où la fonction traverse l'axe des x.

  Si x0 est entré en tant que vecteur, les deux éléments doivent être des points sur les côtés opposés de la solution telle que f (x0 (1)) a un signe différent de f (x0 (2)).

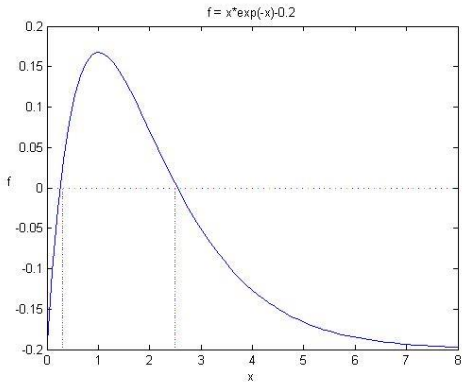
  Quand la fonction a plusieurs solutions, chaque solution peut être déterminée séparément en utilisant la commande fzero en inscrivant des valeurs pour x0 qui sont proches de chacune des solutions.

- Une bonne façon de trouver approximativement où une fonction présente une solution est de faire un tracé de la fonction. Dans de nombreuses applications scientifiques la solution peut être estimée. Souvent, quand une fonction a plus d'une solution seulement une des solutions a un sens physique.

Exemple : Déterminer la solution de l'équation $xe^{-x} = 0.2$

- En première étape l'équation est écrite sous la forme d'une fonction : $f(x) = xe^{-x} - 0.2$ et on représente la fonction graphiquement.
- Une partie du tracé de la fonction montre que f a une solution entre 0 et 1et une autre solution entre 2 et 3

```
>> fplot('x*exp(-x)-0.2',[0 8])
```



Tracé de la fonction $f(x) = xe^{-x} - 0.2$

- Dans la fenêtre de commande les solutions sont trouvées en utilisant la commande fzero pour le 1er point x0 = 0.3 et pour le second point x0 = 2.5.

- Les solutions exactes sont :

```
x1=fzero('x*exp(-x)-0.2',0.3)
x2=fzero('x*exp(-x)-0.2',2.5)
```

Ce qui donne

```
x1 =
    0.2592
x2 =
    2.5426
```

Remarque :

- La fonction fzero trouve une solution à une fonction seulement quand celle-ci coupe l'axe des x ceci dit cette commande ne trouve pas de solution lorsque le tracé de la fonction touche mais ne coupe pas l'axe des x.
- Si une solution ne peut être déterminée NaN est assigné à x.
- La commande fzero dispose d'options supplémentaires (voir la fenêtre Aide). Deux des options les plus importantes sont :
  - **[x fval] = fzero('function', x0)** : affecte la valeur de la fonction en **x** à **fval**
  - **x = fzero('function', x0,optimset('display','iter'))** : affiche le résultat pour chaque itération durant la recherche de la valeur de x.

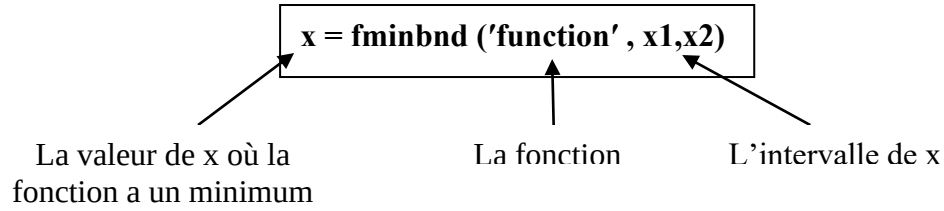
- La commande fzero peut également être utilisé pour trouver la valeur de x où la fonction a une valeur spécifique.

9.3) Recherche d'un minimum ou un maximum d'une fonction

Dans de nombreuses applications, il est nécessaire de déterminer le minimum ou le maximum d'une fonction du type $y = f(x)$.

La valeur de x est déterminée d'abord en trouvant la valeur qui annule la dérivée de la fonction. Ensuite, la valeur de y est déterminée par substitution de la valeur trouvé x dans la fonction.

En Matlab la recherche de la valeur de x où une fonction à une variable f (x) présente un minimum dans l'intervalle $x_1 < x < x_2$ peut être déterminée avec la commande « fminbnd » dont la syntaxe est la suivante:



- La fonction peut être saisie comme une chaîne, comme le nom du fichier de la fonction de la même manière que pour la commande fzero.

- Une fois le minimum calculé, la valeur de *f(x)* correspondante peut être déterminée à l'aide de la commande :
  [x fval] = fminbnd('function', x1,x2) : affecte la valeur de la fonction en **x** à **fval**

- Lorsque la commande « fminbnd » est exécuté, Matlab tente de trouver un minimum local (un point où la pente de la fonction est égal à zéro) dans l'intervalle qui est entré pour x. Si un minimum local n'est pas trouvé, la fonction renvoie le dernier point de (x1 ou x2) correspondant à la valeur la plus faible de la fonction.

Exemple : Soit la fonction $f(x) = x^3 - 12x^2 + 40.25x - 36.5$

- Traçons le graphe de f dans l'intervalle $0 \le x \le 8$ et cherchons le minimum pour *f*

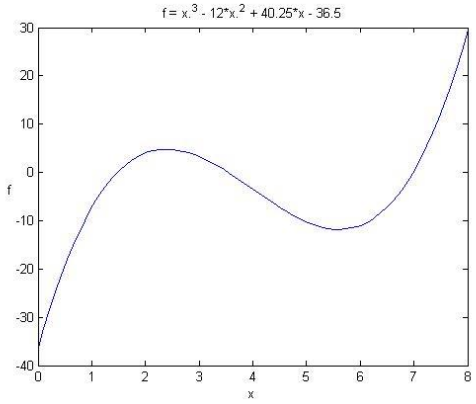


Fig. Tracé de la fonction $f(x) = x^3 - 12x^2 + 40.25x - 36.5$

- On peut remarquer qu'un minimum se situe entre 5 et 6 utilisons la commande fminbnd pour chercher cette valeur.

```
>> [x fval]=fminbnd('x.^3 - 12*x.^2 + 40.25*x - 36.5',5,6)
```

Ce qui donne :

x =

5.6073

fval =

-11.8043

Le minimum est donc au point x = 5.6073. La valeur de la fonction correspondant à ce point est -11.8043

- La commande fminbnd peut également être utilisé pour trouver le maximum d'une fonction. Cela se fait en multipliant la fonction par -1 puis en cherchant le minimum.

Exemple : Cherchons le maximum de la fonction $f(x) = xe^{-x} - 0.2$ dans l'intervalle [0,8]. Ceci se fait en multipliant f par -1. On écrit alors

```
>> [x fval]=fminbnd('-x.*exp(-x) + 0.2',0,8)
```

Ce qui donne

x =

1.0000

fval =

-0.1679

9.4) Interpolation linéaire et non linéaire

La commande « interp1(x,y,z) renvoie un vecteur de même dimension que z dont les valeurs correspondent aux images des éléments de z déterminés par interpolation sur x et y. La syntaxe est : **f = interp1(x,y,z,′type′)**

La chaîne ′type′ spécifie un type d'interpolation parmi les suivants :

′linear′ : interpolation linéaire

′spline′ : interpolation par splines cubiques

′cubic′ : interpolation cubique.

Remarque : Si l'on ne spécifia pas de type, l'interpolation linéaire est prise par défaut.

Exemple : Visualisons les différents types d'interpolation sur les valeurs discrètes de la fonction cosinus.

```
% Partie du script commune aux 3 interpolations
  x = 0:10;
  y = cos(x);
  z = 0:0.25:10; % le pas du vecteur z est inférieur à ce lui de x

% Interpolation linéaire
  f = interp1(x,y,z);
  plot(x,y,'o',z,f), grid
  title('interpolation linéaire')
```

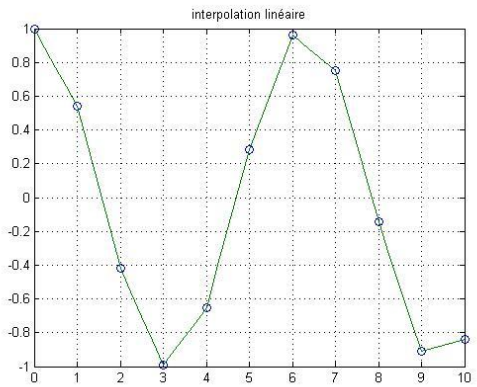


Fig. Interpolation linéaire

```
% Interpolation par splines cubiques
  f = interp1(x,y,z,'spline');   % ou bien  f = spline(x,y,z)
  figure
  plot(x,y,'o',z,f), grid
  title('interpolation par splines cubique')
```

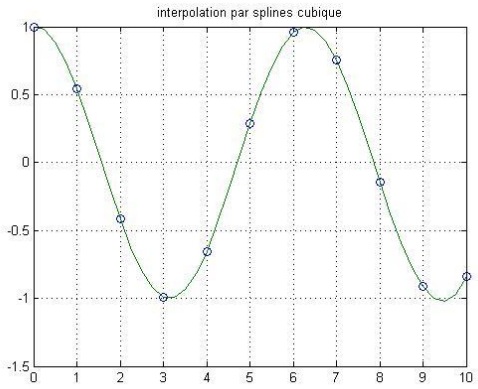


Fig. Interpolation par splines cubique

Remarque : l'interpolation par splines cubique peut être obtenue par invocation de la commande : **spline(x,y,z)**

```
% Interpolation par cubique
  f = interp1(x,y,z,'cubic');
  figure
  plot(x,y,'o',z,f), grid
  title('interpolation par cubique')
```

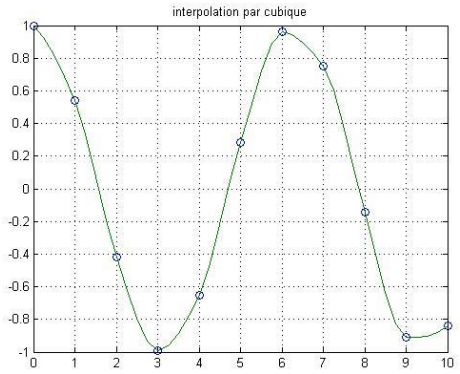


Fig. Interpolation par cubique

9.5) Interpolation au sens des moindres carrés

La commande **p = polyfit (x, y, n)** renvoie le polynôme p de degré n permettant d'approcher la courbe $y = f(x)$ au sens des moindres carrés. Le résultat permet à l'aide d'une courbe approximative d'approcher une courbe expérimentale.

D'une autre manière la commande estime les coefficients du polynôme **p** de degré **n** qui correspondent aux meilleures données de **y** au sens des moindres carrés.

Le résultat est un vecteur ligne de longueur **n +1** contenant les coefficients du polynôme d'ordre décroissant de la forme :

$$p(x) = p_1 x^n + p_2 x^{n-1} + p_3 x^{n-2} + \ldots + p_n x + p_{n+1}$$

Afin de déduire l'erreur entre la courbe expérimentale et le modèle obtenu, on dispose de la fonction **polyval (p, x)** qui renvoie la valeur du polynôme p pour toutes les composantes du vecteur ou de la matrice x.

Exemple : Pour expliquer l'application des fonctions on va simuler une courbe expérimentale à laquelle on superpose un bruit et on va l'approximer par un polynôme d'ordre **1** c'est-à-dire par une droite d'équation : $p(x) = p_1 x + p_2$

```
x = -5:0.1:5;
y = 1./(1+exp(-x)) + 0.05*rand(1,length(x));  % fonction + bruit
plot(x,y,'r')
title('fonction bruitée')

p = polyfit(x,y,1);          %polynôme d'interpolation d'ordre 1
disp('polynôme d''interpolation p(x)= p1x + p2'),p

polynome = polyval(p,x);     %valeurs du polynôme d'interpolation

hold on
plot(x,polynome,'-.')        %tracé du polynôme d'interpolation

erreur = y - polynome;       %calcul de l'erreur d'interpolation
plot(x,erreur,':')           %tracé de la courbe de l'erreur
grid, hold off
```

On obtient les résultats suivants :

```
p =   0.1304    0.5241
```

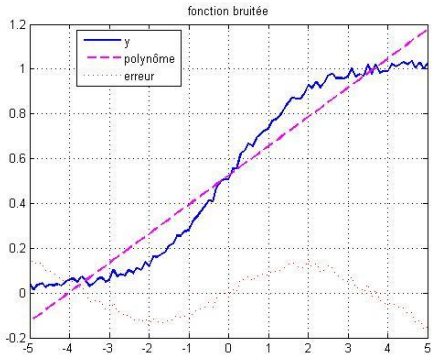


Fig. Interpolation au sens des moindres carrés
par un polynôme d'ordre 1

❖ Faisons maintenant une approximation par un polynôme d'ordre **3** puis d'ordre **5** c'est-à-dire par des polynômes d'équations : $p(x) = p_1x^3 + p_2x^2 + p_3x + p_4$

et $p(x) = p_1x^5 + p_2x^4 + p_3x^3 + p_4x^2 + p_5x + p_6$

```
>> p = polyfit(x,y,3);              %polynôme d'interpolation d'ordre 3
```

p = -0.0043 -0.0000 0.1955 0.5231

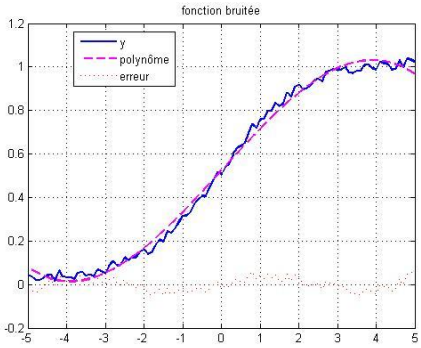


Fig. Interpolation par un polynôme d'ordre 3

```
>> p = polyfit(x,y,3);              %polynôme d'interpolation d'ordre 3
```

p = 0.0002 -0.0001 -0.0102 0.0014 0.2299 0.5215

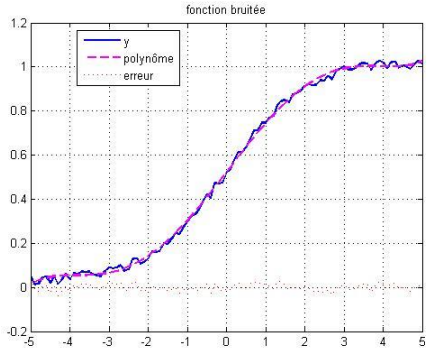


Fig. Interpolation par un polynôme d'ordre 5

9.6) Optmisation

Au paragraphe 9.3 nous avons abordé la minimisation, on appelle généralement optimisation une minimisation sous contraintes. Pour traiter ce type de problème avec Matlab, il faut avoir à disposition « la toolbox Optimisation ». La toolbox inclut des fonctions pour de nombreux types d'optimisation, y compris :

- La minimisation non linéaire sans contrainte
- La minimisation non linéaire avec contrainte
- La résolution des systèmes d'équations non linéaires

❖ **minimisation sans contraintes « fminunc »:** cette commande trouve le minimum d'une fonction à plusieurs variables sans conditions. La syntaxe est :

x = fminunc(fun, x0) commence à x0 et tente de trouver un minimum local x de la fonction fun. LA fonction FUN ayant pour argument d'entrée x0 renvoie un scalaire

x = fminunc(fun, x0,options) minimise avec des options qui ne sont plus définis par défaut mais remplacé par des valeurs choisis et structurés comme arguments dans la fonction OPTIMSET

[x , fval] = fminunc(fun, x0,…) renvoie la valeur trouvée de x après évaluation dans fval.

[x , fval , exitflag] = fminunc(fun, x0,…..) renvoie un EXITFLAG qui décrit la condition de sortie de fminunc.

[x , fval , exitflag , output] = fminunc(fun, x0,…..) renvoie une structure de sortie avec le nombre d'itérations effectuées, le nombre des évaluations de la fonction, l'algorithme utilisé pour la minimisation, …

Exemple : Soit le problème qui consiste à trouver un ensemble de valeurs [x1, x2] qui résout :

$$\min_x imiser\ f_2(x) = e^{x1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right)$$

Pour résoudre ce problème à deux dimensions, on écrit un script qui renvoie la valeur de la fonction $f(x)$, Ensuite invoquer la commande de minimisation sans contrainte « **fminunc** »

Etape 1 : Ecrivons le script pour la valeur de la fonction

```
function f = fonction (x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
```

Etape 2: utilisons fminunc dans l'optimisation sans contraintes

```
x0 =[-1 , 1];
options = optimset('LargeScale','off');
[x,fval,exitflag,output] = fminunc(@fonction,x0,options)
```

- Après plusieurs itérations Matlab renvoie :

  x =

  0.5000 -1.0000

- La valeur de la fonction pour la solution x est donnée dans fval

  fval =

  3.6609e-015

- exitflag indique si l'algorithme de convergence. exitflag = 1 ce qui signifie qu'un minimum local a été trouvé.

  exitflag =

  1

- La structure de sortie donne plus de détails sur l'optimisation. Pour fminunc cette structure comprend le nombre d'itérations, le nombre de fonctions évaluations « funcCount », et le type d'algorithme utilisé.

  output =

  iterations: 8

funcCount: 66

algorithm: medium-scale: Quasi-Newton line search

Remarque :

- Lorsque plus d'un minimum local existe, la proposition initiale pour le vecteur [x1, x2] affecte à la fois le nombre d'évaluations de fonctions et la valeur de la solution. Dans l'exemple précédent, x0 est initialisé à [-1,1].

❖ **minimisation avec contraintes « fmincon »:** cette commande trouve le minimum d'une fonction à plusieurs variables avec des conditions de la forme

$\min_x imiser\ f(x)$ avec les conditions : $Ax \le B$, $A_{eq}x = B_{eq}$ (contraintes linéaires)

$C(x) \le 0$, $C_{eq}(x) \le 0$ (contraintes non linéaires)

$l_{\inf} \le x \le l_{\sup}$

La syntaxe est :

x = fmincon(fun , x0 , A , B) commence à x0 et trouve un minimum x à la fonction fun, sous réserve des inégalités linéaires $Ax \le B$.

x = fmincon(fun , x0 , A , B , A_{eq} , B_{eq}) trouve un minimum x à la fonction fun, sous réserve des inégalités linéaires $A_{eq}x = B_{eq}$ ainsi que $Ax \le B$.

(Remarque : Utiliser des matrices vides pour A et B s'il n'existent pas d'inégalités) A=[] et B=[].

x = fmincon(fun , x0 , A , B , A_{eq} , B_{eq} , l_{inf} , l_{sup}) définit des bornes inférieur et supérieur de sorte qu'une solution soit trouvée dans l'intervalle borné $l_{\inf} \le x \le l_{\sup}$. Utilisez des matrices vides pour $l_{\inf}$ et $l_{\sup}$ si des limites n'ont pas été définies.

[x , fval] = fmincon(fun , x0,…) renvoie la valeur trouvée de x après évaluation dans fval.

[x , fval , exitflag] = fmincon(fun, x0,…..) renvoie un EXITFLAG qui décrit la condition de sortie de fmincon.

[x , fval , exitflag , output] = fmincon(fun, x0,…..) renvoie une structure de sortie avec le nombre d'itérations effectuées, le nombre des évaluations de la fonction, l'algorithme utilisé pour la minimisation, …

Exemple 2: Reprenons l'exemple précédent et utilisons les conditions suivantes :

$$\min_x imiser\ f_2(x) = e^{x1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right)$$

$$\begin{cases} x_1x_2 - x_1 - x_2 \le -1.5 \\ x_1x_2 \ge -10 \end{cases}$$

Etape 1: Le fichier de la fonction reste le même

Etape 2 : Puisque aucune contrainte n'est linéaire, on doit donc créer un fichier function pour ces contraintes. Récrivons les contraintes sous la forme $C(x) \le 0$ cité précédemment :

$$\begin{cases} x_1x_2 - x_1 - x_2 + 1.5 \le 0 \\ -x_1x_2 - 10 \le 0 \end{cases}$$

```
function [c,ceq] = contraintes(x)

c = [x(1)*x(2)- x(1)- x(2) + 1.5; % Contraintes Inégalités non linéaires
     -x(1)*x(2) - 10 ];

ceq = [];              % Contraintes égalités non linéaires
```

Etape 3 : Utilisons la commande fmincon.

```
x0=[-1 , 1];

options = optimset('LargeScale','off');

[x,fval] = fmincon(@fonction,x0,[],[],[],[],[],[],@contraintes,options)
```

- Après plusieurs itérations Matlab renvoie :

```
x =
    -9.5474    1.0474
```

- La valeur de la fonction pour la solution x est donnée dans fval

```
fval =
    0.0236
```

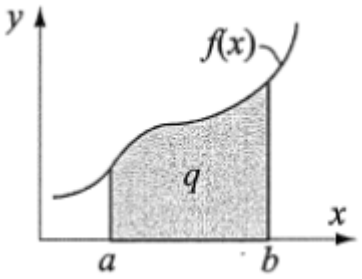
On peut évaluer les contraintes pour les solutions :

```
>> [c,ceq] = contraintes(x)
```

Ce qui donne :

```
c =
   1.0e-007 * (-0.9032)
   1.0e-007 * (0.9032)

ceq =
     [ ]
```

9.7) Intégration numérique



L'intégrale d'une fonction *f(x)* entre a et b est définie par : $q = \int_a^b f(x)\,dx$ où a et b sont les bornes d'intégration.

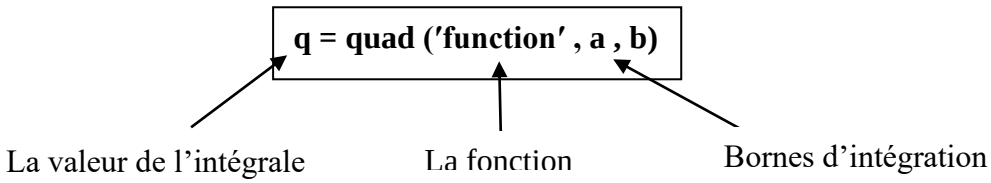
Graphiquement, la valeur de l'intégrale q est l'aire délimitée par le graphe de la fonction, l'axe des x et les limites a et b (la zone ombrée sur la figure).

Quand une intégrale définie est calculé analytiquement *f (x)* est toujours une fonction, lorsque l'intégrale est calculée numériquement *f (x)* peut être une fonction ou un ensemble de points. L'intégration numérique d'une surface totale est obtenue en divisant cette surface en petites sections, on calculer l'aire de chaque section, et on les additionne. Différentes méthodes numériques ont été développées à cet effet. La différence entre ces méthodes réside dans la façon dont la surface est divisée en sections et dans la méthode de calcul de l'aire de chacune de ces sections. Dans cette partie du cours on décrira les trois commandes intégrées à Matlab **« quad, quadl et trapz »**.

Les commandes quad et quadl sont utilisées lors de l'intégration d'une fonction f(x) et trapz lorsque f(x) est représentée par des points.

9.7.1) La commande « quad »

La syntaxe de la commande quad qui est basée sur la méthode adaptative Simpson est :



- La fonction peut être saisie comme une chaîne ou bien comme le nom du fichier de la fonction.

- La fonction f (x) doit être écrite avec l'argument x qui est un vecteur, cette fonction calcule la valeur pour chaque élément de x.

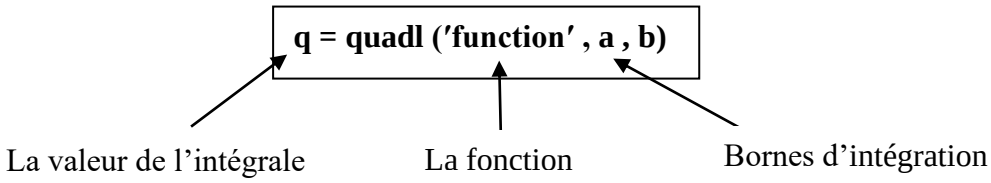
- quad calcule l'intégrale avec une erreur absolue qui est inférieure à 1.0e-6. Cette valeur peut être modifié par l'ajout d'un argument optionnel « tol » à la commande.

  q = quad (′function′ , a , b , tol)

  tol (tolerance) est une valeur qui définit l'erreur maximale. Si on augmente la valeur de l'erreur l'intégrale est calculée rapidement mais avec moins de précision.

9.7.2) La commande « quadl »

La syntaxe de la commande quadl (la dernière lettre est un L minuscule) est exactement la même que celle de la commande quad :



- Tous les commentaires cités ci-dessus pour la commande quad sont valables pour la commande quadl. La différence entre les deux commandes réside dans la méthode numérique utilisée pour le calcul de l'intégration. La commande quadl utilise la méthode d'intégration numérique de Gauss-Lobatto, qui peut être plus efficace pour des précisions élevées.

Exemple : Intégration numérique d'une fonction

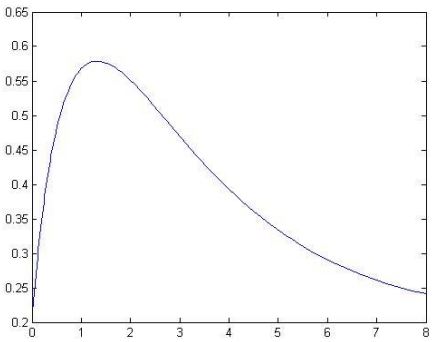
Calculer l'intégrale suivante : $q = \int_0^8 \left(xe^{-x^{0.8}} + 0.2 \right) dx$

- La première méthode consiste à écrire directement la fonction dans la ligne de commande de quad, à titre d'illustration on fait un polt de la fonction dans l'intervalle $0 \le x \le 8$

```
fplot('x*exp(-x^0.8)+0.2',[0 8])
q=quad('x.*exp(-x.^.8)+0.2',0,8)
```

Matlab renvoie la réponse :

```
q =
    3.1604
```



- La deuxième méthode consiste à créer une function qui calcule la fonction à intégrer puis faire appel à celle-ci dans la commande quad

```
% création de la function
function y = exe_integ(x)
y = x.*exp(-x.^.8)+0.2

>> % on fait appel à la fonction dans la commande quad
>> q = quad('exe_integ',0,8)
```

Matlab renvoie la même réponse :

```
q =
   3.1604
```

9.7.3) La commande « trapz »

La commande « trapz » peut être utilisée pour intégrer une fonction définie par des points de données. Elle utilise la méthode numérique d'intégration trapézoïdale. La syntaxe de la commande est :

```
q = trapz (x , y)
```

Où : x et y sont des vecteurs avec les coordonnées x et y de tous les points. Les deux vecteurs doivent être de même longueur.

9.8) Equations différentielles ordinaires

Les équations différentielles jouent un rôle crucial dans les sciences et l'ingénierie, car ils sont le fondement de presque tous les phénomènes physiques qui sont impliqué dans les applications d'ingénierie. Seul un nombre limité d'équations différentielles peuvent être résolues de manière analytique. D'une autre part, les méthodes numériques peuvent donner une solution approchée à presque toutes les équations.

Néanmoins, l'obtention d'une solution numérique n'est pas une tâche facile. En effet, une méthode numérique qui peut résoudre n'importe quelle équation n'existe pas.

Il existe ainsi de nombreuses méthodes de résolution de différents types d'équations. MATLAB possède une grande bibliothèque d'outils qui peuvent être utilisés pour résoudre des équations différentielles. Pour utiliser et profiter pleinement de la puissance de MATLAB, les utilisateurs doivent avoir une bonne connaissance des équations différentielles et les différentes méthodes numériques qui peuvent être utilisés pour les résoudre.

Cette section décrit en détail comment utiliser MATLAB pour résoudre une équation différentielle ordinaire du premier ordre. Les méthodes numériques qui peuvent être utilisés pour résoudre une telle équation sont mentionnées et décrites en termes généraux, mais ne sont pas expliquées à partir d'un point de vue mathématique.

9.8.1) Résolution numérique des équations différentielles ordinaires

Cette section fournit des informations pour résoudre de simples équations différentielles du 1er ordre, cependant, cette solution fournit la base pour résoudre les équations différentielles d'ordre supérieur et des systèmes d'équations.

Une équation différentielle ordinaire (ODE) est une équation qui contient une variable indépendante, une variable dépendante, et des dérivés de la variable dépendante.

Considérons l'équation du premier ordre dont la forme est :

$$\frac{dy}{dx} = f(x, y)$$

Une solution est une fonction y = f (x) qui satisfait l'équation. En règle générale, de nombreuses fonctions peuvent satisfaire une EDO et plus l'information est nécessaire pour déterminer la solution exacte d'un problème spécifique. L'information additionnelle est la valeur de la fonction (la variable dépendante) à une valeur de la variable indépendante.

Etapes à suivre pour résoudre une EDO du 1er ordre

- **Etape 1 :** Ecrire le problème sous une forme standard

  Ecrire l'équation de la forme :

  $$\frac{dy}{dx} = f(x, y) \quad \text{pour} \quad x_0 \le x \le x_f \qquad \text{avec} \qquad y = y_0 \quad \text{pour} \quad x = x_0$$

  Exemple :

  $$\frac{dy}{dx} = \frac{x^3 - 2y}{x} \quad \text{pour} \quad 1 \le x \le 3 \qquad \text{avec} \qquad y = 4.2 \quad \text{pour} \quad x = 1$$

- **Etape 2** : Créer un fichier function qui calcule $\frac{dy}{dx}$ pour des valeurs données de *x* et *y*.

  Pour l'exemple de l'étape 1 on crée le fichier function suivant:

```
function dy = equ_diff(x,y)
dy = (x^3-2*y)/x ;
```

- **Etape 3** : Sélectionner une méthode de résolution

  Sélectionnez la méthode numérique que vous souhaitez que MATLAB utilise dans la solution. Plusieurs méthodes numériques ont été développées pour résoudre les équations différentielles ordinaires du 1er ordre dont plusieurs sont disponibles comme des fonctions intégrées dans MATLAB.

  Le tableau suivant énumère sept commandes ODE intégrés dans MATLAB et qui peuvent être utilisés pour résoudre une équation différentielle du 1er ordre. Une brève description de chaque commande ODE est incluse dans le tableau.

Nom de la commande ODE	Description
Ode45	Pour des problèmes faciles. One-step. Mieux vaut l'appliquer en premier essai pour la plupart des problèmes. Elle est basée sur la méthode de Runge-Kutta.
Ode23	Pour des problèmes faciles. One-step. Elle est basée sur la méthode de Runge-Kutta. Elle est souvent plus rapide mais moins précise que ode45.
Ode113	Pour des problèmes faciles. Multistep.
Ode15s	Pour des problèmes difficiles. Multistep. À utiliser si ode45 échoue
Ode23s	Peut résoudre des problèmes que ode15 ne peut pas.
Ode23t	Pour des problèmes modérés.
Ode23tb	Pour des problèmes difficiles. Souvent plus efficace que ode15s

En général, les commandes peuvent être divisés en deux groupes selon leur capacité à résoudre les problèmes difficiles et selon qu'ils utilisent les méthodes d'une ou de plusieurs étapes. Les commandes one-step (une seule étape) utilisent l'information d'un point pour obtenir la solution au prochain point. Les commandes Multi-step (plusieurs étapes) utilisent les informations provenant de plusieurs points précédents pour trouver la solution au prochain point. Il est impossible de savoir d'avance quelle commande la plus appropriée utiliser pour un problème spécifique. Une suggestion consiste à essayer ode45 qui donne de bons résultats pour de nombreux problèmes. Si une solution n'est pas obtenue parce que le problème est complexe, il est suggéré d'essayer la commande ode 15s.

- **Etape 4** : Résoudre l'équation différentielle

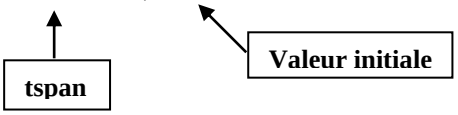
  La syntaxe de la commande pour la résolution est :

[x , y] =nom_de la_commande (′ODEfun′ , tspan , y_0)

- **Informations complémentaires**
  - **nom_de la_commande** : est le nom de la commande ode utilisée (ex : ode45 ou ode23s)
  - **'ODEfun'** : nom du fichier de la fonction qui calcule $\frac{dy}{dx}$ et qui doit être entrée comme chaîne de caractères.
  - **tspan** : Un vecteur qui spécifie l'intervalle de la solution. Ce vecteur doit comprendre au moins deux éléments, mais peut en avoir plusieurs. Si le vecteur n'a que deux éléments, ces derniers doivent être arrangés comme suit [x_0 x_f] qui représente [$val_{initiale}$ val_{finale}]. Ce vecteur peut avoir des points supplémentaires entre [x_0 x_f].
  - **y_0** : la valeur initiale de y (valeur de y au premier point de l'intervalle)
  - **[x , y]** : L'argument de sortie. C'est la solution de l'équation différentielle.

Pour notre exemple on résout l'équation avec ode45 :

```
>>[x,y] = ode45('equ_diff',[1:0.5:3],4.2)
```



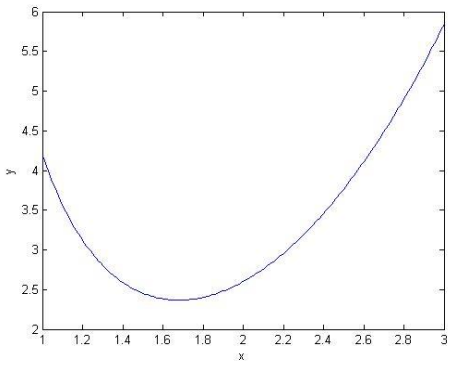
Matlab renvoie :

```
x =
   1
   1.5
   2
   2.5
   3
y =
   4.2000
   2.4528
   2.6000
   3.7650
   5.8444
```

Remarque :

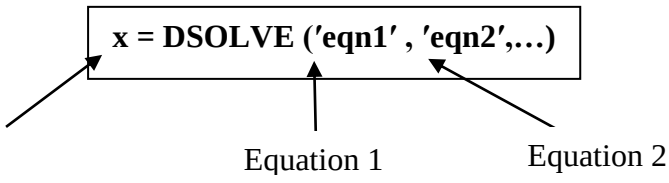
Le pas choisit dans le vecteur tspan est égale à 0.5. Pour voir le comportement de l'équation dans un graphique on résout l'équation différentielle mais cette fois-ci avec un pas plus petit puis on fait un plot de la solution.

```
>> [x,y] = ode45('equ_diff',[1:0.01:3],4.2)
>> plot(x,y),
>> xlabel('x'), ylabel('y')
```



9.8.2) Résolution analytique des équations différentielles ordinaires

Pour la résolution analytique on utilise la commande « **dsolve** ». La syntaxe est la suivante :



La solution

Information complémentaires :

- Dans la commande DSOLVE(′eqn1′,′eqn2′, ...) les équations différentielles ordinaires et les conditions initiales peuvent être représentées par des équations symboliques. Plusieurs équations et/ou plusieurs conditions initiales peuvent être regroupés dans un argument d'entrée unique et séparés par des virgules.

- Par défaut, la variable indépendante est «t». La variable indépendante peut être changé de «t» à une autre variable symbolique en incluant cette variable comme argument d'entrée dernier.
- La lettre **D** désigne la différentielle par rapport à la variable indépendante, c'est-à-dire $\frac{d}{dt}$
- Un « **D »** suivi d'un chiffre indique une différentielle répétée autant de fois que le chiffre indiqué.
- Exemple : **D2** est la représentation symbolique de $\frac{d^2}{dt^2}$.
- Tous les caractères qui suivent immédiatement l'opérateur différentielle sont considérés comme étant les variables dépendantes,

  Exemple : **D3y** désigne la troisième dérivée de y (t) c'est-à-dire $\frac{d^3 y(t)}{dt^3}$.
- Il faut impérativement noter que le nom des variables symboliques ne doit pas contenir la lettre "D".

- Les conditions initiales sont précisées par des équations telles que « **y (a) =b** » ou bien « **Dy (a) =b** » où y est l'une des variables dépendantes et a et b sont des constantes. Si le nombre de conditions initiales donnée est inférieur au nombre de variables dépendantes, les solutions obtenues auront des constantes arbitraires, C1, C2, etc.

- Si aucune solution explicite n'est trouvée, une solution implicite est tentée, pour cela un avertissement est émis indiquant que la solution est implicite. Si aucune solution ne peut être calculée, un avertissement est émis avec et une matrice vide est renvoyée par Matlab pour indiquer la non existence de solution. Dans certains cas impliquant des équations non linéaires, la solution sera une équation différentielle équivalente d'ordre inférieur ou bien une intégrale.

Exemples :

```
% Exemple 1
x = dsolve('Dy = y + sin(t)') % donne comme solution
x =
   -1/2*cos(t)-1/2*sin(t)+exp(t)*C1

% Exemple 2
```

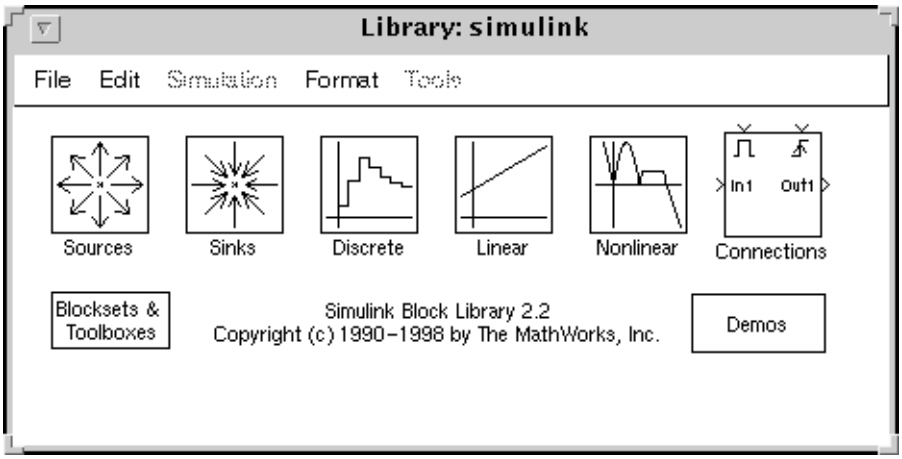
```
z = dsolve('Dy = y+t','y(0)=0') % donne comme solution
z =
   exp(t)-t-1

% Exemple 3
x = dsolve('Dy = a*y', 'y(0) = b') % donne comme solution
x =
   b*exp(a*t)

% Exemple 4
z = dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0')  %donne
z =
   cos(a*t)

% Exemple 5
f = dsolve('Dy -y = exp(-t)*cos(3*t)','y(0)=1')  % donne la solution
f =
  -2/13*exp(-t)*cos(3*t)+3/13*sin(3*t)*exp(-t)+15/13*exp(t)
```

PARTIE 3 : Simulink

Simulink est l'extension graphique de MATLAB permettant de représenter les fonctions mathématiques et les systèmes sous forme de diagramme en blocs, et de simuler le fonctionne- ment de ces systèmes.

➢ POUR DÉMARRER SIMULINK

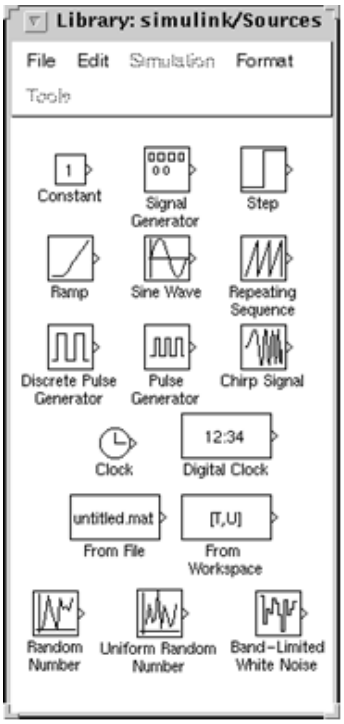
Dans la fenêtre Commande de MATLAB, taper Simulink. La fenêtre Simulink va s'ouvrir.



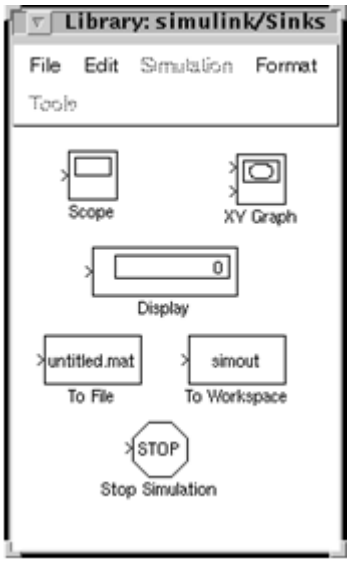
Cette fenêtre contient des collections de blocs que l'on peut ouvrir en cliquant (double) dessus :

Collection de blocs		Signification
Sources	➔	Sources de signaux
Discrete	➔	Blocs discrets
Linear	➔	Blocs linéaires
Nonlinear	➔	Blocs nonlinéaires
Connections	➔	Entrée / sortie, multiplexeur / démultiplexeur, etc.
Demos	➔	Démos
Blocksets & Toolboxes	➔	Boites à outils

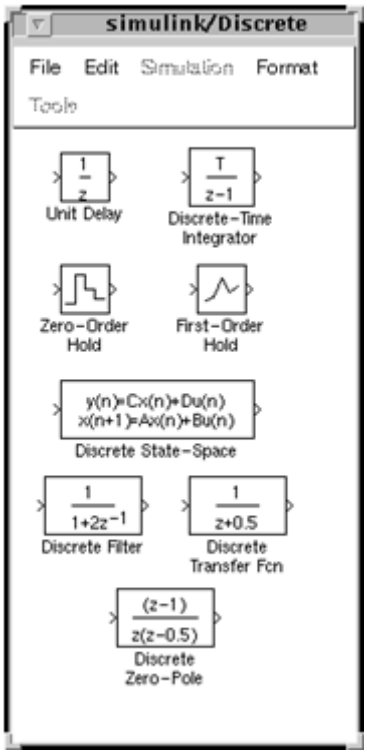
Sources de signaux



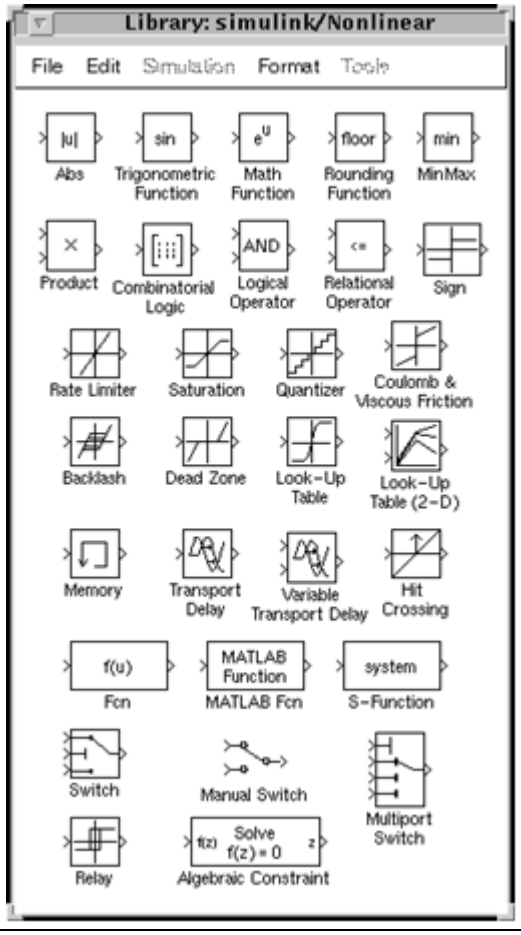
Affichages



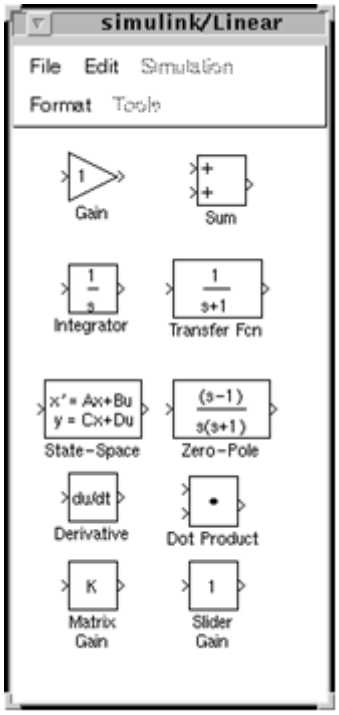
Blocs discrets



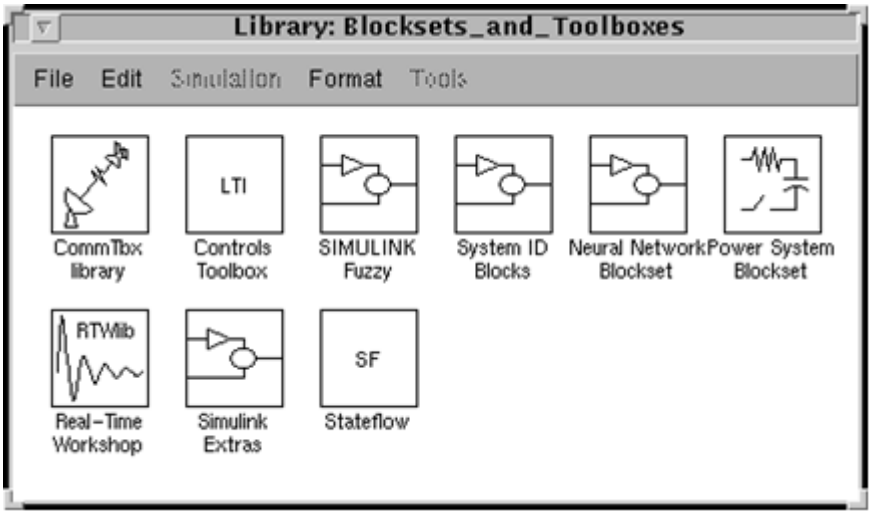
Blocs nonlinéaires



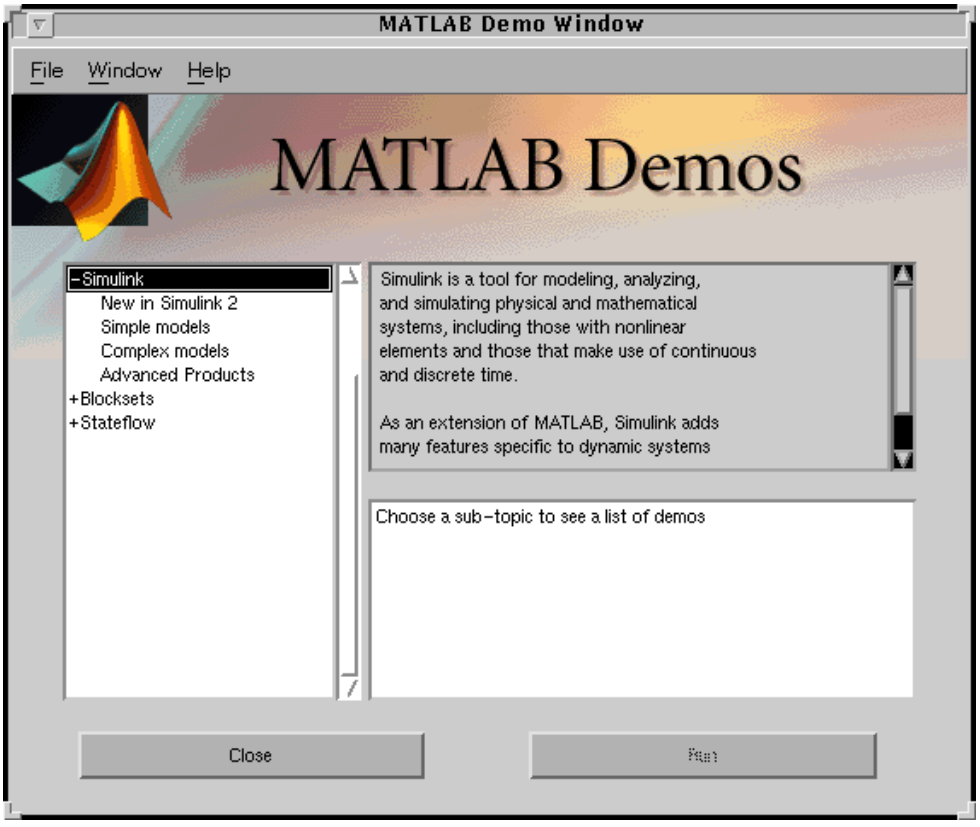
Blocs linéaires



Boites à outils



Démos



Bibliographie :

1. A guide to MATLAB for Beginners and Experienced Users. 2nd edition BRIAN R. Hunt
2. MATLAB Applications for Practical Engineer. Kelly BENNETT
3. Essential MATLAB for Engineers and Scientists. BRIAN Han
4. Matlab version 9.13.0.2126072 (R2022b)