Initiation à Matlab

Nicolas Hudon (nicolas.hudon@polymtl.ca)
URCPC, École Polytechnique de Montréal

22 janvier 2004

Table des matières

1 Introduction **3**

2 Présentation de MATLAB **4**

3 Fichiers SCRIPT et FUNCTION **7**
3.1 Fichiers SCRIPT 8
3.2 Fichiers FUNCTION 9

4 Opérations mathématiques avec MATLAB **11**
4.1 Scalaires, vecteurs, matrices 11
4.2 Graphiques simples 20
4.3 Fonctions mathématiques simples 22
4.3.1 Fonctions mathématiques usuelles 22
4.3.2 Fonctions matricielles 24

5 Programmation avec MATLAB **25**
5.1 Opérateurs logiques 26
5.2 Boucles *if-elseif-else* 27
5.3 Boucles *for* 28
5.4 Boucles *while* 29
5.5 Boucles *switch* 30

6 Fonctions avancées **33**
6.1 Graphiques 33
6.2 Importer et exporter des données 38
6.3 Variables symboliques 40
6.4 Racines d'une équation polynomiale 42
6.5 Régression 43
6.6 Intégration numérique 45
6.7 Solution d'équations différentielles ordinaires avec valeurs initiales 46

7 Références **49**
7.1 Références générales 49
7.2 Références spécifiques au génie chimique 49
7.3 Notes 49

1 Introduction

L'objectif de ce court document est de vous initier au logiciel MATLAB de la compagnie Mathworks et à la programmation dans cet environnement. L'idée est de vous exposer les bases de cet outil de travail et de vous habiliter à résoudre des problèmes de génie chimique avec celui-ci, particulièrement pour les cours suivants :

- GCH2530 Programmation numérique en génie chimique
- GCH3110 Calculs des réacteurs chimiques
- GCH3120 Procédés de séparation
- GCH3130 Commande des procédés de génie chimique I
- GCH3140 Mécanique des fluides appliquée
- GCH3150 Transfert thermique

Le présent document se divise comme suit. Après quelques éléments de base de MATLAB (section 2), vous verrez comment utiliser des fichiers *SCRIPT* et *FUNCTION* (section 3). Ensuite, on introduira les principales opérations usuelles sur les scalaires, les vecteurs et les matrices avec MATLAB ainsi que certaines opérations mathématiques (section 4). La section 5 montre les différentes boucles de programmation en MATLAB. La section 6 présente quelques fonctions plus avancées qui vous seront utiles dans le cadre des cours ci-haut mentionnés. Finalement, on fournit une courte liste de références utiles (livres et sites WEB).

2 Présentation de MATLAB

MATLAB est beaucoup plus qu'un langage de programmation. Il s'agit d'une console d'exécution (*shell*) au même titre que les consoles DOS ou UNIX. Comme toutes les consoles, MATLAB permet d'exécuter des fonctions, d'attribuer des valeurs à des variables, etc. Plus spécifiquement, la console MATLAB permet d'effectuer des opérations mathématiques, de manipuler des matrices, de tracer facilement des graphiques. La figure 1 présente l'écran MATLAB de base.

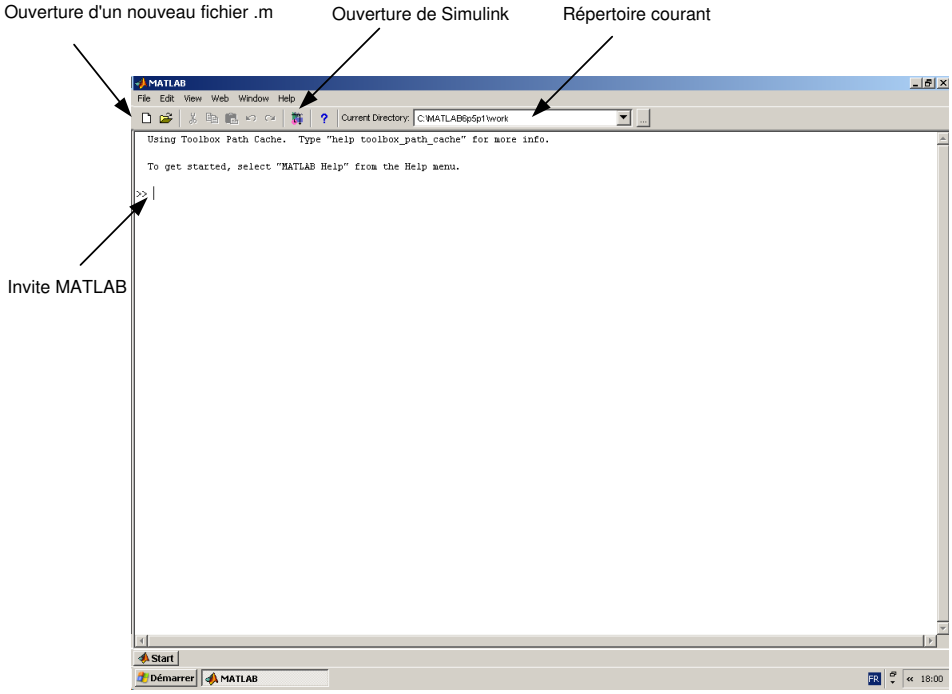

FIG. 1 – Écran MATLAB

SIMULINK n'est pas au programme de cours d'initiation. L'élément le plus important ici est *l'invite MATLAB* où l'utilisateur peut affecter des valeurs à des variables et effectuer des opérations sur ces variables. Par exemple :

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

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

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

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

```
>> x = 4;
>> y = 2;
>> a = x + y
a =
     6
>> b = x * y
b =
     8
>> whos
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 4 elements using 32 bytes
>>
```

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

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

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

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

La sortie de la fonction *whos* donne, entre autre, la classe de la variable. Plusieurs classes de variables sont disponibles à l'utilisateur de MATLAB. Les classes les plus utiles pour l'utilisateur débutant sont le *double* tel que présenté plus haut et les variables *char*, pour le texte, et les variables symboliques, que nous présenteront à la section 6.3. Pour les variables *char*, la déclaration se fait entre apostrophe :

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

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

```
>> mot1 = 'hello';
>> mot2 = 'world';
>> mot1_2 = [mot1 ' ' mot2] % l'emploi de ' ' permet d'introduire un espace
mot1_2 =
hello world
```

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

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

str = ['La racine carrée de ' num2str(a) ' est ' num2str(b)];
                                    % composition de la phrase de sortie
disp(str)      % utilisation de display pour afficher le résultat à l'écran
```

Le résultat serait le suivant :

```
Entrez un nombre: 23  % 23 est entré par l'utilisateur
La racine carrée de 23 est 4.7958   % sortie à l'écran
```

La prochaine section montre comment réaliser de courts programmes comme celui utilisé dans le dernier exemple.

3 Fichiers SCRIPT et FUNCTION

Jusqu'à présent, l'utilisation que nous avons faite de MATLAB s'apparente beaucoup à celle d'une calculatrice. Pour des tâches répétitives, il s'avère beaucoup plus pratique et judicieux d'écrire de courts programmes pour effectuer les calculs désirés. Il existe deux types de fichiers qui peuvent être programmés avec MATLAB : les fichiers **SCRIPT** et **FUNCTION**. Dans les deux cas, il faut lancer l'éditeur de fichier et sauvegarder le fichier avec l'extension *.m*.

3.1 Fichiers SCRIPT

Le fichier SCRIPT permet de lancer les mêmes opérations que celles écrites directement à l'invite MATLAB. **Toutes les variables utilisées dans un SCRIPT sont disponibles à l'invite MATLAB**. Par exemple, le fichier *test.m* qui reprend l'exemple précédent (assurez-vous que le fichier test.m se trouve bel et bien dans le répertoire indiqué dans la fenêtre *Répertoire courant*) :

```
% test.m

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

produit la sortie suivante lorsque qu'appelé :

```
>> test
a =
     6
b =
     8
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 4 elements using 32 bytes
>>
```

Habituellement, on utilise les fichiers SCRIPT afin de :

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

Il est utile ici de noter que le langage MATLAB n'est pas un langage compilé (contrairement au langage *C++*, par exemple). À chaque appel d'un SCRIPT (ou d'une FUNCTION), le logiciel lit et exécute les programmes ligne par ligne. Lorsque MATLAB détecte une erreur, le logiciel arrête et un message d'erreur ainsi que la ligne où l'erreur est détectée s'affichent à l'écran. Apprendre à lire les messages d'erreur est donc important pour "déboguer" vos programmes rapidement et efficacement.

3.2 Fichiers FUNCTION

L'idée de base d'une fonction est d'effectuer des opérations sur une ou plusieurs **entrées** ou **arguments** pour obtenir un résultat qui sera appelé **sortie**. Il est important de noter que les variables internes ne sont pas disponibles à l'invite MATLAB. Par exemple, la fonction suivante (avec une seule sortie, le résultat de l'addition) :

```
function a = ma_function(x,y)
a = x + y;
b = x * y;
```

produit la sortie suivante :

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

Le résultat de la multiplication n'est plus disponible. On peut cependant modifier les sorties de la manière suivante :

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

pour obtenir :

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

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

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

Habituellement, on utilise les fichiers FUNCTION afin de :

- Programmer des opérations répétitives
- Limiter le nombre de variables dans l'invite MATLAB
- Diviser le programme (problème) de manière claire

4 Opérations mathématiques avec MATLAB

Dans cette section, nous présenterons les éléments mathématiques de base de MATLAB.

4.1 Scalaires, vecteurs, matrices

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

Les **scalaires** se déclarent directement, par exemple :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>> V1 = [1 2];
>> V2 = [3 4];
>> V = V1 + V2 % addition de vecteurs
V =
     4     6

>> V = V2 - V1 % soustraction de vecteurs
V =
     2     2

>> V = 2*V1    % multiplication par un scalaire
V =
     2     4
```

Dans le cas de la multiplication et de la division, il faut faire attention aux dimensions des vecteurs en cause. Pour la multiplication et la division élément par élément, on ajoute un point devant l'opérateur (.* et ./). Par exemple :

```
>> V = V1.*V2  % multiplication élément par élément
V =
     3     8

>> V = V1./V2  % division élément par élément
V =
    0.3333    0.5000
```

Cependant, MATLAB lance une erreur lorsque les dimensions ne concordent pas (remarquez les messages d'erreur, ils sont parfois utiles pour corriger vos programmes) :

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

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

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

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

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

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

De même, pour les vecteurs colonnes :

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

On peut aussi créer des matrices, par exemple,

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

qui n'est pas équivalent à :

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

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

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

Les matrices peuvent aussi être construites directement :

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

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

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

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

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

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

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

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

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

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

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

Il est possible d'inverser *inv()*, detransposer *transpose() ou avec l'apostrophe (')* les matrices :

```
>> invM = inv(M)
invM =
   -2.0000    1.0000
    1.5000   -0.5000

>> transpM = M'
transpM =
     1     3
     2     4
```

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

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

>> C = A+B % addition
C =
     5     5
     5     5

>> D = A-B % soustraction
D =
    -3    -1
     1     3

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

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

```
>> C = A*B  % multiplication de matrices
C =
     8     5
    20    13

>> D = A/B  % division de matrices
D =
    1.5000   -2.5000
    2.5000   -3.5000
```

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

```
>> C = A.*B  % multiplication élément par élément
C =
     4     6
     6     4

>> D = A./B  % division élément par élément
D =
    0.2500    0.6667
    1.5000    4.0000
```

D'autres opérations sur les matrices seront présentées dans les sections subséquentes. Il faut noter certaines matrices spéciales qui peuvent être utilisées, par exemple la matrice identité :

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

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

```
>> V_nul = zeros(1,2) % un vecteur de 1 ligne, 2 colonnes de 0
V_nul =
     0     0

>> V_un = ones(1,2) % un vecteur de 1 ligne, 2 colonnes de 1
V_un =
     1     1

>> M_un = ones(2,2) % une matrice 2x2 de 1
M_un =
     1     1
     1     1
```

Dans certaines applications, il est parfois utile de connaître les dimensions d'une matrice, et la longueur d'un vecteur (retournés, par exemple, par une fonction). Dans ce cas, on utilise les fonctions *length* et *size*.

```
>> V = [0:0.1:10]; % utilisation de length - vecteur 1x101
>> n = length(V)

n =
   101

>> M = [1 2 3; 4 5 6]; % utilisation de size - matrice 2x3
>> [n,m] = size(M)
n =
     2
m =
     3

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

Voici quelques exercices sur la manipulation des vecteurs et des matrices :

Exercices

1. On veut vérifier que la multiplication de matrices n'est pas commutative. Soient deux matrices :

$$A = \begin{pmatrix} 3 & 4 & 4 \\ 6 & 5 & 3 \\ 10 & 8 & 2 \end{pmatrix}$$

$$B = \begin{pmatrix} 4 & 5 & 8 \\ 3 & 11 & 12 \\ 2 & 1 & 7 \end{pmatrix}$$

Réalisez un code MATLAB qui permet, pour les matrices A et B données, de vérifier que :

$$A * B - B * A \neq 0$$

2. En utilisant les matrices définies au numéro **1**, vérifiez l'identité suivante :

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

Utilisez la fonction *transpose* de MATLAB pour transposer les matrices.

3. On désire solutionner un système d'équations algébriques linéaires, c'est-à-dire, un système représenté par :

$$\mathbf{A} * \mathbf{x} = \mathbf{b}$$

Avec $\mathbf{A}$, une matrice de dimension nxn, $\mathbf{x}$ et $\mathbf{b}$, des vecteurs colonne de dimension n. La solution de ce système est donné par :

$$\mathbf{x} = \mathbf{A}^{-1} * \mathbf{b}$$

En utilisant la fonction *inv* de MATLAB pour inverser la matrice, solutionnez le système décrit par la matrice A, définie au numéro **1** et le vecteur b, **la première colonne** de la matrice B, de ce numéro.

4.2 Graphiques simples

Cette sous-section ne vise qu'à vous permettre de tracer les solutions pour les exercices qui suivent. Veuillez vous référer à la section 6 pour plus d'informations.

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

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

qui produit la sortie graphique de la figure 2 :

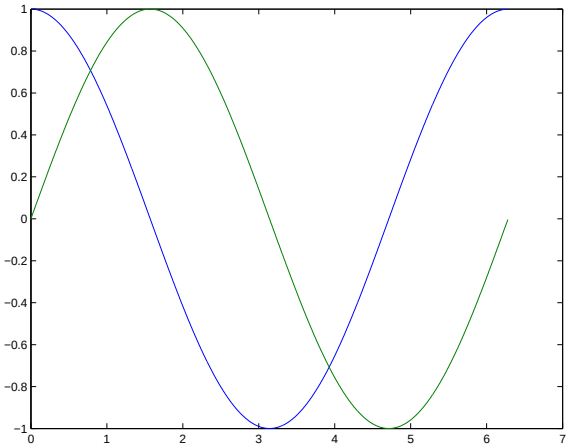

FIG. 2 – sin(x) vs cos(x)

On aurait aussi pu simplifier par :

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

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

```
% graphique.m

clear all
close all % ferme les anciennes figures

x = [0:0.01:2*pi]; y1 = cos(x); y2 = sin(x);

figure(1)
plot(x, y1,'.',x, y2,'+') % cos(x) en points, sin(x) en +
title('sinus et cosinus') xlabel('x') ylabel('f(x)')
legend('cos(x)','sin(x)',0) % le 0 place la légende à côté des courbes
```

Et on obtient la figure 3.

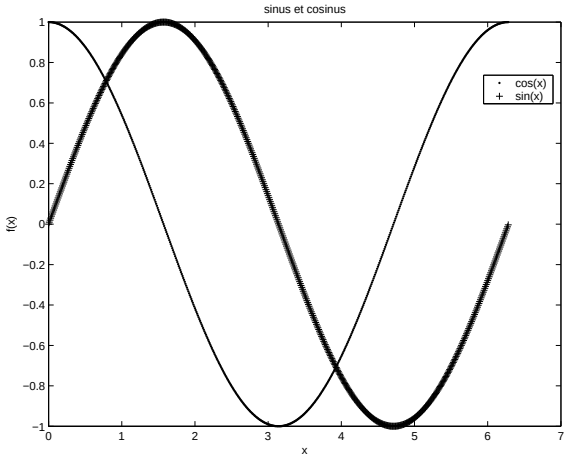

FIG. 3 – sin(x) vs cos(x)-corrigé

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

Fonction	Description
sin(x)	sinus de x ; x en radians
cos(x)	cosinus de x ; x en radians
tan(x)	tangente de x ; x en radians
exp(x)	exponentielle de x
log(x)	logarithme en base e de x
sqrt(x)	racine carrée de x
power(x,a)	puissance a de x
abs(x)	valeur absolue de x
asin(x)	$\sin^{-1}$ de x ; résultat en radians
acos(x)	$\cos^{-1}$ de x ; résultat en radians
atan(x)	$\tan^{-1}$ de x ; résultat en radians
sinh(x)	sinus hyperbolique de x
cosh(x)	cosinus hyperbolique de x
tanh(x)	tangente hyperbolique de x
round(x)	arrondit un nombre à l'entier le plus près
floor(x)	arrondit vers l'entier immédiatement au-dessous
ceil(x)	arrondit vers l'entier immédiatement au-dessus

TAB. 1 – Fonctions courantes en MATLAB

4.3 Fonctions mathématiques simples

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

```
>> help cos

 COS    Cosine.
    COS(X) is the cosine of the elements of X.

 Overloaded methods
    help sym/cos.m
```

Dans un premier temps, on présente les fonctions mathématiques usuelles et leur appel dans MATLAB. Ensuite, on présente les principales fonctions spécifiques aux matrices.

4.3.1 Fonctions mathématiques usuelles

Toutes les fonctions mathématiques de base sont déjà programmées dans MATLAB. Le tableau 1 présente les plus courantes.

Exercices

4. On veut vérifier graphiquement la définition de la fonction $sinh(x)$:

$$sinh(x) = \frac{e^x - e^{-x}}{2}$$

Produisez un code MATLAB qui trace deux courbes sur un même graphique. La première, en utilisant directement la fonction $sinh(x)$ de MATLAB et la deuxième en utilisant la définition avec les exponentielles. Utilisez un vecteur x de -2 à 2 par incréments de 0.1.

5. Vérifiez graphiquement (pour les mêmes valeurs de x qu'au numéro **4** que :

$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$

ATTENTION à la division de deux vecteurs !

Fonction	Description
det(A)	déterminant de la matrice A
trace(A)	trace de la matrice A
rank(A)	rang de A (nombre de colonnes ou de lignes linéairement indépendantes)
norm(A)	norme de A (peut s'appliquer à un vecteur V)
inv(A)	inverse de A
poly(A)	polynôme caractéristique de la matrice A
eig(A)	valeurs propres de la matrice A
svd(A)	décomposition en valeur singulière de la matrice A
min(V)	indice du plus petit él. du vecteur V
max(V)	indice du plus grand él. du vecteur V
sum(V)	somme des élément du vecteur V

TAB. 2 – Fonctions matricielles en Matlab

4.3.2 Fonctions matricielles

Toutes les fonctions matricielles de base sont déjà programmées dans MATLAB. Le tableau 2 présente les plus courantes.

Exercices

6. En utilisant la fonction *eye*, vérifiez que (utilisez la matrice A définie au numéro **1**) :

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

7. En utilisant les fonctions *trace* et *eig* vérifiez que (utilisez la matrice A définie au numéro **1**) :

$$tr\mathbf{A} = \Sigma_{i=1}^{n} \lambda i$$

Où les λ_i sont les valeurs propres de la matrice A.

5 Programmation avec MATLAB

Cette section présente les différentes structures de programmation avec MATLAB. Pour l'étudiant familier avec la programmation C++ ou Fortran, il ne s'agit que de se familiariser avec la syntaxe propre à MATLAB. Pour l'étudiant moins familier avec les structures de base en programmation, les exercices à la fin de la section sont indiqués pour devenir fonctionnels avec ces structures.

Opérateur	Description
$\sim$ a	NOT - retourne 1 si a égal 0, 1 si a égal 0
a == b	retourne 1 si a égal b, 0 autrement
a < b	retourne 1 si a est plus petit que b, 0 autrement
a > b	retourne 1 si a est plus grand que b, 0 autrement
a <= b	retourne 1 si a est plus petit ou égal à b, 0 autrement
a >= b	retourne 1 si a est plus grand ou égal à b, 0 autrement
a $\sim$=b	retourne 1 si a est différent de b, 0 autrement

Tab. 3 – Opérateurs de comparaisons

5.1 Opérateurs logiques

Avant de débuter la présentation des boucles de programmation conditionnelles, il est important de se familiariser avec les opérateurs logiques. Le premier type de ces opérateurs permet de comparer des valeurs entre elles (tableau 3).
Par exemple, on veut comparer deux valeurs entre elles :

```
>> a = sin(2*pi);
>> b = cos(2*pi);
>> bool = (a>b)
bool =
     0
>> a
a =
 -2.4493e-016  % ici a devrait égaler 0, la précision est limitée!
>> b
b =
     1
```

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

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

P	Q	P et Q
1	1	1
1	0	0
0	1	0
0	0	0

Tab. 4 – Table de vérité - opérateur ET

P	Q	P ou Q
1	1	1
1	0	1
0	1	1
0	0	0

Tab. 5 – Table de vérité - opérateur OU

Il est aussi possible de lier entre elles des conditions par l'opérateur 'et' (&) et 'ou' (|). Les tables de vérités de ces deux opérateurs sont présentées aux tableaux 4 et 5.

Ces notions seront utiles pour la construction des conditions des prochaines sections.

5.2 Boucles *if-elseif-else*

Ce type de structure de programmation est très utile pour vérifier des conditions. En pseudo-code, on peut résumer par le schéma suivant :

```
si CONDITION1, FAIRE ACTION1. % condition 1 remplie

sinon et si CONDITION2, FAIRE ACTION2. % condition 1 non-remplie,
                                       % mais condition 2 remplie

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

En MATLAB, le pseudo-code précédent devient :

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

Par exemple, on reçoit un entier a, s'il est impair négatif, on le rend positif. S'il est impair positif, on lui ajoute 1. S'il est pair, on ajoute 2 à sa valeur absolue. La courte fonction suivante permet de réaliser cette transformation (notez ici, l'emploi du modulo pour déterminer si l'entier est divisible par 2).

```
function b = transf_entier(a)

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

5.3 Boucles *for*

Les boucles *for* sont très utiles dans la plupart des applications mathématiques (par exemple, pour effectuer un calcul sur tous les éléments d'un vecteur). En MATLAB, il est parfois beaucoup plus efficace d'utiliser les opérateurs algébriques usuels définis plus tôt (par exemple, le '.*'). Dans les cas où il est impossible de se soustraire à l'utilisation de ces boucles, voici le prototype en pseudo-code de ces boucles.

```
incrément = valeur initiale

Pour incrément=valeur_initiale jusqu'à valeur finale

    ACTION1...N
    AJOUTER 1 à incrément
```

En MATLAB, ce pseudo-code devient :

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

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

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

5.4 Boucles *while*

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

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

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

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

Ce type de boucle est très souvent utilisé pour converger vers une valeur désirée. Par exemple, on veut trouver le nombre d'entiers positifs nécessaires pour avoir une somme plus grande que 100. On pourrait réaliser cette tâche de la manière suivante :

```
function n = nombre_entier

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

5.5 Boucles *switch*

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

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

En MATLAB, on obtient le code suivant :

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

Par exemple, on veut faire une calculatrice simple en MATLAB, pour déterminer l'exponentielle ou le logarithme en base e d'un nombre entré par l'utilisateur. Une manière simple de rendre le programme interactif serait d'utiliser le SCRIPT suivant :

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

switch operation
    case 1
        b = exp(nombre)
    case 2
        b = log(nombre)
    otherwise
        disp('mauvais choix -- operation')
end
```

Avec la sortie (par exemple) suivante :

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

b =
    1.6487
>>
```

Exercices

8. En utilisant la fonction *xor* de MATLAB, construisez la table de vérité de l'opérateur OU EXCLUSIF.

9. On veut comparez le temps de calcul nécessaire à MATLAB avec une boucle *for* versus le temps nécessaire avec une opération vectorielle. Pour ce faire, on utilise les commandes *tic* et *toc* :

```
>> tic
>> OPÉRATION
>> toc
```

MATLAB affiche alors le temps utilisé pour réaliser l'opération. Dans un fichier SCRIPT, effectuez le calcul suivant :

$$x(t) = \frac{e^t}{1 + e^t}$$

Utilisez le vecteur [0 : 0.001 : 1]. Dans un premier temps, solutionnez à l'aide des opérateurs vectoriels. Ensuite, utilisez la boucle *for* :

```
clear x
for i=1:length(t) % incrément dans tout le vecteur t
    x(i) = exp(t(i))/(1 + exp(t(i));
end
```

Essayez ensuite en mettant la ligne *clear x* en commentaire.

10. Soit l'équation quadratique suivante :

$$ax^2 + bx + c = 0$$

On désire déterminer, pour des paramètres a, b et c donnés, de quels types sont les racines de x. Il s'agit, selon la valeur du discriminant, d'afficher à l'utilisateur le type des racines (réelles identiques, réelles distinctes, complexes). Pour l'affichage, utilisez :

```
message = (['Les racines de l'équation sont ...']);
```

Utilisez une fonction qui prend les paramètres a, b et c comme argument et qui renvoie le diagnostic.

11. La fonction arctan(x) peut être approximée par :

$$tan^{-1}(x) \approx \Sigma_{m=0}^{M} \frac{(-1)^m x^{2m+1}}{2m+1} (|x| < 1)$$

On désire savoir le nombre M qui permet, pour une valeur de x donnée, d'approximer avec une erreur inférieure à 10^{-6}. Réalisez une fonction MATLAB qui prend comme argument, le nombre x, qui vérifie si la condition est remplie et, à l'aide d'une boucle *while*, détermine le nombre d'itérations M. Le programme doit lancer un message d'erreur si la condition n'est pas remplie.

12. La méthode de Newton-Raphson permet de déterminer les racines d'une équation algébrique non-linéaire. L'algorithme est le suivant :

```
0. Poser x(0)   % estimé initial
1. Tant que f(k) - 0 plus grand qu'un critère
    1.1. Calculer x(k+1) = x(k) - f(x_k)/f'(x_k)
    1.2. k = k + 1
2. Sortie
```

À l'aide de cet algorithme, déterminer les racines de l'équation :

$$2x - 2^x = 0$$

Vérifiez l'effet de l'estimé initial sur le nombre d'itérations.

6 Fonctions avancées

Les différentes parties de cette section sont plus spécifiques aux différents cours (sauf les sous-sections 6.1 et 6.2 qui traitent des sorties graphiques et des entrées-sorties dans des fichiers). Quelques exercices sont présentés dans chacune des sous-sections.

6.1 Graphiques

On a vu précédemment comment faire des figures simples à l'aide de la fonction *plot*. On peut aussi faire plusieurs fenêtres graphiques sur une même figure (dans le cas, par exemple, où l'on voudrait tracer deux courbes d'ordres de grandeur différents). La figure 4 présente un exemple à ne pas faire, soit de tracer la température et la concentration en fonction du temps dans un réacteur sur la même figure.

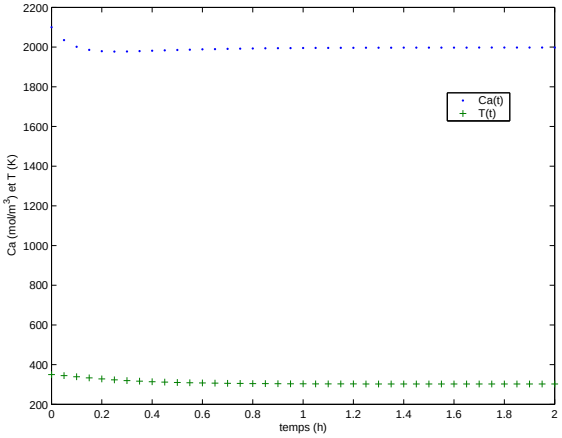

FIG. 4 – C(t) et T(t) sur une même figure

La figure 5 présente les mêmes données, mais dans deux fenêtres graphiques :

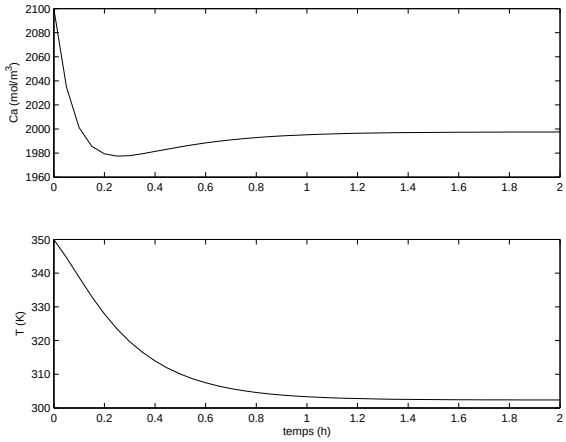

FIG. 5 – C(t) et T(t) sur une même figure - corrigé

Le code utilise le concept de *figure* et de *subplot*, tel que demontré dans le code suivant :

```
figure(1) % déclaration d'une figure.

subplot(2,1,1) % on déclare une matrice de graphiques
               % de 2 lignes (indice1), 1 colonne (indice2)
               % et le premier plot est dans la première fenêtre
plot(t,Ca)
ylabel('Ca (mol/m^3)')

subplot(2,1,2)   % 2 lignes, 1 colonne, 2e graphique
plot(t,T)
ylabel('T (K)')
xlabel('temps (h)')
```

Fonction	Description
semilogx	axe des x logarithmique
semilogy	axe des y logarithmique
loglog	axes des x et des y logarithmique
polar	graphique en coordonnées polaires
mesh	graphique 3D

Tab. 6 – Fonctions de graphiques

Dans l'utilisation de *subplot*, MATLAB compte les fenêtres de graphique comme dans le cas des matrices (en complétant chaque ligne).

Il existe plusieurs fonctions pour tracer certaines formes spéciales de graphique. Une synthèse de ces fonctions est présentée au tableau 6.
Les exercices qui suivent permettront de tester l'utilisation de quelques'unes de ces fonctions. Dans le doute au sujet des appels de ces fonctions, n'hésitez pas à utiliser le *help*...

t (min)	C_A (mol L^{-1})
1	1.00
2	0.90
3	0.80
4	0.73
5	0.67
6	0.62
7	0.58
8	0.55
9	0.53
10	0.52

TAB. 7 – Données d'expérimentation

Exercices

13. Une expérience dans un réacteur *batch* sont présentées au 7.
Sur une même figure, faites tracez les graphiques suivants :

a - $C_A(t)$ en fonction du temps ;

b - $\frac{-dC_A(t)}{dt}$ en fonction de $C_A(t)$;

c - $\frac{ln(-dC_A(t))}{dt}$ en fonction de $ln(C_A(t))$ sur un graphique *loglog* ;

Évaluez les dérivées numériquement (pour les points i = 1...10) par :

$$\left(\frac{dC_A(t)}{dt}\right)_{i=1} \approx \frac{-3C_{A1} + 4C_{A2} - C_{A3}}{2\Delta t}$$

$$\left(\frac{dC_A(t)}{dt}\right)_{i=2...9} \approx \frac{C_{A(i+1)} - C_{A(i-1)}}{2\Delta t}$$

$$\left(\frac{dC_A(t)}{dt}\right)_{i=10} \approx \frac{C_{A8} - 4C_{A9} + 3C_{A10}}{2\Delta t}$$

14. On veut tracer avec MATLAB la fonction

$$z(x, y) = e^{-0.5[x^2+0.5(x-y)^2]}$$

pour x et y entre -4 et 4. À l'aide de deux boucles *for* imbriquées, calculez la valeur de la fonction à chaque point du domaine (utilisez des incréments de 0.05 dans les deux directions). Tracez ensuite la courbe obtenue à l'aide de la fonction *mesh(vecteurX, vecteurY, matriceZ)*. Vérifiez l'appel à cette fonction en tapant *help mesh* à l'appel MATLAB. Tracez ensuite les mêmes valeurs à l'aide des fonctions *surf(vecteurX, vecteurY, matriceZ)* et *contour((vecteurX, vecteurY, matriceZ)*.

6.2 Importer et exporter des données

Dans plusieurs cours, il vous sera nécessaire de conserver vos solutions numériques d'une session de travail à l'autre. La méthode la plus facile est d'utiliser les fonctions *save* et *load* de MATLAB. Les fichiers créés porte l'extension '.mat' par défaut (ceux-ci ne sont accessibles qu'avec MATLAB). Voici un exemple d'utilisation simple.

```
>> A = ones(3);    % matrice
>> b = zeros(3,1); % vecteur
>> save ma_session  % le fichier ma_session.mat est créé dans
                    % le répertoire courant
>> clear
>> whos             % les variables ne sont plus dans la mémoire
>> load ma_session  % On va chercher tous les éléments de la session
>> whos
  Name      Size                    Bytes  Class
  A         3x3                        72  double array
  b         3x1                        24  double array
Grand total is 12 elements using 96 bytes
```

Il est aussi possible de sauvegarder les variables dans un fichier ASCII et lui donner un format désiré. Elles pourront ensuite être lues par d'autres logiciels (par exemple, EXCEL) et avec MATLAB (en utilisant aussi la fonction *load*). Dans le cas suivant, on ne sauvegarde que la matrice A de l'exemple précédent.

```
>> save ma_session.dat -ASCII A
>> load ma_session.dat
```

La fonction *xlsread* permet de charger en MATLAB des fichiers EXCEL.

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

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

Exercices

15. Sauvegardez la matrice $z(x, y)$ calculez à l'exercice **14** dans un fichier ASCII (.dat). Ouvrez le fichier à l'aide d'EXCEL.

16. MATLAB permet de définir différentes formes de structures. Une structure associe un nom de variable (une chaîne de caractère) à la variable dans une cellule (type générique de variable - voir *help cell* pour plus d'information). Supposons que l'on utilise la fonction de l'exercice **14** :

$$z(x, y) = e^{-a[x^2+b(x-y)^2]}$$

avec $a = 0.5$ et $b = 0.5$ dans l'exercice précédent. On pourrait stocker cet essai dans une seule et unique variable :

```
OBJET_ESSAI = struct('matrice',{z},'a', {a}, 'b', {b})  % {}: cellules
```

Avec la sortie suivante :

```
>>
OBJET_ESSAI =

    z: [161x161 double]
    a: 0.5000
    b: 0.5000
```

On peut accéder aux différentes variables de cette structure par :

```
>> OBJET_ESSAI.a
ans =
    0.5000
>>
```

On désire organiser la sauvegarde des variables x, y, z, a et b dans un fichier .mat à l'aide d'une structure. Créez une forme plus complète de la structure présentée. Sauvegardez ensuite cet objet dans un fichier .mat.

6.3 Variables symboliques

MATLAB permet la manipulation d'expressions symboliques (c'est-à-dire de variables sans affectation numérique) de la même manière que MAPLE. Il faut cependant déclarer les variables au préalable. Par exemple, on veut calculer la matrice Jacobienne associées à un système de deux équations différentielles ordinaires.

```
%déclaration des variables symboliques
syms z1 z2 real

%déclaration de fonctions mathématiques
f1 = z1-z2^2+4; % f1:= del z1/del t
f2 =  5*(1-exp(-z1/66)); % f2:= del z2/del t

%Création de la matrice jacobienne

% A=[(del f1)/(del z1) (del f1)/(del z2)
%    (del f2)/(del z1) (del f2)/(del z2)];

A = jacobian([f1;f2],[z1 z2]);
```

MATLAB produit le résultat suivant (remarquez la classe des variables) :

```
>> A = jacobian([f1;f2],[z1 z2])
A = [                  1,              -2*z2]
    [5/66*exp(-1/66*z1),0]

>> whos
  Name      Size                  Bytes  Class
  A         2x2                     354  sym object
  f1        1x1                     142  sym object
  f2        1x1                     158  sym object
  z1        1x1                     128  sym object
  z2        1x1                     128  sym object
Grand total is 63 elements using 910 bytes
```

Il est possible ensuite d'évaluer l'expression à différents points (z1,z2) en utilisant la fonction *feval*.

```
>> z1 = 0;
>> z2 = 0;
>> A1 = eval(A)
>> A1
A1 =
    1.0000         0
    0.0758         0
>>
```

Les exercices qui suivent permettront d'illustrer d'autres fonctions qui peuvent être appliquées aux variables et objets de la classe symbolique.

Exercices

17. En utilisant la fonction *solve*, et en utilisant des variables symboliques, trouvez les solutions de l'équation $x^2 + 1 = 0$.

18. Soit l'expression suivante :

$$y = t^3 + cos(t)$$

En utilisant les variables symboliques et la fonction *diff*, déterminez l'expression de $\frac{dy}{dt}$. Évaluez la dérivée pour $t = 5$ (utilisez la fonction *eval*).

6.4 Racines d'une équation polynomiale

Ce problème est relativement courant (dans certains problèmes de thermodynamique par exemple). Supposons que l'on ait à résoudre l'équation polynomiale d'ordre 5 suivante :

$$x^5 + 3x^4 - 8x^3 + 12x^2 - x + 4 = 0$$

Il suffit de déclarer un vecteur contenant les coefficients du polynôme :

```
>> poly = [1 3 -8 12 -1 4];
```

et d'appeler la fonction *roots* de MATLAB :

```
>> roots(poly)
ans =
  -5.0623
   1.1051 + 1.1056i
   1.1051 - 1.1056i
  -0.0739 + 0.5638i
  -0.0739 - 0.5638i
```

Exercices

19. Programmez une fonction qui permet de déterminez le nombre de racines réelles d'un polynôme. Pour ce faire, utilisez la fonction *isreal(x)* de MATLAB qui retourne 1 si x est réel et 0 sinon.

6.5 Régression

Cette sous-section présente une méthode relativement simple pour effectuer des régressions dans MATLAB (qui vous permettront de **vérifier** les résultats obtenus par d'autres méthodes enseignées dans les cours). Cette méthode peut s'avérer une alternative intéressante à EXCEL dans vos cours de laboratoire...

Supposons deux vecteurs de données :

```
>> x = [0 1 2 3 4 5]; % variable indépendante
>> y = [0 2.2 4.5 5.4 5.5 5.5]; % var. dépendante
```

On veut avoir le meilleur modèle d'ordre 3 reliant ces données entre elles :

```
>> vect_modele = polyfit(x,y,3)
vect_modele =
    0.0074   -0.4091    2.9636   -0.0865
```

Le modèle est donc :

$$0.0074x^3 - 0.4091x^2 + 2.9636x - 0,0865 = 0$$

Vérifions maintenant ce modèle. Tout d'abord, on utilise le modèle pour déterminer les nouvelles valeurs de y, appelées y_{modele} et on trace un simple graphique :

```
>> y_modele=polyval(vect_modele,x);
>> plot(x,y,'+',x,y_modele,'*')
```

La figure 6 présente le résultat.

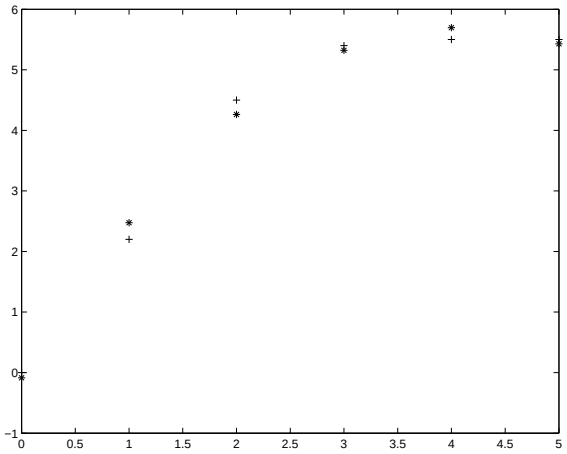

FIG. 6 – Essai d'une régression

Exercices

20. On se propose d'approximer la fonction

$$y(t) = 10 * e^{-t} + k$$

pour t entre 0 et 10 (utilisez des incréments de 0.1) par un polynôme d'ordre 5. Le paramètre k est généré par une fonction aléatoire :

```
k = rand(length(t),1)  % t est un vecteur colonne
```

6.6 Intégration numérique

L'intégration numérique d'intégrale est très importante en génie chimique (par exemple en Calculs des réacteurs chimiques, en Procédés de séparations, etc.). La fonction d'intégration la plus populaire est *quad*. Il s'agit d'une méthode adaptive de l'algorithme de Simpson. Supposons que l'on veuille évaluer l'intégrale suivante :

$$\int \frac{sin(x)dx}{x}$$

pour l'intervalle $]0, 1]$. Il suffit d'utiliser les lignes de code suivantes :

```
>> S = quad('sin(x)./x',0,1)
S =
    0.9461
```

Encore une fois, on vous rappelle que cette fonction vous permet de **vérifier** les résultats obtenus par une méthode enseignée dans un cours (et ne vous soustrait pas à l'obligation d'en programmer quelques unes) !

Exercices

21. On désire calculer la valeur de l'intégrale suivante :

$$\int_0^1 \frac{dx}{1+\sqrt{x}}$$

Dans un premier temps, on vous demande de réaliser une fonction MATLAB qui utilise la méthode de Simpson 3/8 (quatre points équidistants notés 0...3) :

$$\int f(x)dx \approx \frac{3h}{8}\left(f_0 + 3f_1 + 3f_2 + f_3\right)$$

Comparez la solution à celle obtenue à l'aide de la fonction *quad*.

Élément	Description
t	vecteur colonne du temps résultant
y	matrice des solutions
fonctionDiff	fonction qui produit les dérivées à chaque incrément de temps
vecteur temps	vecteur du temps d'appel [tinitial tfinal]
vecteur C.I.	vecteur colonne des conditions initiales

TAB. 8 – Éléments d'appel à ode45

6.7 Solution d'équations différentielles ordinaires avec valeurs initiales

Le problème de résolution d'équations différentielles ordinaires aux conditions initiales est courant en génie chimique. Dans plusieurs cours du baccalauréat, vous avez appris des méthodes pour solutionner ce type de problème (méthode d'Euler, algorithme de Runge-Kutta). Ces méthodes sont programmées dans MATLAB et elles sont relativement semblables dans leur utilisation. La plus populaire est la méthode de Runge-Kutta d'ordres 4 et 5 à pas variable implantée dans la fonction *ode45*. Les autres fonctions, par exemple *ode23* (méthode d'Euler) et *ode15s* (pour les problèmes plus ardus à résoudre) sont appelés, dans leur appel de base, de la même manière que *ode45*. Le prototype d'appel est le suivant.

```
>>  [t, y] = ode45('fonctionDiff',[vecteur temps],[vecteur cond. initiales])
```

Les éléments de cet appel sont résumés au tableau 8.

Il est donc nécessaire d'écrire un fichier fonction contenant l'expression numérique des dérivées avant d'appeler *ode45*. Par exemple, on veut résoudre le système d'équations différentielles suivant :

$$\frac{dx_1}{dt} = asin(x(1)) + cos(x(2))$$
$$\frac{dx_2}{dt} = cos(x(1)) + bsin(x(2))$$

Avec les conditions initiales $x_1(0) = 0$ et $x_2(0) = 1$ pour une durée de 10 unités de temps. a et b sont des paramètres. La fonction *fonctionDiff* serait la suivante :

```
% fonctionDiff.m

function dxdt = fonctionDiff(t,x)

% paramètres
a = 1; b = 0.5;

% équations différentielles
dxdt(1) = a*sin(x(1)) + cos(x(2));
dxdt(2) =  cos(x(1)) + b*sin(x(2));

% vecteur colonne
dxdt = [dxdt(1);dxdt(2)];
```

et l'appel à ode45 (normalement fait dans un SCRIPT) :

```
>> [t,y] = ode45('fonctionDiff',[0 10],[0;1]);
```

Regardons maintenant les sorties t (un vecteur colonne de 69 éléments) et y (une matrice de 69 lignes et de 2 colonnes, soient x_1 et x_2) :

```
>> whos
  Name      Size                    Bytes  Class
  t        69x1                      552  double array
  y        69x2                     1104  double array
Grand total is 207 elements using 1656 bytes
```

Pour tracer les graphiques, on pourrait utiliser le code suivant :

```
figure(1)
subplot(2,1,1)
plot(t,y(:,1)) % y(:,1) signifie la première colonne de y
subplot(2,1,2)
plot(t,y(:,2)) % y(:,2) signifie la deuxième colonne de y
```

Exercices

22. Programmez un fichier SCRIPT et un fichier FUNCTION qui permet de résoudre le système d'équations différentielles non-linéaire suivant :

$$\frac{dC_A(t)}{dt} = 0.5714 * (10 - C_A(t)) - (\frac{5}{6}) * C_A(t) - (\frac{1}{6}) * C_A^2(t)$$
$$\frac{dC_B(t)}{dt} = -0.5714 * C_B(t) + (\frac{5}{6}) * Ca - (\frac{5}{3}) * C_B(t)$$

Avec $C_A(0) = 5$ mol/L et $C_A(0) = 0$ mol/L. Faites tracez les courbes de $C_A(t)$ et $C_B(t)$ en fonction du temps pour une période de 10 unités de temps sur deux graphiques différents (sur une même figure).

7 Références

7.1 Références générales

Il existe beaucoup de livres sur le marché sur l'utilisation et la programmation avec MATLAB. Deux sont très intéressants et très complets :

Duane Hanselman et Bruce Littlefield, **Mastering MATLAB 6- A Comprehensive Tutorial and Reference**, Prentice-Hall, Upper Saddle (NJ), 2001.

E.B. Magrab et.al., **An Engineering Guide to MATLAB**, Prentice-Hall, Upper Saddle (NJ), 2000.

Il faut noter que la documentation complète de MATLAB (en incluant tous les *toolboxes* se trouvent sur le site de Mathworks (**www.mathworks.com**)

7.2 Références spécifiques au génie chimique

Ici encore, il existe une beaucoup de livres et de documentation sur le WEB. Quelques'uns des plus utilisés sont les suivants :

Alkis Constantinides et Navid Mostoufi, **Numerical Methods for Chemical Engineers with MATLAB Applications**, Prentice Hall International Series in the Physical and Chemical Engineering Sciences, Upper Saddle (NJ), 1999.

B. Wayne Bequette, **Process Dynamics - Modeling, Analysis, and Simulation**, Prentice Hall International Series in the Physical and Chemical Engineering Sciences, Upper Saddle (NJ), 1998.

7.3 Notes

Il existe plusieurs *toolboxes* de MATLAB aux usages très spécifiques. Beaucoup sont reliés au contrôle et seront introduits, lorsque nécessaire, dans le cours de commande. Le *toolbox optimization* est aussi très utile en génie chimique et son utilisation est traitée dans les livres de la section 7.1. Finalement, certains problèmes de transfert peuvent être solutionnés à l'aide du *PDE toolbox*. Dans les cours concernés, ces outils peuvent aider à **valider** les résultats obtenus à l'aide des techniques enseignées dans ces cours. Il est toujours possible d'obtenir la liste des fonctions disponible dans un *toolbox* avec la commande :

```
>> help nom_toolbox
```

Certains *toolbox* finalement, possèdent des interfaces usagers et des démonstrations qui facilitent grandement leur apprentissage. À condition de fouiller un peu !