Université Moulay Ismaïl
Faculté des Sciences et Techniques;
Errachidia
Département de physique

Licence en science et technique :
Energies Renouvelables, Option : Technologie Solaires et éoliennes

Elément de Module : Calcul Scientifique (P522)
Polycopié de Cours

2023-2024

Pr. Sara TEIDJ

Table des matières

Introduction

Ce cours constitue une introduction au Calcul Scientifique. Son objectif est de présenter des méthodes numériques permettant de résoudre avec un ordinateur des problèmes mathématiques qui ne peuvent pas être traités simplement avec une feuille et un stylo.

Après la présentation de quelques éléments de base de *Matlab,* on introduira les principales opérations usuelles sur les scalaires, les vecteurs et les matrices. On verra ensuite comment utiliser des fichiers script (M-file) et function avec *Matlab* avant de présenter certaines opérations graphiques offertes par ce logiciel.

Dans le chapitre 4 on apprendra la syntaxe des tests et des différentes boucles de programmation en *Matlab,* et dans le chapitre 5 on verra quelques fonctions plus avancées existant dans *Matlab*.

Dans le chapitre 5, 6 et 7 on apprendra comment résoudre les équations non-linéaires, les équations linéaires par les méthodes directes et des équations linéaires par les méthodes itératives.

Il faut rappeler ici qu'un langage scientifique comme *Matlab* exige rigueur et maîtrise parfaite. Sa simplicité apparente ne doit pas cacher l'effort qu'il faut fournir pour apprendre correctement la syntaxe qu'il utilise et savoir ensuite comment traduire dans ce langage les algorithmes mathématiques. Un langage de programmation ne peut se substituer au travail amont qui consiste à traduire sous forme d'algorithme et ensuite d'organigramme ce que l'on veut programmer.

Chapitre1: Introduction à l'environnement Matlab

1. Présentation de Matlab

Matlab est un langage de programmation, mais il est beaucoup plus que ça. Il s'agit en fait de ce qu'on appelle une console d'exécution (*shell*) qui partage certaines des caractéristiques des consoles DOS ou UNIX.

Comme toutes les consoles, *Matlab* permet d'exécuter des fonctions, d'attribuer des valeurs à des variables, d'effectuer des opérations mathématiques, de manipuler des matrices, de tracer facilement des graphiques, etc.

La figure 1 présente l'écran *Matlab* de base: la fenêtre de commandes.

Le symbole >> s'appelle prompt ou bien invite de Matlab. Il invite l'utilisateur à taper une commande.

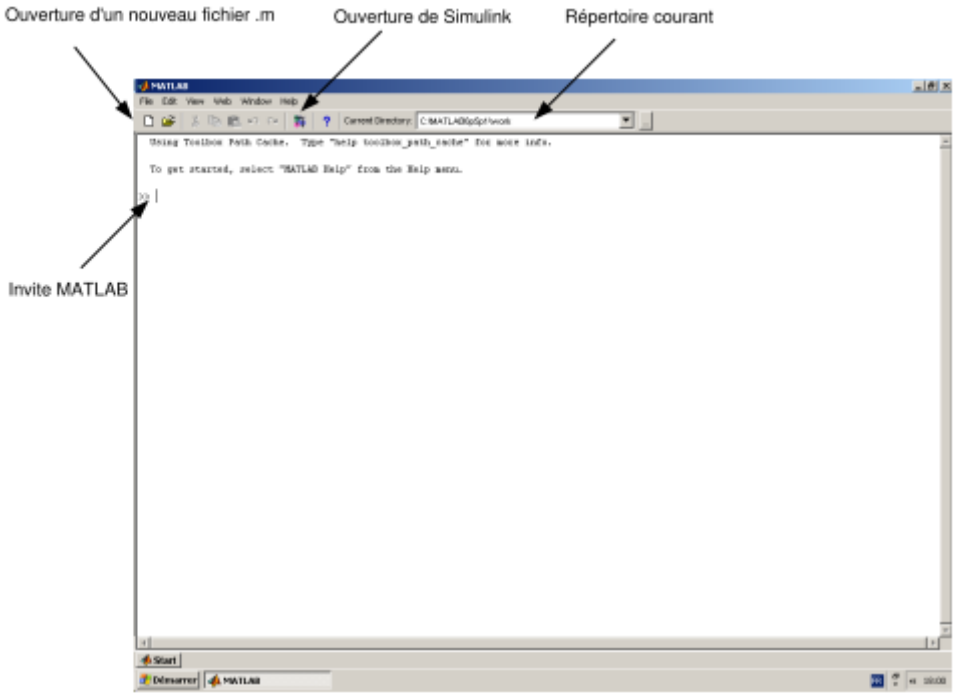


Figure 1 : Fenêtre de commandes de *Matlab*

La commande *quit* permet de quitter MATLAB :

>>quit

La commande *help* permet de donner l'aide sur un problème donné.

Il est utile de noter que le langage *Matlab* n'est pas un langage compilé (contrairement au langage C++, par exemple). Le logiciel lit et exécute les programmes instruction par instruction et ligne par ligne.

Lorsque *Matlab* détecte une erreur, le logiciel s'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" les programmes rapidement et efficacement.

2. L'environnement de Matlab

Matlab affiche au démarrage plusieurs fenêtres. Selon la version on peut trouver les fenêtres suivantes:

Current Folder : indique le répertoire courant ainsi que les fichiers existants.

Work space: indique toutes les variables existantes avec leurs types et valeurs.

Command History: utilisée pour formuler nos expressions et interagir avec Matlab.

C'est la fenêtre que nous utilisons tout au long de ce chapitre.

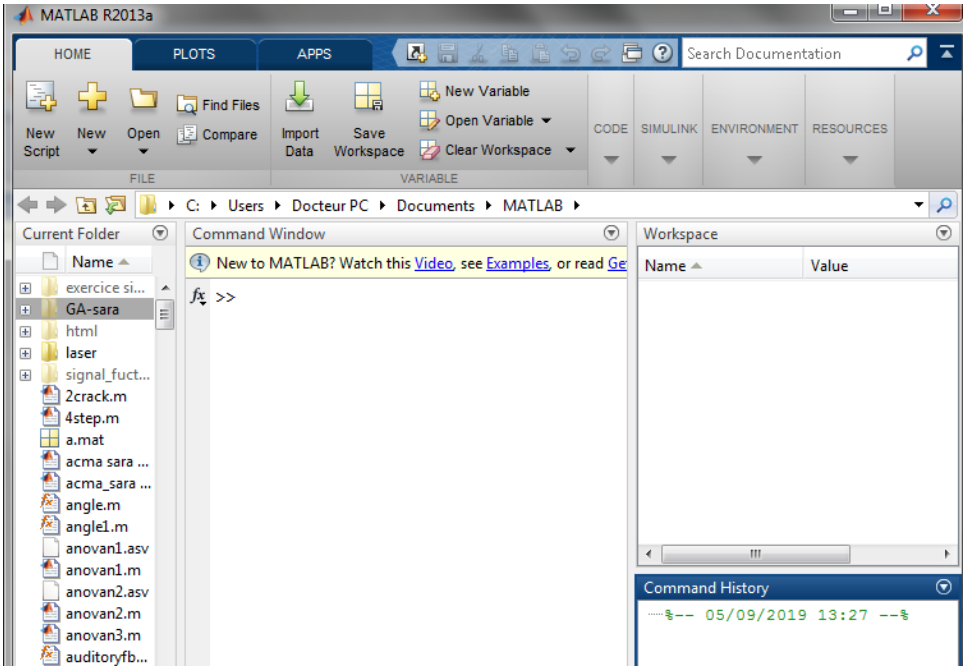


Figure 2 : L'environnement de *Matlab*

Dans la fenêtre de commande, l'utilisateur peut affecter des valeurs à des variables et effectuer des opérations sur celles-ci.

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 par défaut le résultat d'une opération dans la variable *ans*. Il est toujours possible de connaître les variables utilisées et leur type à l'aide de la commande *who* ou bien *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.

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

La 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
>>
```

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*.

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

Les classes les plus utiles pour l'utilisateur débutant sont l'entier, le réel simple, le réel double et les variables caractère 'char'.

Pour les variables *char*, la déclaration se fait entre apostrophes:

```
>> mot1 = 'bonjour'
mot1 = bonjour
```

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

```
>> mot1 = 'bonjour';
```

>> mot2 = ’tout le monde’;

>> mot1_2 = [mot1 ’ ’ mot2] % l’emploi de ’ ’ permet d’introduire un espace

mot1_2 = bonjour tout le monde.

3. Les principales constantes, fonctions et commandes

Matlab définit les constantes suivantes:

La constante	Sa valeur
pi	π=3.1415.....
exp(1)	e=2.7183
i	$=\sqrt{-1}$
j	$=\sqrt{-1}$
Inf	∞
NaN	Not a Number (Pas un nombre)
eps	$\varepsilon \approx 2\times10^{-16}$

Parmi les fonctions fréquemment utilisées, on peut noter les suivantes:

La fonction	Sa signification
sin(x)	Le sinus de x (en radian)
cos(x)	Le cosinus de x (en radian)
tan(x)	Le tangent de x (en radian)
asin(x)	L'arc sinus de x (en radian)
acos(x)	L'arc cosinus de x (en radian)
atan(x)	L'arc tangent de x (en radian)
sqrt(x)	La racine carrée de x : $\sqrt{x}$
abs(x)	La valeur absolue de x : $\lvert x \rvert$
exp(x)	$=e^{x}$
log(x)	Logarithme naturel de x : $\ln(x)=\log_e(x)$
log10(x)	Logarithme à base 10 de x : $\log_{10}(x)$
imag(x)	La partie imaginaire du nombre complexe x
real(x)	La partie réelle du nombre complexe x

round(x)	Arrondi un nombre vers l'entier le plus proche
floor(x)	Arrondi un nombre vers l'entier le plus petit: $\max\{n \mid n \le x, n \text{ entier}\}$
ceil(x)	Arrondi un nombre vers l'entier le plus grand: $\max\{n \mid n \ge x, n \text{ entier}\}$
conj(X)	conjugué du nombre complexe X
angle(X)	argument (en radians)

Matlab offre beaucoup de commandes pour l'interaction avec l'utilisateur. Nous nous contentons pour l'instant d'un petit ensemble, et nous exposons les autres au fur et à mesure de l'avancement du cours.

La commande	Sa signification
who	Affiche le nom des variables utilisées
whos	Affiche des informations sur les variables utilisées
clear x y	Supprime les variables x et y
clear, clear all	Supprime toutes les variables
clc	Efface l'écran
exit,quit	Fermer l'environnement Matlab
format: format long format short e format long e	Définit le format de sortie pour les valeurs numériques. format long à 15 chiffres. format court à 5 chiffres avec notation en virgule flottante. format long à 15 chiffres avec notation en virgule flottante.
disp	permet d'afficher un tableau de valeurs numériques ou de caractères.
Num2str	pour convertir une valeur numérique en une chaîne de caractères
Input	permet de demander à l'utilisateur d'un programme de fournir des données (La syntaxe est var = input(' une phrase ').

Chapitre 2: Opérations mathématiques de base avec *Matlab:* Scalaires, vecteurs et matrices

Plusieurs types de données sont disponibles dans *Matlab*. Les types traditionnels que l'on retrouve dans tous les langages de programmation: les types numériques (single, double, int8, etc...), caractères char, les tableaux de réels, et les tableaux creux sparse, et les types composées cell, structure ainsi que les types définis par l'utilisateur, comme les fonctions *inline*. Le type de donnée privilégiée sous *Matlab* est les tableaux à une ou deux dimensions, qui correspondent aux vecteurs et matrices utilisés en mathématiques et qui sont aussi utilisés pour la représentation graphique. Nous allons donc nous attarder sur leur définition et leur maniement dans les paragraphes qui suivent.

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;
```

1. Vecteurs

Les vecteurs lignes se déclarent de la manière suivante :

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

Ou bien

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

Pour les vecteurs colonnes, 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 le point 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:

```
>>  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
```

La création d'un vecteur dont les composants sont ordonnés par intervalles régulier et avec un nombre d'éléments bien déterminé peut se réaliser avec la fonction *linspace* (début, fin, nombre d'éléments).

Le pas d'incrémentation est calculé automatiquement par Matlab selon la formule suivante:

$$\text{le pas} = \frac{\text{fin} - \text{début}}{\text{nombre d'éléments} - 1}$$

```
>> X=linspace(1,10,4) % un vecteur de quatre élément de 1 à 10
X =
    1    4    7   10
```

La taille d'un vecteur (le nombre de ses composants)peut être obtenue avec la fonction *length* comme suit:

```
>>length(X)    % la taille du vecteur X
ans =
    4
```

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

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

>>  V=V1+V2  %  addition  de  vecteurs
V  =
4       6
>>  V=V2-V1  %  soustraction  de  vecteurs
V  =
2       2
>>  V=2*V1   %  multiplication  par  un  scalaire
V  =
2       4
```

Dans le cas de la multiplication et de la division, il faut faire attention aux dimensions des vecteurs en cause.

Pour la multiplication et la division élément par élément, on ajoute un point devant l'opérateur (.* et ./). Par exemple :

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

```
V  =
3       8
>> V=V1./V2   % division élément par élément
V  =
0.3333      0.5000
```

Cependant, *Matlab* lance une erreur lorsque les dimensions ne concordent pas. Les messages d'erreur sont utiles pour corriger les programmes (parenthèse oublié par exemple). Il faut cependant procéder à la vérification systématique de l'instruction ou du programme avant de lancer l'exécution (reflexe de base d'un programmeur):

```
>> 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
```

2. Matrices

On peut aussi créer des matrices à partir de vecteurs, 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 colonne (de haut en bas) avant d'accéder à la colonne suivante. Ainsi, dans la matrice 3x3 suivantes:

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

les valeurs des éléments ai,j sont données par leur rang affecté par *Matlab*. Le4e élément est 2 :

```
>> a4=A(4)
a4 =
2
```

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 =
8 5 6
7 8 9
```

Il est possible d'inverser *inv*(), de transposer *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 prédéfinies pour les matrices. L'addition et la soustraction sont directes (attention aux dimensions) ainsi que la multiplication par un scalaire :

```
>>  A=[1  2;3  4];
>>  B=[4  3;2  1];
>>  C=A+B  %  addition
C=
5        5
5        5
>>  D=A-B  %  soustraction
D=
-3      -1
1        3
>>  C=3*A  %  multiplication  par  un  scalaire
C=
3        6
9       12
```

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

```
>>  C=A*B    %  multiplication  de  matrices
C=
8        5
20       13
>>  D=A/B    %  division  de  matrices
D=
1.5000    -2.5000
2.5000    -3.5000
```

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
```

Dans ce cas *length* donne la plus grande dimension, ici le nombre de colonnes.

```
A=[1 2 3 ; 2 4 5 ; 6 7 8 ];
det(A) % calcule le déterminant de A
>>det(A)
ans =
  -5.0000
>> B=[-5 -2; 2  1];
>>abs(B)      % la valeur absolue
ans =
    5    2
    2    1
```

Il existe aussi des commandes qui sont propres aux vecteurs. Ces commandes s'appliquent aussi aux matrices. Dans ce cas la commande porte sur chaque vecteur colonne de la matrice.

La commande	Sa signification
sum(x)	Somme des éléments du vecteur x
prod(x)	produit des éléments du vecteur x
max(x)	plus grand élément du vecteur x
min(x)	plus petit élément du vecteur x
mean(x)	moyenne des éléments du vecteur x
sort(x)	ordonne les éléments du vecteur x par ordre croissant

3. Les polynômes dans Matlab

Dans Matlab, les polynômes sont représentés sous forme de vecteurs lignes dont les composantes sont données par ordre des puissances décroissantes. Un polynôme de degré n est représenté par un vecteur de taille (n+1).

a) Représentation d'un polynôme

Le polynôme: $p(x)=3x^2 - 5x + 2$

On commence par définir un " vecteur " qui contient les coefficients du polynôme :

```
p = [3 -5 2]
p =
    3    -5     2
```

b) Les racines d'un polynôme:

La fonction *roots* permet de trouver les racines d'un polynôme. L'exemple suivant montre l'utilisation de cette fonction.

```
roots(p)  % trouver les racines d'un polynôme
ans =
1.0000
0.6667
```

c) Détermination des coefficients d'un polynôme à partir de ses racines

La fonction *poly* permet de trouver le polynôme à partir de ses racines.

On cherche, par exemple, le polynôme qui a pour racines: 2 et 1

Celle-ci peuvent être définies comme les éléments d'un vecteur a.

```
a=[2 1]
a =
    2     1
>>poly(a)    %trouve le polynôme à partir de ses racines
ans =
    1    -3     2
```

Qui correspond à $f(x)= x^2 -3x +2$

d) Evaluer le polynôme

Pour évaluer un polynôme en un point, on utilise la fonction *polyval*

Essayons de trouver la valeur du polynôme p en 1 et celle du polynôme a en 0.

```
>> p = [3   -5   2]
```

```
p =
    3    -5     2
>>polyval(p,1)   % évalue le polynôme
ans =
    0
>> a=[2 1]
a =
  2     1
>>polyval(a,0)
ans =
    1
```

e) Les opérations de polynôme

La multiplication et la division de polynôme peuvent être réalisées facilement avec MATLAB.

Soit deux polynômes P1 et P2 définis par :

$P1(x)=x+2$

$P2(x)=x^2-2x+1$

```
>> P1=[1 2]
P1 =
    1     2
>> P2=[1 -2 1]
P2 =
    1    -2     1
```

Le résultat de la multiplication de P1 par P2 est le polynôme P3 qui s'obtient avec la fonction *conv*.

```
>> P3=conv(P1,P2)
P3 =
    1     0    -3     2
```

La division de deux polynômes se fait par la fonction *deconv*. Le quotient Q et le reste R de la division peuvent être obtenus sous forme d'élément d'un tableau.

```
>> [Q, R] = deconv (P2, P1)
Q =
    1    -4
R =
    0     0     9
```

4. Extraction d'une sous-matrice

On peut utiliser les deux points pour extraire une sous-matrice d'une matrice A.

A(:,j) :extrait la jème colonne de A. On considère successivement toutes les lignes de A et on choisit le jème élément de chaque ligne.

A(i,:) :extrait la ième ligne de A.

A(:) :reforme le matrice A en un seul vecteur colonne en concaténant toutes les colonnes de A.

A(j:k) :extrait les éléments j à k de A et les stocke dans un vecteur ligne

A(:,j:k) :extrait la sous-matrice de A formée des colonnes j à k.

A(j:k,:) : extrait la sous-matrice de A formée des lignes j à k.

A(j:k,q:r) :extrait la sous-matrice de A formée des éléments situés dans les lignes j à ket dans les colonnes q à r.

Ces définitions peuvent s'étendre à des pas d'incrémentation des lignes et des colonnes différents de 1.

Par exemple:

```
>> A=[ 1 2 3 4; 5 6 7 8;9 10 11 12]   % création de la matrice A
A =
    1     2     3     4
    5     6     7     8
    9    10    11    12
>> A(2,3)  % l'élément dur la 2éme ligne à la 3éme colonne
ans =
    7
>>A(1,:)   % tous les éléments de la 1ére ligne
ans =
    1     2     3     4
>>A(:,2)     % tous les éléments de la 2éme colonne
ans =
    2
    6
   10
>>A(2:3,:)   % tous les éléments de la 2éme et la 3éme ligne
ans =
```

```
    5     6     7     8
    9    10    11    12
>>A(1:2,3:4)  % la sous matrice supérieure droite de taille 2x2
ans =
    3     4
    7     8
>>A([1,3],[2,4])   % la sous matrice: ligne(1,3) et colonnes (2,4)
ans =
    2     4
   10    12
>>A(:,3)=[]   % supprimer la 3éme colonne
A =
    1     2     4
    5     6     8
    9    10    12
>>A(2,:)=[]    % supprimer la 2éme ligne
A =
    1     2     4
    9    10    12
>> A=[A , [0;0]]  % Ajouter une nouvelle colonne  ou A(:,4)=[0;0]
A =
    1     2     4     0
    9    10    12     0
>> A=[A ;[1, 1, 1, 1]] % Ajouter une nouvelle ligne ou A(3,:)=[1 1 1 1]
A =
    1     2     4     0
    9    10    12     0
    1     1     1     1
```

5. Génération automatique des matrices:

Dans Matlab, il existe des fonctions qui permettent de générer automatiquement des matrices particulières. Dans le tableau suivant nous présentons les plus utilisées:

La fonction	Signification
zeros(n)	Génère une matrice nxn avec tous les éléments=0
zeros(m,n)	Génère une matrice mxn avec tous les éléments=0
ones(n)	Génère une matrice nxn avec tous les éléments=1
ones(m,n)	Génère une matrice mxn avec tous les éléments=1
eye(n)	Génère une matrice identité de dimension nxn
magic(n)	Génère une matrice magique de dimension nxn
rand(m,n)	Génère une matrice dimension mxn de valeur aléatoire

Chapitre 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 des programmes pour effectuer les calculs désirés.

Il existe deux types de fichiers qui peuvent être programmé savec *Matlab*: les fichiers *script*(M-file) et *function*. Dans les deux cas, il faut lancer l'éditeur de fichier et sauvegarder le fichier avec l'extension **.m**.

1. Fichiers *script*

Comme tout langage, *Matlab* possède aussi un certain nombre d'instructions syntaxiques (boucles simples, conditionnelles, etc...) et de commandes élémentaires (lecture, écriture, etc...). Ces instructions syntaxiques seront vues dans la partie suivante du cours.

Dès que le calcul à effectuer implique un enchaînement de commandes un peu compliqué, il vaut mieux écrire ces dernières 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 les icônes **New** ou **open** dans la barre de menu.

Une fenêtre d'édition comme celle-ci va apparaître:

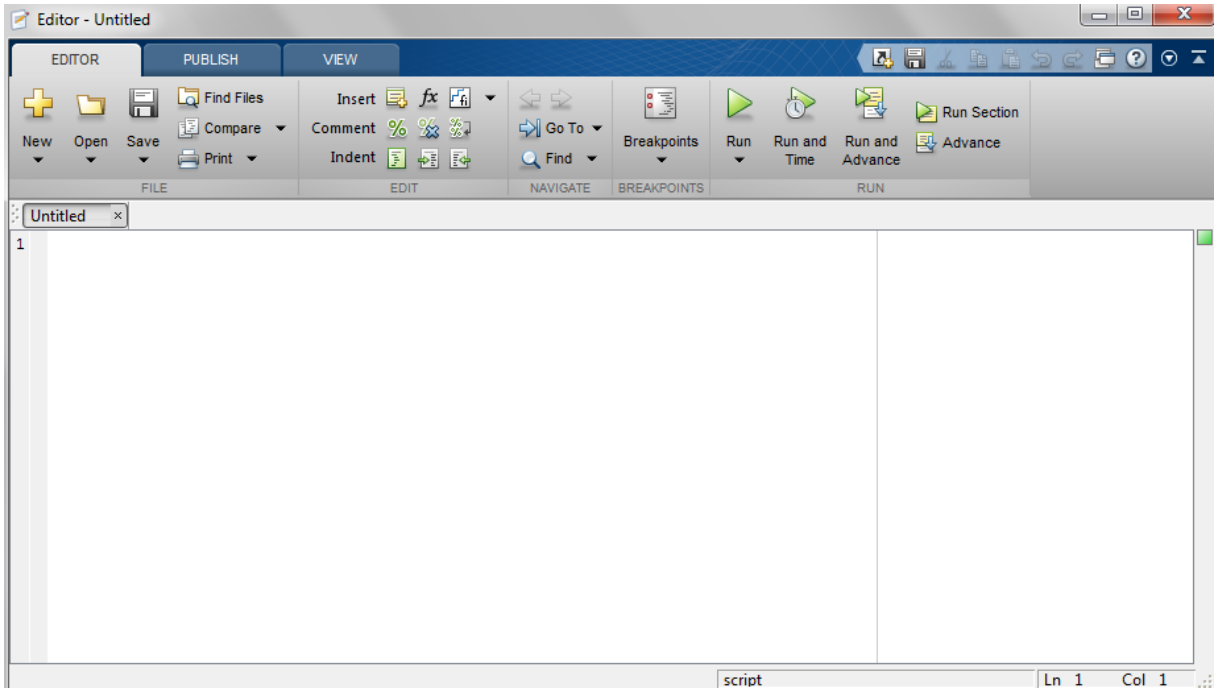


Figure 3 : La fenêtre d'édition de *Matlab*

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. Si vous avez ouvert l'éditeur comme indiqué, à partir de la fenêtre de commande, les M-file seront créés dans le répertoire courant, accessible depuis cette fenêtre, et vous n'aurez pas de problème d'accès. Si vous voulez exécuter des scripts qui se trouvent ailleurs dans l'arborescence des fichiers, vous aurez éventuellement à modifier le *Path* en cliquant sur le menu **file− >SetPath** ou bien en changeant de répertoire de travail(cliquer sur l'onglet **current directory**).

Le fichier *script* permet de lancer les mêmes opérations que celles écrites directement à la fenêtre de commandes de *Matlab* après le symbole prompt (>>). Toutes les variables utilisées dans un *script* sont disponibles à l'invite *Matlab* (fenêtres de commandes) une fois le script exécuté.

Un script *Matlab* est composé d'une suite d'instructions, toutes séparées par une virgule (ou de manière équivalente, un passage à la ligne) ou un point virgule. La différence entre ces deux types de séparation est liée à l'affichage ou non du résultat à l'écran (seulement effectué dans le premier cas).

Par exemple, créons à l'aide de l'éditeur intégré de *Matlab* dans le répertoire de travail choisi, déjà déclaré par SetPath, le fichier test. m. Supposons qu'il contient les instructions suivantes :

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

Ecrivons dans la fenêtre de commandes le nom du fichier

```
>>test
a  =
6
b  =
8
```

En tapant *whos* ensuite, on produit la sortie suivante:

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 :

– Déclarer des variables ;
– Effectuer des opérations mathématiques ;
– Appeler des fonctions ;
– Tracer des figures ;
– Programmer des algorithmes.

2. Fichiers *function* (M-file 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 estime portant de noter que l'appel de la fonction se fait en précisant ses variables entrées si ces dernières ne sont pas disponibles à l'invite *Matlab*.

Il est possible de créer nos propres fonctions en écrivant leurs codes " source" dans des fichiers M-files (portant le même nom de fonction) en respectant la syntaxe suivant:

function $[r_1, r_2, \ldots, r_n]$ = nom_fonction $(arg_1, arg_2, \ldots, arg_n)$

% le corps de la fonction

r_1=… % la valeur retournée pour r_1

r_2 =… % la valeur retournée pour r_2

r_n =… % la valeur retournée pour r_n

% le end est facultatif

end

où : $r_1, r_2, \ldots, r_n$ sont les valeurs retournées ,et $arg_1, arg_2, \ldots, arg_n$ sont les arguments.

le rôle d'une fonction est d'effectuer des opérations sur une ou plusieurs entrées pour obtenir un résultat qui sera appelé sortie.

Par exemple, la fonction suivante admet une seule sortie a qui constitue le résultat de l'addition des deux arguments d'entrée x et y:

function a = ma_fonction(x,y)

```
a=x+y;
end
```

Lorsqu'on tape dans la fenêtre de commandes:

```
>> a = ma_fonction(4,2)
```

on obtient

```
a = 6
```

Ensuite, on vérifieque

```
>>whos
Name        Size        Bytes       Class
a           1x1 8       double      array
Grand total is 1 element using 8 bytes
>>
```

Modifions la fonction pour lui demander de calculer aussi le produit de x et y de la façon suivate :

```
function [a,b] = ma_fonction2(x,y)
a=x+y;
b=x*y;
end
```

Dans ce cas, on vérifie que:

```
>> [a,b] = ma_fonction2(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_fonction2(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

Remarquons que nous pouvons appeler la fonction ma_fonction2 pour calculer uniquement la somme de x et y. On obtient alors

```
>> a = ma_fonction2(4,2)
a =
6
>>whos
Name        Size        Bytes       Class
a           1x1 8       double      array
Grand total is 1 element using 8 bytes
```

Remarques importantes :

* Le passage des arguments d'entrée dans les fonctions se fait par valeur. Aussi, même si elles sont modifiées dans la fonction les valeurs des paramètres ne sont pas modifiées dans le programme appelant.

* Si une des variables de la procédure n'est pas définie à l'intérieur de celle-ci elle doit obligatoirement être fournie en argument d'entrée.

* La récupération des valeurs calculées par la fonction se fait par les paramètres de sortie.

* Le nom du fichier contenant la fonction porte obligatoirement le nom de cette dernière. On peut mettre plusieurs fonctions dans le même M-file mais seule la fonction du même nom que le fichier peut être utilisé, appelée, à partir de la fenêtre de commandes ou d'une autre fonction ou d'un script. Les autres fonctions éventuellement stockées dans le fichier peuvent s'appeler entre elles mais ne sont pas visibles de l'extérieur.

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.

3. Définition d'une fonction par la commande « *inline* »

Une fonction ne comportant qu'un petit nombre d'instructions peut être définie directement dans la fenêtre de commandes de la manière suivante :

```
>>angle=inline('atan(y/x)')
angle  =
Inlinefunction:
angle(x,y) =  atan(y/x)
>>angle(5,4)
ans   =
0.6747
```

Les arguments de la fonction angle sont normalement fournis à l'appel dans l'ordre d'apparition dans la définition de la fonction. On peut aussi spécifier les arguments d'appel explicitement

```
>>f =inline('sin(alpha*(x+y))','x','y','alpha')
f  =
Inlinefunction:
f(x,y,alpha) =sin(alpha*(x+y))
>>f(0.2,0.3,pi)
ans  =
1
```

4. Fonctions outils

Enfin, notez que certaines commandes spéciales ne peuvent s'utiliser qu'en relation à une fonction: *nargin*, donne le nombre d'arguments d'entrée passés à l'appel de la fonction.

```
function c=testarg1(a,b)
if  (nargin ==  1)
c=2*a;
elseif  (nargin  ==  2)
c=a+b;
end
```

nargin peut aussi être utilisée pour connaître le nombre prévu d'arguments d'entrée

```
>>nargin('testarg1')
ans  =
2
```

La commande *nargout* fonctionne de manière analogue pour les arguments de sortie.

Chapitre 4: Fonctions et représentation graphique sous *Matlab*

1. Graphiques simples

Cette section vise une initiation aux nombreuses facultés graphiques offertes par *Matlab*.
Dans toutes les représentations graphiques, le logiciel se base sur des données discrètes rangées dans des matrices ou des vecteurs colonnes. Par exemple, pour représenter des courbes du type y = f(x) ou des surfaces z = f(x, y), les données x, y, z doivent être des vecteurs colonnes (x et y) ou des matrices (z) aux dimensions compatibles. L'instruction de dessin correspondante (par exemple plot(x,y) pour tracer des courbes planes) est alors utilisée et éventuellement complétée par des arguments optionnels (couleur, type de trait, échelle sur les axes, etc...). La visualisation du résultat s'effectue dans une fenêtre graphique (avec possibilité de zoom, de rotation, d'impression).

a) La fonction plot:

La fonction **plot** est utilisable avec des vecteurs ou des matrices. Elle trace des lignes en reliant des points de coordonnées définis dans ses arguments, et elle a plusieurs formes:
Si elle contient deux vecteurs de la même taille comme arguments: elle considère les valeurs du premier vecteur comme les éléments de l'axe X (les abscisses), et les valeurs du deuxième vecteur comme les éléments de l'axe Y (les ordonnées), comme dans l'exemple qui suit :

```
>> A=[2 5 3 -2 0]
A =
   2    5    3   -2    0
>> B=[-4 0 3 1 4]
B =
  -4    0    3    1    4
       >>plot(A,B)
```

Si elle contient un seul vecteur comme argument: elle considère les valeurs du vecteur comme les éléments de l'axe Y (les ordonnées), et leurs positions relatives définiront l'axe X (les abscisses),
Exemple:1

```
>> V=[2 1 6 8 -3 0 5]
```

```
V =
  2     1     6     8    -3     0     5
>>plot(V)
>>
```

Exemple:2

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

b) Modification de l'apparence d'une courbe

Ces graphiques manquent cependant de clarté. Il est possible de manipuler l'apparence d'une courbe en modifiant la couleur de la courbe, la forme des points de coordonnées et le type de ligne reliant les points.

Pour cela, on ajoute un nouvel argument (qu'on peut appeler un marqueur) de type chaîne de caractères à la fonction *plot* comme ceci:

```
plot(x,y,' marqueur')
```

Le contenu du marqueur est une combinaison d'un ensemble de caractères spéciaux rassemblés dans le tableur suivant:

Couleur de la courbe		Représentation des points	
Le caractère	Son effet	-	En ligne pleine
b	Courbe en bleu	:	En pointillé
g	Courbe en vert	--	En tiret
r	Courbe en rouge	.	Un point
y	Courbe en jaune	o	Un cercle
k	Courbe en noir	x	Le symbole x

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

c) Annotation d'une figure:

Dans une figure, il est préférable de mettre une description textuelle aidant l'utilisateur à comprendre la signification des axes et de connaître le but ou l'intérêt de la visualisation concernée.

Il est très intéressant également de pouvoir signaler des emplacements ou des points significatifs dans une figure par un commentaire signalant leurs importances.

- ✓ Pour donner un titre à une figure contenant une courbe nous utilisons la fonction *title* comme ceci:

  >>title(' titre de la figure')
- ✓ Pour donner un titre pour l'axe vertical des ordonnées y, nous utilisons la fonction *ylabel* comme ceci:

  >>ylabel(' ceci est l''axe des ordonnées Y ')
- ✓ Pour donner un titre pour l'axe horizontal des abscisses x , nous utilisons la fonction *xlabel* comme ceci:

  >>xlabel(' ceci est l''axe des abscisses X ')
- ✓ Pour écrire un texte (un message) sur la fenêtre graphique à une position indiquée par les coordonnées x et y, nous utilisons la fonction *text* comme ceci:

  >>text(x,y,'ce point est important')
- ✓ Pour mettre un texte sur une position choisie manuellement par la souris, nous utilisons la fonction *gtext*, qui a la syntaxe suivant:

  >>gtext('ce point est choisi manuellement')
- ✓ Pour mettre un quadrillage (une grille), nous utilisons la commande *grid* (ou *grid on*) pour l'enlever nous réutilisons la même commande *grid* (ou *grid off*).
- ✓ Pour fixer les limites sur les axes des abscisses et des ordonnées

  >>axis([xmin xmax ymin ymax]

Par exemple:

Dessinons la fonction: $y=-2x^3+x^2-2x+4$ pour x variant de -4 jusqu'à 4, avec un pas 0.5

```
clear all
close all
x=-4:0.5:4;
y=-2.*x.^3+x.^2-2.*x+4;
plot(x,y)
grid on
title(' Dessiner une courbe')
xlabel('L''axe des abscisses')
ylabel('L''axe des ordonnées')
```

d) Afficher plusieurs courbes dans une même fenêtre (hold on)

Il est possible d'afficher plusieurs courbes dans une même fenêtre graphique grâce à la commande *hold on*. Les résultats de toutes les instructions graphiques exécutées après appel à la commande *hold on* sera superposés sur la fenêtre graphique active. Pour rétablir la situation antérieure (le résultat d'une nouvelle instruction graphique remplace dans la fenêtre graphique le dessin précédent) on tapera *hold off*.

Voici un exemple d'utilisation de la commande *hold on*

```
clear all
close all
x=linspace(0,pi,30);
y1=cos(x);
plot(x,y1,'o-r')
y2=sin(x);
hold on
plot(x,y2,'x-b')
y3=exp(-x);
hold on
plot(x,y3,'*-g')
```

e) Utiliser plot avec plusieurs arguments.

On peut utiliser plot avec plusieurs couple (x, y) ou triplets (x,y,' marqueur') comme arguments. Un script est tout indiqué :

```
%  graphique.m
clear  all
close  all
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
```

Remarque

On dispose donc de deux façons de superposer plusieurs courbes sur une même figure.

On peut soit donner plusieurs couples de vecteurs abscisses/ordonnées comme argument de la commande plot, soit avoir recours à la commande *hold on*. Suivant le contexte on privilégiera l'une de ces solutions plutôt que l'autre.

2. Afficher plusieurs graphiques (subplot)

Voilà une fonctionnalité très utile pour présenter sur une même page graphique un grand nombre de résultats.

L'idée générale est de découper la fenêtre graphique en pavées de même taille, et d'afficher un graphe dans chaque pavé. On utilise l'instruction *subplot* en lui spécifiant le nombre de pavés sur la hauteur, le nombre de pavés sur la largeur, et le numéro du pavé dans lequel on va tracer:

subplot (Nbre pavés sur hauteur, Nbre pavés sur largeur, Numéro pavé)

La virgule peut être omise. Les pavés sont numérotés dans le sens de la lecture d'un texte : de gauche à droite et de haut en bas.

Une fois que l'on a tapé une commande *subplot*, toutes les commandes graphiques suivantes seront exécutées dans le pavé spécifié.

Comme exemple taper la suite d'instructions suivantes :

```
clear all
close all
x=[0:0.01:2*pi];
subplot(221)
plot(x,sin(x),'b')
subplot(222)
plot(x,cos(x),'r')
subplot(223)
plot(cos(2*x),'g')
subplot(224)
plot(sin(2*x),'k')
```

3. Echelles logarithmiques

On peut tracer des échelles log en abscisse, en ordonnée ou bien les deux. Les fonctions correspondantes s'appellent respectivement *semilogx*, *semilogy* et *loglog*. Elles s'utilisent exactement de la même manière que *plot*.

Par exemple :

>> x=1:100;

>>semilogx(x,log(x))

4. Autres types de représentation

Outre la représentation cartésienne de courbes ou de surfaces, il existe d'autres possibilités pour illustrer graphiquement un résultat. On peut citer parmi les plus utiles, les instructions *contour*,*ezmesh* (pour tracer les courbes de niveau d'une surface paramétrique), *mesh*,*ezplot3* (courbes paramétriques dans l'espace),*hist*, *rose* (histogramme d'un échantillon de données statistiques), etc...

Type	Description	commande
semilogy	axe des y en log de base 10 et axe des x linéaire	semilogy(x,y)
semilogx	axe des x en log et axes des y linéaire	semilogx(x,f(x))
loglog	les deux axes sont en log de base 10	loglog(x,y)

errorbar	graphique avec bar d'erreur en y sur chaque valeur	errorbar(x,y,e);
		e : vecteur erreur en chaque point de x.
		errorbar(x,y,e^{up},e^{down});
		e^{up} : étant la limite supérieure de l'erreur et e^{down} : la limite inférieure.

bar barh	graphique à bars verticales ou horizontales	bar(x,y) barh(x,y)
hist	histogramme	hist(y,nbins); nbins = nbre de barreaux hist(y,x) ; x = location du centre du barreau

plot3	Tracé d'une ligne paramétrique en 3D	plot3(x,y,z)
mesh	Tracé d'une surface en 3D, à partir de matrices de maillage	mesh(x,y,z)
surf	Tracé d'une surface en 3D avec dégradé de couleur, à partir de matrices de maillage	surf(x,y,z)

5. Fonctions mathématiques simples

Les opérateurs algébriques (+, -, *, /, .*,./) ont été définis précédemment 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 function cosinus:

```
>>  help cos
COS    Cosine of argument in radians.
   COS(X) is the cosine of the elements of X.
```

Dans la suite, on présente les fonctions mathématiques usuelles et leur appel dans *Matlab*. Ensuite, on présente les principales fonctions spécifiques aux matrices.

6. Fonctions mathématiques usuelles

Toutes les fonctions mathématiques de base sont déjà programmées dans *Matlab*.

Toutes les fonctions courantes et moins courantes existent. La plupart d'entre elles fonctionnent **en complexe**. On retiendra que pour appliquer une fonction à une valeur, il faut mettre cette dernière entre parenthèses. Exemple :

```
>>sin(pi/12)
ans  =
0.16589613269342
```

Voici une liste non exhaustive :

– fonctions trigonométriques et inverses : sin, cos, tan, asin, acos, atan
– fonctions hyperboliques (on rajoute «h») : sinh, cosh, tanh, asinh, acosh, atanh
– racine, logarithmes et exponentielles : sqrt, log, log10, exp
– fonctions erreur : erf, erfc

– fonctions de Bessel et Hankel: besselj, bessely, besseli, besselk, besselh et hankel. Il faut deux paramètres : l'ordre de la fonction et l'argument lui-même. Ainsi J1(3) s'écrirabesselj(1,3)

La notion de fonction est plus générale dans *Matlab*, et certaines fonctions peuvent avoir plusieurs entrées (comme besselj par exemple) mais aussi plusieurs sorties.

a) Fonctions matricielles

Toutes les fonctions matricielles de base sont déjà programmées dans *Matlab*.

Voici quelques exemples :

Size, length, diag, det, norm, rank, trace, sum, prod, mean, std, var, max, min, rand, null, inv, pinv, sort, reshape, fliplr, flipud, tril, triu,…

b) Fonctions avancées

Ce sont des fonctions qui interviennent en analyse numérique telles que: lu, chol, qr, cond, eig, fzero,…

Chapitre 5: Programmation avec *Matlab* et structures de contrôle

Nous avons vu jusqu'à présent comment utiliser *Matlab* pour effectuer des commandes ou pour évaluer des expressions en les écrivant dans la ligne de commande, par conséquent les commandes utilisées s'écrivent généralement sous forme d'une seule instruction (éventuellement sur une seule ligne).

Cependant, il existe des problèmes dont la description de leurs solutions nécessite plusieurs instructions, ce qui réclame l'utilisation de plusieurs lignes. Comme par exemple la recherche des racines d'une équation de second degré (avec prise en compte de tous les cas possibles).Une collection d'instructions bien structurées visant à résoudre un problème donné s'appelle un programme. Dans cette partie, on va présenter les mécanismes d'écriture et d'exécution des programmes en *Matlab.* On va parler ensuite des tests et des boucles en commençant par introduire les opérateurs de comparaison et les opérateurs logiques.

1. Principe général

Le principe est simple: regrouper dans un fichier une série de commandes *Matlab* et les exécuter en bloc. Tout se passera comme si vous les tapiez au fur et à mesure dans une session *Matlab*. Il est fortement conseillé de procéder de cette façon en créant un fichier programme (M-file) car cela permet notamment de récupérer facilement le travail fait la veille.

Les fichiers de commandes peuvent porter un nom quelconque mais doivent finir par l'extension .m (attention toutefois à certains caractères qui sont interdits : le blanc, le symbole +,.., *Matlab* ne le dira pas tout de suite mais enverra à la première tentative d'exécution un message d'erreur qui dit que le fichier est introuvable).

2. Où doit se trouver le fichier de commande?

Le plus simple, c'est qu'il se trouve dans le répertoire courant (c'est-à-dire celui où on a lancé *Matlab*). Il peut se trouver aussi dans un répertoire quelconque mais référencé dans la variable path *Matlab*. Tapez cette commande pour en voir le contenu, *Matlab* vous affichera tous les répertoires accessibles. Il est en général conseillé de se créer un répertoire propre ou d'utiliser le répertoire par défaut de *Matlab*. Pour connaître le répertoire actuel il suffit de taper la commande *pwd* dans l'invite de *Matlab*.

Le path peut être modifié avec la commande *addpath*. Cette commande permet de placer le chemin d'accès au fichier dans le fichier qui contient tous les chemins d'accès par défaut ou

déclarés, c'est-à-dire path, et qui est exécuté automatiquement au démarrage de *Matlab*. Voici un exemple:

addpath (genpath('C:\Documents and Settings\admin\Mes documents\MATLAB\Dossier'))

Cette commande permet de rajouter le nouveau répertoire « Dossier », crée dans le répertoire par défaut de *Matlab* qui est nommé MATLAB, au path d'accès.

Ainsi tous les fichiers de commandes présents dans le nouveau répertoire « Dossier » seront accessibles de n'importe où.

3. Commentaires et auto-documentation

Tout ce qui se trouve après le symbole % sera considéré comme un commentaire. Il est également possible d'auto-documenter ses fichiers de commande.

4. Suppression de l'affichage

L'affichage des résultats de toutes les commandes n'est pas nécessaire. Pour certaines commandes (création de gros tableaux), cela peut s'avérer fastidieux.

On peut donc placer le caractère ; à la fin d'une ligne de commande pour indiquer à *Matlab* qu'il ne doit pas afficher le résultat.

5. Pause dans l'exécution

Si l'on entre la commande pause dans un fichier de commandes, le programme s'arrêtera à cette ligne tant qu'on n'a pas tapé «Entrée» ou «Enter» en cas d'un clavier QWERTY.

6. Mode verbeux

Si l'on souhaite qu'au fur et à mesure de son exécution, *Matlab* affiche la séquence de commandes qu'il est en train d'exécuter, il suffit de taper :

>>echo on

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

```
echo  on
pause  %  Appuyez sur une touche pour continuer !
echo  off
```

7. Opérateurs de comparaison et logiques

Matlab utilise le langage C. Notons tout d'abord le point important suivant, justement inspiré du langage C:

Matlab représente la constante logique «FAUX» par 0 et la constante «VRAIE» par 1.

Ceci est particulièrement utile par exemple pour définir des fonctions par morceaux.

Il est important de se familiariser avec les opérateurs logiques. Le premier type de ces opérateurs permet de comparer des valeurs entre elles.

Opérateur	Syntaxe Matlab
Egal à	= =
Différent de	~ =
Supérieur à	>
Supérieur ou égal à	>=
Inférieur à	<
Inférieur ou égal à	<=
Négation	~
Ou	|
Et	&

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
```

Les opérateurs logiques sont intéressants pour construire des fonctions par morceaux. Imaginons que l'on veuille définir la fonction suivante: $f(x)=\begin{cases}\sin(x) & si \quad x>0\\ \sin(2x) & sinon\end{cases}$.

Voilà comment écrire la fonction:

```
>> f = inline('sin(x).*(x>0) + sin(2*x).*not(x>0)')
f =
    Inline function:
f(x) = sin(x).*(x>0) + sin(2*x).*not(x>0)
```

On ajoute les deux expressions sin x et sin 2x en les pondérant par la condition logiquedéfinissant leurs domaines de validité. On peut tester que ça marche en représentant la courbe:

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

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
```

Il est aussi possible de lier entre elles des conditions par l'opérateur 'et' (&) et 'ou' (|).

Ces notions seront utiles pour la construction des conditions qui seront présentées dans les prochaines sections.

Critère sur les valeurs : Fonction *find*

Nous avons vu qu'il était aisé d'appliquer un opérateur logique sur un tableau. Cela nous renvoie un tableau contenant des 1 ou des 0 (valeurs logiques true ou false) selon que le critère logique est vérifié ou non. Ce principe peut être exploité pour écrire facilement une fonction définie par morceaux, mais cela ne permet pas d'extraire ou de modifier des valeurs selon un test logique. Pour cela, on peut utiliser la fonction *find*.

La fonction *find* est utile pour identifier simplement les éléments non nuls d'un tableau, et par extension, d'identifier les valeurs vérifiant un critère logique donné.

```
indices = find(M)       %renvoie dans la variable indices la liste des indices du tableau M dont les éléments sont non nuls.
```

indices = find(opération logique sur M) %renvoie dans la variable indices la liste des indices du tableau M vérifiant l'opération logique.

Par exemple:

```
>>  x = [ -1.2 0 3.1 6.2 -3.3 -2.1]
x =
  -1.2000        0    3.1000    6.2000   -3.3000   -2.1000
```

>>find(x) %La fonction *find* permet d'identifier les éléments comportant des valeurs non nulles

```
ans =
    1     3     4     5     6
```

>>inds = find(x < 0) % permet de trouver tous les éléments correspondant à un critère logique :

```
inds =
    1     5     6
```

8. Les entrées/sorties

a) Entrée au clavier :

L'utilisateur peut saisir des informations au clavier grâce à la commande de x=input(…).

```
>> X=input('saisir une valeur de x :')
saisir une valeur de x :5
X =
    5
```

b) Sortie à l'écran

Pour afficher quelque chose à l'écran, l'utilisateur peut utiliser la commande disp, qui affiche le contenu d'une variable (chaîne de caractères, vecteur, matrice…).

```
>> A=[1 2 3] ;
disp(A);
    1     2     3
```

9. Instructions de contrôle

Les instructions de contrôle sous Matlab sont très proches de celles existant dans d'autres langages de programmation.

a) Boucles if-elseif-else

Dans un programme interviennent souvent des conditions. Les boucles *if-elseif-else* sont une structure de programmation qui est très utile pour rendre compte de cette situation.

En pseudo-code, on peut résumer la chose de la façon suivante:

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;
end
```

Si la condition est évaluée à vrai (true), les instructions entre le *if* et le *end* seront exécutées), sinon elles ne seront pas (ou si un *else* existe les instructions entre le *else* et le *end* seront exécutées). S'il est nécessaire de vérifier plusieurs conditions au lieu d'une seule, on peut utiliser des clauses *elseif* pour chaque nouvelle condition, et à la fin on peut mettre un *else* dans le cas ou aucune condition n'a été évaluée à vrai.

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

Ainsi, le programme va être exécuté en suivant les instructions écrites dans son M-File. Si une instruction est terminée par un point virgule, alors la valeur de la variable concernée ne sera pas affichée, par contre si elle se termine par une virgule ou un saut à la ligne, alors les résultats seront affichés.

Remarque

Il existe la fonction *solve* prédéfinie en *Matlab* pour trouver les racines d'une équation (et beaucoup plus). Si nous voulons l'appliquer sur notre exemple, il suffit d'écrire :

```
>>solve('-2*x^2+x+3=0','x')
ans =
 -1
 3/2
```

b) 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 estime possible de se soustraire à l'utilisation de ces boucles, voici le prototype en pseudo-code qui les traduit.

```
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 carrées des nombres pairs entre 0 et 10 :

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

c) Boucles while

Une boucle *while* permet de répéter une opération tant qu'une condition (critère) 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 faire converger une itération vers une valeur désirée dont la précision est fixée par un test de convergence.

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
```

d) Boucles switch

Les boucles *switch* permettent parfois de remplacer les boucles *if-elseif-else*, particulièrement dans le cas de menus. La boucle *switch* exécuté des groupes d'instructions selon la valeur d'une variable ou d'une expression. Chaque groupe est associé à une clause *case* qui définit si ce groupe doit être exécuté ou pas selon l'égalité de la valeur de ce *case* avec les résultats d'évaluation de l'expression de *switch.* Si toutes les *cases* n'ont pas été acceptées, il est possible d'ajouter clause *otherwise* qui sera exécutée seulement si aucun *case* n'est exécuté.

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 =
1.6
```

Chapitre 5: Résolution des équations non-linéaires $f(x)=0$

Dans ce chapitre, nous présentons plusieurs techniques de résolution des équations non linéaires. Les méthodes proposées sont :

- La méthode de la bissection
- La méthode de Newton-raphson
- La méthode de point fixe

1. Définition:

L'objet essentiel de ce chapitre est l'approximation des racines d'une fonction réelle d'une variable réelle, c'est-à-dire la résolution approchée du problème suivant :
Étant donné une équation non linéaire, à une seule variable, est définie par : $f(x)=0$
La valeur de la variable x qui vérifie cette égalité est appelée solution (ou racine) de l'équation, elle est notée c.

2. Méthode de la bissection

Cette méthode est appelée aussi « Dichotomie », elle repose sur un théorème important c'est le théorème des valeurs intermédiaires qui est à la base de l'étude de celle-ci ainsi que des autres méthodes.

a) Théorème :

Si f est une fonction continue sur l'intervalles $[a, b]$ et si on a $f(a).f(b)<0$ alors l'équation $f(x)=0$ possède au moins une racines dans l'intervalle $[a,b]$.

b) Développement de la méthode

On s'assure que si $f(a).f(b)$ sur l'intervalles $[a,b]$, la racine est unique (c.à.d. que f est monotone dans l'intervalle $[a,b]$).

On partage $[a,b]$, en 2 intervalles $[a,c]$, et $[c,b]$, tel que $c = \frac{a+b}{2}$

Si $f(a).f(c) < 0$ cela implique que la racine $\in [a,c]$

Si $f(c).f(b) < 0$ cela implique que la racines $\in [c,b]$

Si $f(c) = 0$, c serait la racine exacte

Le domaine gardé sera à son tour partage jusqu'à arriver à un petit domaine selon une précision donnée dont sa moitié sera considérée comme racine approchée de l'équation $f(x)$=0.

La figure ci-dessous illustré le principe de la méthode de Dichotomie :

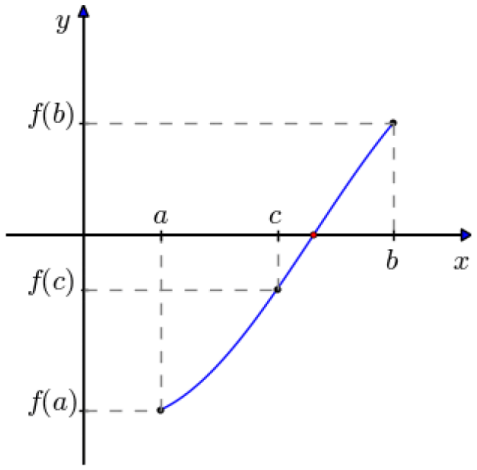


Figure 1: Principe de la méthode de Dichotomie

Exemple 1

Calculer la racines de l'équation $f(x) = x^6 - x - 1 = 0$ dans l'intervalle [1,2] avec la précision ε=0,001.

Solution 1 :

La fonction $f(x)$=0 est un polynôme donc continue sur [1,2]

$f(1) = -1$, $f(2) = 61$ $\Rightarrow$ $f(1).f(2)<0$,

Donc d'après le théorème des valeurs intermédiaires, il existe au moins une racine $c \in$[1,2] tel que $f(c)$=0.

De plus $f'(x)= 6x^5-1$ $\forall\ x\in$[1,2] $f'(x)>0$ $\Rightarrow$ $f \nearrow$ (f est monotone)

On déduit alors que la racines de f dans [1,2] est unique.

On cherche maintenant à calculer une approximation de cette racine.

Nous allons construire les suites $(a_n)_{n\in\mathbb{N}}$,$(b_n)_{n\in\mathbb{N}}$ et $(c_n)_{n\in\mathbb{N}}$

Etape 0 : a_0=1 , b_0=2 , $c_0 = (a_0 + b_0)/2$=1.5 , $f(1.5)$=8.8609

Test. $f(a_0).f(c_0) = f(1).f(1.5) < 0$ $\Longrightarrow$ on resserre l'intervalle du côté droit (côté de b)

n	a	b	c	$b-c$	$f(c)$	test

1	1.0000	2.0000	1.5000	0.5000	8.8609	$f(a_0).f(c_0) = f(1).f(1.5) < 0$ (côté de b)
2	1.0000	1.5000	1.2500	0.2500	1.5647	$f(a_0 = a_1).f(c_1) = f(1).f(1.25) < 0$ (côté de b)
3	1.0000	1.2500	1.1250	0.1250	-0.0977	$f(a_1 = a_2).f(c_2) = f(1).f(1.125) > 0$ (côté de a)
4	1.1250	1.2500	1.1875	0.0625	0.6167	$f(a_3 = c_2).f(c_3) = f(1.125).f(1.1875) < 0$ (côté de b)
…	…	…	…	…	…	…
10	1.1300	1.1348	1.1338	0.001	0.0096	…

c) Algorithme de la méthode

Pas1 : $c \leftarrow (a+b)/2$

Pas 2: Si $b - c \leq \varepsilon$, c est considérée racines approchée (stop)

Pas 3: Si $f(c) = 0$, c est considérée racine exacte (stop)

Pas 4: Si $f(a).f(c) < 0$ alors $c \leftarrow b$

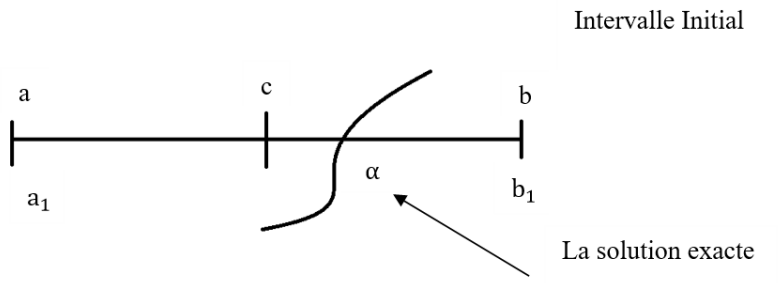
Si non $c \leftarrow a$

Pas 5: Retour au pas 1.

d) Convergence et estimation de l'erreur

On démontre que la méthode de la bissection est convergente (vers la solution unique de l'équation $f(x)$=0 dans l'intervalle [a,b]).

On cherche à déterminer l'erreur maximale commise en utilisant la méthode de la bissection dans l'intervalle [a,b].



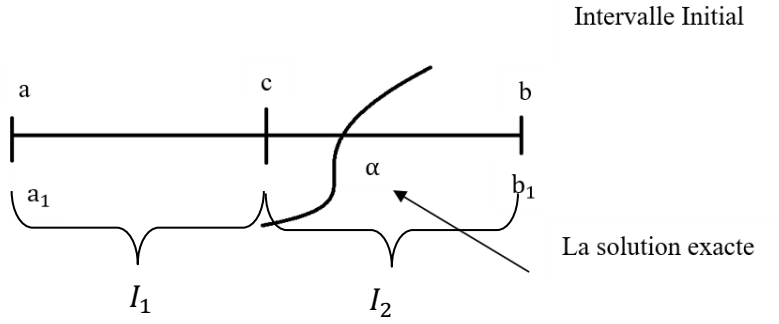
Pour i=1: La longueur du premier intervalle $[a_1, b_1][a, b]$ est $b_1 - a_1 = b - a$

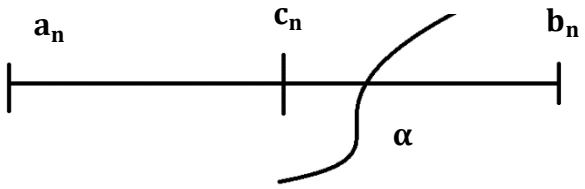
Pour i=2 : La longueur du deuxième intervalle $[a_2,b_2]$ est $b_2 - a_2 = \frac{b-a}{2}$

Pour i=3 : La longueur du troisième intervalle $[a_3,b_3]$ est $b_3 - a_3 = \frac{b-a}{2^2}$

Pour i=4 : La longueur du quatrième intervalle $[a_4,b_4]$ est $b_4 - a_4 = \frac{b-a}{2^3}$

Pour i=n : La longueur de n^{ième} (dernier) intervalle $[a_n,b_n]$ est $b_n - a_n = \frac{b-a}{2^{n-1}}$





La racine approchée est $c_n = \frac{a_n+b_n}{2}$, Si on désigne par α la solution exacte, alors on a:

$$|c_n - \alpha| \leq b_n - c_n = \frac{b_n - a_n}{2} = \frac{b-a}{2^n} \leq \varepsilon$$

Donc, l'erreur $|c_n - \alpha| \leq \frac{b-a}{2^n}$ (1)

La relation (1) permet aussi de calculer à l'avance le nombre max d'itérations nécessaires,

$n \geq \frac{\ln\left(\frac{b-a}{\varepsilon}\right)}{\ln(2)}$ (2) (n ne dépend pas de f)

Comme exemple : Si l'intervalle est [1,2] et $\varepsilon = 10^{-3}$ alors $n \geq 9{,}93$

On prend n=10. Il est important de remarquer que le nombre d'itérations nécessaire donné par la formule ci-dessus est, dans plusieurs cas, une surestimation du nombre réel d'itérations nécessaire (pour le calcul on peut se contenter de n=9 tout en ayant atteint la précision voulue).

3. Méthode de Newton-Raphson

Soit α la racine exacte de l'équation $f(x) = 0$. Si f est continue et continument dérivable au voisinage de α, alors le développement en série de Taylor autour de x_0 (x_0 étant la valeur Initiale) s'écrit:

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \underbrace{\frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \cdots}_{\mathbf{R}}.$$

Donc on peut écrire :

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + R$$

Posons $x = \alpha$ on trouve:

$$f(\alpha) = f(x_0) + (\alpha - x_0)f'(x_0) + R = 0$$
$$\Rightarrow \alpha = x_0 - \frac{f(x_0)}{f'(x_0)} + R_1$$

En ignorant R_1 on obtient une nouvelle approximation x_1(meilleure que x_0)

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

On considère maintenant le développement en série de Taylor autour de x_1

De la même manière que précédemment on trouve une nouvelle valeur x_2 tel que:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

Ainsi on obtient la relation récursive suivante :

$$\begin{cases} x_0 \ donné \\ x_k = x_{k-1} - \dfrac{f(x_{k-1})}{f'(x_{k-1})} \end{cases}$$

a. Interprétation graphique de la méthode

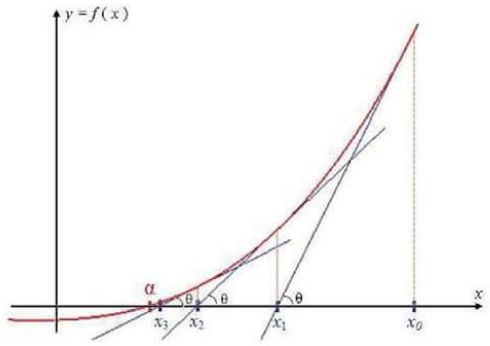


Figure 1.1. Méthode de Newton-Raphson

$tg\theta$ Représente la dérivée:

$$f'(x_k) = \frac{f(x_{k-1}) - 0}{x_{k-1} - x_k}$$

Exemple 2 :

Trouver par la méthode de Newton-Raphson la valeur approximative de la racine de $x^5 - x + 2 = 0$ à partir de $x_0 = -1$ pour une précision $\varepsilon = 10^{-6}$

Solution 2:

$f(x) = x^5 - x + 2 = 0 \Rightarrow f'(x) = 5x^4 - 1$

En appliquant la formule de Newton-Raphson :

$$\begin{cases} x_0 \ donné \\ x_k = x_{k-1} - \dfrac{f(x_{k-1})}{f'(x_{k-1})} \end{cases}$$

nous obtenons: $x_k = x_{k-1} - \frac{x_{k-1}^5 - x_{k-1} + 2}{5x_{k-1}^4 - 1}$

Alors en partant de $x_0 = -1$, on obtient les itérations illustrées dans le tableau suivant:

K	x_k
0	-1.000000
1	-1.500000
2	-1.331620
3	-1.273516
4	-1.267237
5	-1.261768
6	-1.267168

$|x_k - x_{k-1}| = |x_6 - x_5| \leq \varepsilon$ donc la racine approchée de l'équation $f(x) = 0$ est $c \simeq -1{,}267168$

b. Algorithme de la méthode

Pas 1 : k $\leftarrow 1$

Pas2 : $x_k \longleftarrow x_{k-1} - \frac{f(x_{k-1})}{f\prime(x_{k-1})}$

Pas 3 : Si $|x_k - x_{k-1}| \leq \varepsilon$ Alors on arrête ($x_k$racine approchée)

Sinon Pas 4 : K $\longleftarrow K + 1$

Pas 5 : Retour au pas 2

c. Convergence et estimation de l'erreur

Théorème:

On démontre que si f est définie sur l'intervalle $[a, b]$ tel que:

1. $f(a).f(b) < 0$
2. $\forall\, x \in [a, b] \quad f'(x) \neq 0$
3. $\forall\, x \in [a, b] \quad f"(x) \neq 0$

Alors la méthode de Newton- Raphson engendre une suite qui converge vers la solution unique de $f(x) = 0$, en partant de l'approximation x_0 vérifiant:

$$f"(x_0).\, f(x_0) > 0$$

On démontre aussi que l'erreur commise en utilisant la méthode de Newton-Raphson comme outil de résolution s'écrit:

$$|\alpha - x_k| \leq \frac{M(x_k - x_{k-1})^2}{2m}$$

Avec $M = Max\{|f''(x)|\}$, $x \in [a, b]$

$m = min\{|f''(x)|\}$, $x \in [a, b]$

4. Méthode de point fixe

Avant d'aborder la méthode du point fixe, il est important de définir ce que signifie un point fixe d'une fonction.

a. Définition

Soit la fonction $g(x)$ définie dans l'intervalle $[a, b]$. Tout point $c \in [a, b]$ tel que : $c = g(x)$ est dit point fixe de $g(x)$.

L'équation $f(x) = 0$, avec f continue sur $[a, b]$ peut être mise sous la forme $x = g(x)$ tel que:

$$f(x) = x - g(x).$$

Le choix d'une valeur initiale de la racine x_0 permet d'avoir une première approximation x_1 Tel que :$x_1 = \mathrm{g}(x_0)$ puis une meilleure approximation x_2 tel que : $x_2 = \mathrm{g}(x_1)$ ainsi on obtient une suite définie par la relation récursive suivante:

$$\begin{cases} x_0 \\ x_k = \mathrm{g}(x_{k-1}) \end{cases}$$

Exemple 3

Soit à Résoudre l'équation $x^3 - 2 = 0$ à partir de la valeur initiale $x_0 = 1.2$

Solution 3:

Il est possible de transformer l'équation précédente en plusieurs formes $x = g(x)$

Par exemple :

a. On a $x^3 - 2 = 0$

$x^3 - 2 + x = x = g_1(x)$

$$x_0 = 1.200$$
$$x_1 = 0.928$$
$$x_2 = -0.273$$
$$x_3 = -2.293$$
$$x_4 = -16.349$$

en plus on sait que $x^3 - 2 = 0 \Longrightarrow x^3 = 2 \Longrightarrow x = 2^{1/3}$

Alors on remarque qu'on s'éloigne de la racine exacte $2^{1/3} \Longrightarrow$ Divergence.

b. On a $x^3 - 2 = 0$

$$x^3 - 2 - 5x = -5x$$
$$\frac{x^3 - 2 - 5x}{-5} = x$$

$x = g_2(x) = (2 + 5x - x^3)/5$ on obtient:

$$x_0 = 1.200$$
$$x_1 = 1.2544$$

$$x_2 = 1.2596$$
$$x_3 = 1.2599$$
$$x_4 = 1.2599$$

On remarque qu'on s'approche rapidement de la racine exacte $2^{1/3} \Longrightarrow$ Convergence
De là on conclure que le choix de la forme $x = g(x)$ est capital dans la détermination de la convergence ou la divergence de la méthode du point fixe.

b. Algorithme de la méthode

Pas 1 : $k \leftarrow 1$
Pas 2: $x_k \leftarrow g(x_{k-1})$
Pas 3: Si $|x_k - x_{k-1}| \leq \varepsilon$ alors on arrête (x_k racine approchée)
Sinon Pas 4: : $k \leftarrow k + 1$
Pas 5: Retour au pas 2

c. Etude de la convergence de la méthode

Théorème:
On démontre que si $g: [a, b] \rightarrow [a, b]$ possède un point fixe dans l'intervalle $[a, b]$ et si $|g'(x)| \leq k < 1$ ce point est unique.

Exemple 4:

On montre que la fonction $g(x) = \frac{x^2-1}{3}$ possède un point fixe unique dans l'intervalle $[-1,1]$.

Solution 4:

Division l'intervalle en 2 parties : $[-1,0]$ et $[0,1]$

$$g(-1) = g(1) = 0 \quad \in \quad [-1,1]$$
$$g(0) = -\frac{1}{3} \quad \in \quad [-1,1]$$

Et nous avons :

$\forall\ x \in [-1,0]$: $g(x) \searrow \Rightarrow g(0) = -\frac{1}{3} \leq g(x) \leq g(-1) = 0$

$\forall\ x \in [0,1]$: $g(x) \nearrow \Rightarrow g(0) = -\frac{1}{3} \leq g(x) \leq g(1) = 0$

Donc $g: [-1,1] \rightarrow [-1,1]$

En plus $|g'(x)| = |2x/3| \leq \frac{2}{3} < 1$

On conclut que le point fixe est unique dans $[-1,1]$,

Théorème:

On démontre que si la fonction $g: [a, b] \rightarrow [a, b]$

vérifie $|g'(x)| \leq k < 1 \quad \forall \quad x \in \quad [a, b]$, alors la suite définie par la relation récursive suivante:

$$\begin{cases} x_0 \\ x_k = g(x_{k-1}) \end{cases}$$

est convergente et converge vers le point fixe unique de g dans $[a, b]$

d. Estimation de l'erreur

On démontre que l'erreur commise en utilisant la méthode du point fixe comme outil de résolution vérifie la relation:

$$|x_n - \alpha| \leq \frac{k}{1-k} |x_n - x_{n-1}|$$

Où

$$\begin{cases} \alpha: la\ solution\ exacte \\ x_n: la\ solution\ approchée \\ k = \dfrac{|x_n - x_{n-1}|}{|x_n - x_{n-2}|} \end{cases}$$

Mise en œuvre sous Matlab

Ecrire un programme sous Matlab qui permet de trouver la racine de: $f(x) = 2x^2 - x - 1$ sur l'intervalle $[0.5\ ,\ 1.5]$ en utilisant la méthode de dichotomie, la méthode de Newton-Raphson et la méthode de point fixe jusqu'à la convergence avec une précision de 10^{-3} et $x_0 = 0.8$.

Chapitre 6 : Résolution des équations linéaires (Méthodes directes)

1. Introduction:

Dans ce chapitre, nous allons aborder deux principales méthodes de résolution des systèmes linéaires, à savoir:

- La méthode de Carmer
- La méthode d'élimination de Gauss
- La méthode Gauss avec pivot

De façon générale, la résolution d'un système d'équations linéaires consiste à trouver un vecteur $\vec{X} = [\, x_1 \;\; x_2 \;\; x_3 \; \dots \; x_n]^{\,T}$

($\vec{X}$ dénotera un vecteur colonne et l'indice supérieur T désignera sa transposée) solution de:

$$\begin{cases} a_{11}\, x_1 + a_{12}\, x_2 + a_{13}\, x_3 + \cdots + a_{1n}\, x_n = b_1 \\ a_{21}\, x_1 + a_{22}\, x_2 + a_{23}\, x_3 + \cdots + a_{2n}\, x_n = b_2 \\ \vdots \\ a_{n1}\, x_1 + a_{n2}\, x_2 + a_{n3}\, x_3 + \cdots + a_{nn}\, x_n = b_n \end{cases}$$

On peut utiliser la notation matricielle, qui est beaucoup plus pratique et surtout plus compacte, On écrit alors le système précédent sous la forme:

$$A\,\vec{X} = \vec{b}$$

Où: A est la matrice :

Et $\vec{b} = [b_1 \;\; b_2 \;\; b_3 \;\; \dots b_n]^{,T}$ Bien entendu, la matrice A et le vecteur $\vec{b}$ sont connus.

Il reste à déterminer le vecteur $\vec{X}$.

2. Méthode de Cramer:

La méthode de Cramer est basée sur le calcul du déterminant de la matrice A et les déterminants associés aux inconnues x_i ($i = 1, \dots, n$)

a. Solution utilisant les déterminants

$\det A$: Déterminant de la matrice A, avec $\det A \neq 0$

$\det i$: Déterminant associé à l'inconnue x_i

$$x_i = \frac{deti}{\det A}$$

Soit le système à 3 équations :

$$\begin{cases} a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 = b_1 \\ a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 = b_2 \\ a_{31}\,x_1 + a_{32}\,x_2 + a_{33}\,x_3 = b_3 \end{cases}$$

$$\det A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$det1 = \begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix} \quad \text{et} \quad x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}}$$

$$det2 = \begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix} \quad \text{et} \quad x_2 = \frac{\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}}$$

$$det3 = \begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{33} & b_3 \end{vmatrix} \quad \text{et} \quad x_3 = \frac{\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{33} & b_3 \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}}$$

Remarque:

Si $\mathbf{det}\,\boldsymbol{A} = |\boldsymbol{A}| = \mathbf{0} \implies$ le nombre solutions est infini ou inexistant.

b. Solution utilisant la matrice inverse $\boldsymbol{A^{-1}}$

Si $\mathbf{det}\,\boldsymbol{A} = |\boldsymbol{A}| \neq \mathbf{0} \implies \boldsymbol{A^{-1}}$ existe.

Soit le système d'équations $\boldsymbol{AX} = \boldsymbol{B}$ *;* multiplions les 2 membres par $\boldsymbol{A^{-1}}$ (l'inverse de $\boldsymbol{A}$) *:*

$\boldsymbol{A^{-1}\,AX} = \boldsymbol{A^{-1}\,B}$ *cela devient :* $\boldsymbol{X} = \boldsymbol{A^{-1}\,B}$

Pour calculer l'inverse de $\boldsymbol{A}$*:*

$\boldsymbol{A^{-1}} = \frac{\mathbf{1}}{\boldsymbol{detA}}\,\boldsymbol{C^T}$ avec $\boldsymbol{C^T}$ est le transposé de la comatrice A

Remarque:

La méthode de Cramer exige un grand nombre d'opérations de calcul $\left((\boldsymbol{n^2}+\boldsymbol{n})\boldsymbol{n}! - \mathbf{1}\right)$.

Si n est élevé, le nombre d'opérations augmente et par conséquent le temps de calcul. De plus, pour les moyens et grands systèmes l'erreur cumulée d'arrondi augmente avec le

nombre d'opérations et altère la précision des résultats. On va aborder d'autres méthode qui nécessitent

Un nombre limité d'opérations de calcul, donc rapides et plus précises.

3. Méthode de Gauss

a. Principe

Cette méthode est basée sur la transformation du système linéaire $\boldsymbol{A}\,\vec{\boldsymbol{X}} = \vec{\boldsymbol{b}}$

En un système équivalent $\boldsymbol{A'}\,\vec{\boldsymbol{X}} = \vec{\boldsymbol{b'}}$ tel que la matrice $\boldsymbol{A'}$ est une matrice triangulaire supérieure.

La transformation de la matrice $\boldsymbol{A}$ en $\boldsymbol{A'}$ et le vecteur $\boldsymbol{b}$ en $\boldsymbol{b'}$ passe par plusieurs étapes nous présentons à travers l'exemple suivant:

Pour une meilleure praticabilité, on forme la matrice $\widetilde{\boldsymbol{A}}$ telle que $\widetilde{\boldsymbol{A}} = [\boldsymbol{A} : \vec{\boldsymbol{b}}]$ et qu'on appelle matrice augmentée de $\boldsymbol{A}$.

Exemple 1:

Soit à résoudre par Gauss le système linéaire suivant:

$$\begin{cases} \boldsymbol{x_1 + 3x_2 + 3x_3 = 0} \\ \boldsymbol{2x_1 + 2x_2 = 2} \\ \boldsymbol{3x_1 + 2x_2 + 6x_3 = 11} \end{cases}$$

Solution 2:

Soit à résoudre le système suivant :

$$\begin{cases} \boldsymbol{x_1 + 3x_2 + 3x_3 = 0} \\ \boldsymbol{2x_1 + 2x_2 = 2} \\ \boldsymbol{3x_1 + 2x_2 + 6x_3 = 11} \end{cases}$$

Où:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{3} & \boldsymbol{3} \\ \boldsymbol{2} & \boldsymbol{2} & \boldsymbol{0} \\ \boldsymbol{3} & \boldsymbol{2} & \boldsymbol{6} \end{bmatrix} \; ; \; \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \boldsymbol{x_3} \end{pmatrix} \; ; \boldsymbol{b} = \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{2} \\ \boldsymbol{11} \end{pmatrix}$$

On adopte l'écriture suivante:

$$\widetilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{3} & \boldsymbol{3} & \vdots & \boldsymbol{0} \\ \boldsymbol{2} & \boldsymbol{2} & \boldsymbol{0} & \vdots & \boldsymbol{2} \\ \boldsymbol{3} & \boldsymbol{2} & \boldsymbol{6} & \vdots & \boldsymbol{11} \end{bmatrix} \begin{matrix} \boldsymbol{E_1} \\ \boldsymbol{E_2} \\ \boldsymbol{E_3} \end{matrix}$$

Etape 1: puisque le pivot $\boldsymbol{a_{11}^{(0)} = 1 \neq 0}$

Alors on élimine de $\boldsymbol{x_1}$ de $\boldsymbol{E_2}$ et $\boldsymbol{E_3}$

$$\boldsymbol{E_2 \leftarrow E_2 - \frac{2}{1}E_1} \quad , \boldsymbol{E_3 \leftarrow E_3 - \frac{3}{1}E_1}$$

$$\widetilde{\boldsymbol{A}}^{(1)}=\begin{bmatrix} \boldsymbol{1} & \boldsymbol{3} & \boldsymbol{3} & \vdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{-4} & \boldsymbol{-6} & \vdots & \boldsymbol{2} \\ \boldsymbol{0} & \boldsymbol{-7} & \boldsymbol{-3} & \vdots & \boldsymbol{11} \end{bmatrix} \begin{matrix} \boldsymbol{E_1} \\ \boldsymbol{E_2} \\ \boldsymbol{E_3} \end{matrix}$$

Etape 2 : puisque le pivot $\boldsymbol{a_{22}^{(1)} = -4 \neq 0}$

Alors on élimine de $\boldsymbol{x_2}$ de $\boldsymbol{E_3}$

$$\boldsymbol{E_3 \leftarrow E_3 - \left(\frac{-7}{-4}\right)E_2}$$

$$\widetilde{\boldsymbol{A}}^{(2)}=\begin{bmatrix} \boldsymbol{1} & \boldsymbol{3} & \boldsymbol{3} & \vdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{-4} & \boldsymbol{-6} & \vdots & \boldsymbol{2} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{15/2} & \vdots & \boldsymbol{15/2} \end{bmatrix} \begin{matrix} \boldsymbol{E_1} \\ \boldsymbol{E_2} \\ \boldsymbol{E_3} \end{matrix}$$

Etape 3 : comme le pivot $\boldsymbol{a_{33}^{(2)} = 15/2 \neq 0}$

$$\tilde{A}^{(3)}=\begin{bmatrix} 1 & 3 & 3 & \vdots & 0 \\ 0 & -4 & -6 & \vdots & 2 \\ 0 & 0 & 1 & \vdots & 1 \end{bmatrix} \begin{matrix} E_1 \\ E_2 \\ E_3 \end{matrix}$$

Par substitution inverse on obtient:

$$\boldsymbol{x_3 = 1}$$

$$\boldsymbol{-4x_2 - 6x_3 = 2 \Longrightarrow x_2 = -2}$$

$$\boldsymbol{x_1 + 3x_2 + 3x_3 = 0 \Rightarrow x_1 = 3}$$

Donc, la solution est :

$$\vec{X}=\begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix}$$

Et le déterminant de A est donné par :

$$\boldsymbol{\det(A) = \prod_{i=1}^{3} a_{ii}^{(i-1)} = a_{11}^{(0)} * a_{22}^{(1)} * a_{33}^{(2)} = 1 \times (-4) \times \frac{15}{2} = -30}$$

b. Algorithme

1. Triangularisation

$$\omega = \frac{a_{ik}}{a_{kk}}$$

$$a_{ij} = a_{ij} - \omega a_{kj}\ ;\quad j = k+1\ ,\ n+1\ ;\ \ \mathrm{i} = \mathrm{k}+1, \mathrm{n}\ ; \mathrm{k} = 1, \mathrm{n}-1$$

2. Substitution inverse

$$x_i = \frac{\left(b_i - \sum_{j=i+1}^{n} a_{ij} x_i\right)}{a_{ii}};\quad i = n, n-1, \dots, 1$$

4. Méthode de Gauss avec pivot:

Dans le processus d'élimination de la méthode de Gauss, on a supposé à chaque étape k $(k = \overline{1, n-1})$ que l'élément $a_{kk} \neq 0$, mais cette supposition n'est pas toujours vraie. Si $a_{kk} = 0$, on cherche l'équation E_y parmi celles qui suivent E_k où l'élément $a_{jk} \neq 0$ pour faire la permutation $\left(E_k \leftrightarrow E_j\right)$ et ainsi avoir $a_{kk} \neq 0$.

Aussi, on remarque que la division par de petites valeurs génère une grande erreur ce qui nous amène à choisir a_{kk} le plus grand en valeur absolue , c.à.d.:

$\left|a_{jk}\right| = Max\{|a_{ik}|\}\ ;\ k \leq i \leq n$

Après le choix de a_{kk}, on poursuit normalement l'étape en cours selon Gauss.

Mise en œuvre sous Matlab

Ecrire un programme sous Matlab qui permet de résoudre le système des équations linéaires suivant en utilisant la méthode d'élimination de Gauss.

$$\begin{cases} 2x_1 + x_2 + 2x_3 = 10 \\ 6x_1 + 4x_2 = 26 \\ 8x_1 + 5x_2 + x_3 = 35 \end{cases}$$

Chapitre 7 : Résolution des équations linéaires (Méthodes itératives)

1. Introduction:

On a vu que les méthodes directes donnent la solution exacte du système d'équations linéaires, cependant elles restent gourmandes en mémoire. Dans ce chapitre on va introduire les méthodes itératives ou indirectes qui donnent une solution approximative du système d'équations linéaires. Ces méthodes sont très faciles à mettre en œuvre et à programmer, elles ne consomment pas la mémoire et donnent des résultats autant précis que l'on veut.

2. Méthode Jacobi

Soit le système d'équations suivant:

$$\begin{cases} \boldsymbol{a_{11}\, x_1 + a_{12}\, x_2 + a_{13}\, x_3 + \cdots + a_{1n}\, x_n = b_1} \\ \boldsymbol{a_{21}\, x_1 + a_{22}\, x_2 + a_{23}\, x_3 + \cdots + a_{2n}\, x_n = b_2} \\ \vdots \\ \boldsymbol{a_{n1}\, x_1 + a_{n2}\, x_2 + a_{n3}\, x_3 + \cdots + a_{nn}\, x_n = b_n} \end{cases} \quad (1)$$

Transformons le système en supposant que les éléments du pivot sont non nuls $a_{ii} \neq 0 \quad i = 1,2, \dots n$

$$\begin{cases} x_1 = (b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n)/a_{11} \\ x_2 = (b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n)/a_{22} \\ \vdots \\ x_n = \left(b_n - a_{n2}x_2 - a_{n3}x_3 - \cdots - a_{n,n-1}x_{n-1}\right)/a_{nn} \end{cases} \quad (2)$$

Cette forme est appelée forme réduite du système $Ax = b$. Elle peut s'écrire autrement

$x_i = \left[b_i - \sum_{j=1}^{n} a_{ij}x_j\right]/a_{ii} \qquad ; i = 1, \dots n$

Pour résoudre le système (1) on utilise l'écriture (2) en portant les termes de droite à l'itération (k) et ceux à gauche à l'itération (k).

$$\begin{cases} x_1^{(k+1)} = \left(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}\right)/a_{11} \\ x_2^{(k+1)} = \left(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}\right)/a_{22} \\ \vdots \\ x_n^{(k+1)} = \left(b_n - a_{n2}x_2^{(k)} - a_{n3}x_3^{(k)} - \cdots - a_{n,n-1}x_{n-1}^{(k)}\right)/a_{nn} \end{cases} \quad (3)$$

En prenant une estimation initiale $X^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}\right)$ et en utilisant le système (3) on calcule $X^{(1)} = \left(x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}\right)$ensuite on remplace le vecteur $X^{(1)}$dans le système (3) avec k=1 on calcule $X^{(2)}$et continue de la même façon de calculer les vecteurs $X^{(3)}$, $X^{(4)}$, $X^{(5)}$,… jusqu'à la convergence.

Le processus itératif aura lieu en utilisant un vecteur initial $X^{(0)}$ et la forme réduite peut être formulée comme suite :

$X_i^{(k+1)} = \left[b_i - \sum_{j=1}^{n} a_{ij}X_j^{(k)}\right]/a_{ii}$; $i = 1, \dots, n$ et $i \neq j$

3. Méthode de Gauss-Seidel:

La méthode de Gauss-Seidel est une amélioration de la méthode de Jacobi en effet elle rend le processus itératif plus rapide.

Soit le système des 3 équations à trois inconnues:

$$\begin{cases} \boldsymbol{a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 = b_1} \\ \boldsymbol{a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 = b_2} \\ \boldsymbol{a_{31}\,x_1 + a_{32}\,x_2 + a_{33}\,x_3 = b_3} \end{cases}$$

Ce système peut s'écrit avec les éléments du pivot sont non nuls $a_{ii} \neq 0 \;\; et \;\; i = 1,2, \dots n$

$$\begin{cases} x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{cases}$$

À la première itération, on calcule à partir du vecteur initial : $x^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, x_3^{(0)}\right)$

Les valeurs de x de la première itération se calculent ainsi.

$$\begin{cases} x_1^{(1)} = \left(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)}\right)/a_{11} \\ x_2^{(1)} = \left(b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(0)}\right)/a_{22} \\ x_3^{(1)} = \left(b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)}\right)/a_{33} \end{cases}$$

Et on continue jusqu'à aboutir à une précision suffisante

Le système réduit reste le même sauf que la valeur nouvelle de $X_i^{(k+1)}$obtenue dans la ième équation sera injectée dans la suivante:

$$\begin{cases} X_1^{(k+1)} = \left(b_1 - a_{12}X_2^{(k)} - a_{13}X_3^{(k)} - \cdots - a_{1n}X_n^{(k)}\right)/a_{11} \\ X_2^{(k+1)} = \left(b_2 - a_{21}X_1^{(k+1)} - a_{23}X_3^{(k)} - \cdots - a_{2n}X_n^{(k)}\right)/a_{22} \\ \vdots \\ X_n^{(k+1)} = \left(b_n - a_{n1}X_1^{(k+1)} - a_{n2}X_2^{(k+1)} - \cdots - a_{n,n-1}X_{n-1}^{(k+1)}\right)/a_{nn} \end{cases}$$

Le processus itératif se fera avec un vecteur initial et l'utilisation de la formule

$X_i^{(k+1)} = \left[b_i - \sum_{j=1}^{i-1} a_{ij}X_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}X_j^{(k)}\right]/a_{ii}$; $i = 1, \ldots, n$ et $i \neq j$

La méthode de Gauss-Seidel est une variante améliorée de la méthode de Jacobi. En effet, à l'itération $k+1$, au moment du calcul de $X_2^{(k+1)}$,on possède déjà une meilleure approximation de X_1 que $X_1^{(k)}$, à savoir $X_1^{(k+1)}$. De même, au moment du calcul de $X_3^{(k+1)}$, on peut utiliser $X_1^{(k+1)}$ et $X_2^{(k+1)}$ qui ont été déjà calculés. Plus Généralement, pour le calcul de $X_i^{(k+1)}$,on peut utiliser $X_1^{(k+1)}$, $X_2^{(k+1)}$,…, $X_{i-1}^{(k+1)}$ déjà calculés et les $X_1^{(k+1)}$, $X_2^{(k+1)}$,…, $X_{i-1}^{(k+1)}$ de l'itération précédente.

4. Critère d'arrêt

On utilise le plus souvent les critères suivants:

- $\left|x_i^{(n)} - x_i^{(n-1)}\right| \leq \varepsilon$ $\quad i = \overline{1,n}$
- Ou $\left\|\vec{x}^{(n)} - \vec{x}^{(n-1)}\right\| \leq \varepsilon$ (erreur absolue)
- Ou $\frac{\left\|\vec{x}^{(n)} - \vec{x}^{(n-1)}\right\|}{\left\|\vec{x}^{(n)}\right\|} \leq \varepsilon$ (erreur relative)

5. Condition de convergence :

On démontre que si A est une matrice à diagonale strictement dominante (condition suffisante), la méthode de Jacobi et de Gauss-Seidel sont convergentes. C.à.d.:

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} \left|a_{ij}\right| < |a_{ii}| \qquad \forall\ i = \overline{1,n}$$

Mise en œuvre sous Matlab

Ecrire un programme sous Matlab qui permet de résoudre le système des équations linéaires suivant en utilisant la méthode de Gauss-Seidel.

Avec la solution estimée $X^{(0)} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$ et erreur $\varepsilon = 10^{-4}$

Références

1. Nicolas Hudon. Initiation à Matlab, (nicolas.hudon@polymtl.ca), URCPC, Ecole Polytechnique de Montréal, 22 janvier 2004

2. Marie Postel. Introduction au logiciel Matlab, Version révisée septembre 2004. Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie.

3. Alfio Quarteroni, Fausto Saleri, Paola Gervasio. Calcul Scientifique; Cours, exercices corrigés et illustrations en MATLAB et Octave. Springer-Verlag, Italie 2010.

4. Baba Hamid Fatima Zohra. Le Calcul scientifique appliqué au Génie Civil sous Matlab, Cours. Université des Sciences et de la Technologie d'Oran Mohamd Boudiaf