© *Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>*

Thème - 1 *Notions de base sur Matlab*

Table des matières

1 L'environnement Matlab

`Matlab` est un outil de développement pratique de par ses nombreuses bibliothèques et son environnement de développement intégré.

Il est pour l'étudiant un outil de premier recours pour tester de nouveaux algorithmes grâce à la possibilité offerte de s'affranchir des soucis comme, sans être exhaustif, l'inversion de systèmes linéaires, le calcul de valeurs propres (bref tout ce qui a trait à l'algèbre linéaire), la recherche de solutions approchées des équations non-linéaires et différentielles ordinaires.

Pour des développements poussés, **`Matlab`** dispose d'un bon outil de *profilage* et de *déboggage* et d'une bonne bibliothèque de fonctions graphiques. Ces avantages font de Matlab un environnement garantissant un gain en temps certain dans une activité de recherche académique.

Cependant **`Matlab`** offre un langage interprété, avec un typage dynamique. Ceci peut poser des soucis en terme d'efficacité. Fort heureusement des solutions sont offertes (pas toujours évidentes du premier abord) d'interfaçage avec d'autres langages, notamment compilés et à typage statique, comme Fortran, C, C++ etc.

Bien que la syntaxe qu'offre **`Matlab`** soit simple et soutienne un paradigme impératif, un conseil est prodigué pour une programmation avec une vision vectorielle de l'exécution.

Le présent document présente l'outil **`Matlab`** de manière succincte, en laissant au lecteur de soin d'approfondir ses connaissances à travers la dense ressource documentaire existante.

2 Matlab comme langage de scripts

2.1 Accès à l'aide

Lorsque vous recherchez de l'aide sur une fonction précise de Matlab, vous pouvez simplement taper la commande :

```
>> help nom_de_la_fonction
```

Exemple

```
>> help det
```

Essayer aussi

```
>> lookfor det
```

```
>> helpwin
```

2.2 Création, modification de matrices, accès aux éléments, gestion de la mémoire.

2.2.1 Créer une matrice

On peut créer une matrice A explicitement :

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

La matrice est délimitée par des crochets, on entre les éléments ligne par ligne, avec un point-virgule pour séparer les lignes de la matrice. De la même façon, on peut créer un vecteur ligne :

```
>> B = [1 5]
```

On accède aux éléments d'une matrice en spécifiant entre parenthèses et séparés d'une virgule, la ligne et la colonne de l'élément désiré :

```
>>  A(2,3)
```

Pour les scalaires, on peut ne pas spécifier ni la ligne, ni la colonne :

```
>>  F(1,1)
>>  F(1)
>>  F
```

2.2.2 Modifier une matrice

Il est possible de modifier un des éléments d'une matrice :

```
>> A(1,3) = 2
```

Si on modifie un élément inexistant d'une matrice, la matrice est agrandie jusqu'à ce que cet élément existe :

```
>>  A(3,3) = 1
```

2.2.3 Gestion de la mémoire

La facilité avec laquelle on crée de nouvelles variables dans Matlab fait que la mémoire risque vite d'être encombrée avec une multitude de variables dont on ne se sert plus. La commande **who** donne la liste des variables de l'espace de travail. La commande **whos** donne la même liste, ainsi que des informations sur la nature et la taille de chaque variable.

```
>> who
>> whos
```

Pour effacer une variable et libérer la mémoire qu'elle occupait, il faut effectuer la commande :

```
clear nom_de_la_variable
>> clear F
>> who
```

La commande **clear all** efface toutes les variables de l'espace de travail. Il n'est pas possible d'annuler (undo) la commande clear.

2.2.4 Affichage

Lorsqu'on souhaite que le résultat d'une commande ne s'affiche pas, il faut placer un point-virgule à la fin de celle-ci :

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

Aucun résultat n'est affiché, pourtant, la commande a bien été exécutée.

```
>>  T
```

2.3 Opérations élémentaires sur les matrices.

Pour cette partie, on réutilisera les matrices qui ont été définies à la partie précédente (les bases).

2.3.1 Opérations sur les scalaires

Matlab permet d'effectuer les opérations classiques sur les scalaires :

```
>> toto = (E-a)^a * a/E + 1
```

Les fonctions classiques comme sin, cos, log, ... existent aussi. (essayez helpwin ops, ou helpwin elfun)

2.3.2 Opérations sur les matrices

Les opérations $+, -, *, /$ existent aussi pour les matrices. S'il n'y a pas d'ambiguïtés pour l'addition et la soustraction, il faut faire attention pour les opération de multiplication, division, puissance,.... Exemples :

```
>>  a*A
```

Les matrices doivent avoir des dimensions compatibles !

2.3.3 Résolution de systèmes linéaires

Une façon de résoudre un système linéaire $A * x = C$, est de calculer l'inverse de A :

```
>> x = inv(A)*C
```

Cette façon de procéder peut être très imprécise si la matrice A est mal conditionnée. En outre calculer l'inverse de la matrice demande beaucoup plus d'opérations qu'il n'en faut pour résoudre le système uniquement. Une meilleure façon de procéder est d'utiliser l'opérateur de division à gauche de Matlab (\) Matlab utilise alors un ensemble de méthodes mieux appropriées pour résoudre le système (Par exemple la factorisation LU de la matrice, suivie de la résolution de deux systèmes triangulaires).

```
>> x = A \ C
```

IL faut noter que cette approche (par backslash) fait appel aux méthodes directes pour résolutions de systèmes linéaires (pleins ou creux). Si l'on souhaite utiliser les méthodes itératives de type Krylov (connues pour être moins gourmandes en mémoire) il faut choisir à sa guise parmi celles disponibles : **`pcg, cgs, bicg, bicgstab, gmres, ...`**. Pour se renseigner sur ces méthodes on pourra utiliser l'aide de Matlab :

```
>> help pcg
```

2.3.4 Concaténation et extraction

Il est possible de créer de nouvelles matrices en concaténant des matrices déjà existantes. La syntaxe est similaire à celle de la création de matrices à partir de scalaires.

```
>> [A C;1 T]
```

Ici, on a extrait la sous matrice composée de la deuxième ligne de A et des colonnes 1 et 3.

```
>> A(2,[3 1 2])
```

2.4 L'opérateur *colon* " :"

L'opérateur " :" (*colon*) est un opérateur permettant de générer des suites de nombres. Il est très utile pour extraire des sous-matrices, gérer les boucles d'itérations (voir la partie : structures de contrôle), ...

2.4.1 Générer une suite de nombres

```
>> a = -2:3
>> a = fliplr([-2:3])
```

2.4.2 Sélectionner une sous-matrice

```
>> B = A(1:3,1:3)
>>  A(:,2)
```

Il faut lire : A(toutes les lignes, deuxième colonne). Cette notation est équivalente à :

```
>> A(1:size(A,1),2)
```

2.4.3 Tracer le graphe d'une fonction

Un moyen simple de visualiser le graphe d'une fonction est d'évaluer cette fonction en un certain nombres de points, et de relier ces points par des segments de droite. Visualisons la fonction $\sin(x)$ pour x variant de 0 à 12.

```
>> x = [0:0.05:12];
>> y = sin(x);
>> plot(x,y)
```

A la première ligne, on génère des valeurs de 0 à 12, espacées de 0.05. Ensuite, on évalue la fonction sinus en ces points. Enfin on affiche ces points, reliés par des lignes.

2.4.4 Apprendre plus "avec l'aide intégrée"

Si vous voulez de l'aide sur l'opérateur " :", tapez

```
>> help colon
```

essayez aussi : **`help linspace, help logspace`**

3 Matlab comme langage impératif

Le langage de Matlab n'aurait pas beaucoup d'intérêt s'il n'était pas possible d'effectuer des tests, de programmer des boucles, etc., comme dans la majorité des langages de programmation. L'aide concernant les structures de contrôle est accessible via :

`helpwin lang`

3.1 *Opérations booléennes*

Les valeurs booléennes "vrai" et "faux" sont respectivement représentées dans Matlab par les valeurs 1 et 0. Les opérateurs booléens de Matlab sont décrits à la rubrique d'aide relop (helpwin relop). On a les opérateurs :

Listing 1 – Opérateurs booléens

```
> , >= , < , <= , == (égal), ~= (différent), & (et), | (ou), ~ (non), xor (ou exclusif).
```

Exemple :

```
>> a = 2; b = 3;
>> a > b
>> a >= a
>> a == a
```

Attention : Lorsque vous écrivez une expression qui combine des opérations arithmétiques et des opérations booléennes, veillez à bien utiliser les parenthèses.

3.2 Structures de contrôle

Instruction IF

La structure if est similaire à celle de nombreux langages de programmation **(C, C++, Java,...) :**

Listing 2 – Instruction IF

```
if expression


    statements2


end
```

Listing 3 – Instruction IF ELSE

```
if expression

    statements1

else
    statements2
end
```

Listing 4 – Instruction IF ELSE IF

```
if expression1
     statements1
elseif expression2
     statements2
else
     statements3
end
```

Si l'expression expression1 est vraie, alors les commandes statements1 sont exécutées, sinon, si l'expression expression2 est vraie alors les commandes statements2 sont exécutées, enfin, si ni expression1 ni expression2 ne sont vraies, alors ce sont les commandes statements3 qui sont exécutées. Les clauses ELSEIF et ELSE sont optionnelles.

Attention : Une erreur fréquente est d'écrire un THEN après expression1. THEN n'existe pas dans Matlab

Instruction WHILE

Listing 5 – Instruction WHILE

```
while expression
    statements
end
```

La structure while est une structure puissante permettant de répéter une séquence d'instructions, tant qu'une certaine condition est vérifiée :

Tant que expression est vraie, alors les commandes statements sont exécutées. A titre d'exemple, calculons la somme des carrés des entiers de 1 à 11.

Listing 6 – Instruction WHILE appliquée

```
i = 0;
somme = 0;
while (i <= 10)
  i = i+1;
  somme = somme + i*i;
end
somme  %affichage
```

On aurait aussi pu se passer de l'utilisation d'une boucle :

```
>> sum([1:11].^2)
```

Il faut faire attention à ne pas créer de boucle sans fin. Exemple :

Listing 7 – Instruction WHILE appliquée

```
m = 1;
```

```
while (m > 0)
  m = m + 1
end
```

Cette boucle ne s'arrêtera jamais. On peut toujours stopper une commande Matlab en cours d'exécution en tapant : $[CTRL] + C$ (combinaisons des touches Ctrl et C).

Instruction FOR

Listing 8 – Instruction FOR

```
for variable = expr
   statement
end
```

Si n est la longueur du vecteur représenté par expr , les commandes statement seront exécutées n fois, lors de la première exécution, variable aura la valeur expr(1), ... et ainsi de suite jusqu'à la dernière fois où variable aura la valeur expr(n). A titre d'exemple, calculons le produit des entiers impairs de 1 à 11.

Listing 9 – Instruction FOR appliquée

```
p = 1;
for i=[1:2:11] % on peut aussi ecrire  i=1:2:11. i.e on va de 1 à 11 par pas de 2
  p = p*i;
end
p  %affichage
```

Sans utiliser de boucle, on aurait fait :

```
>> prod([1:2:11])
```

4 Matlab comme langage structuré

4.0.1 Scripts et fonctions

Matlab aurait une utilité fort limitée s'il n'était pas possible d'écrire des scripts et des fonctions :

`Script` : Un ensemble d'instructions, écrites dans un fichier qu'on peut exécuter en une fois. **`Fonction`** : Un ensemble d'instructions, prenant certains arguments en entrée et renvoyant un ou plusieurs résultats.

SCRIPTS ou PROCEDURES

Les scripts sont stockés dans des fichiers dont l'extension est ".m". Ils contiennent une suite d'instructions Matlab, écrites comme si on les avait entrées dans la fenêtre de Matlab. Pour exécuter un script, il faut juste taper son nom (sans l'extension ".m") dans la fenêtre de commande de Matlab.

`Exemple : monscript.m`

Listing 10 – fichier : monscript.m

```
disp('Bonjour, ceci est un script.');
disp('Appuyez sur [Espace] pour continuer');
pause;
% ceci est un commentaire
A = [1 2 3; 4 5 6]
B = [3 5 7]'
```

```
A(3,3) = 1
C = A*B;
disp('C = A*B ,.... voici le resultat');
C
```

Pour l'exécuter on entre la commande suivante :

```
>> monscript
```

Les scripts peuvent accéder aux variables qui ont été définies avant son lancement, et les variables qui ont été définies durant le script restent accessible après par l'utilisateur.

Pour plus d'informations : helpwin script

FONCTIONS

Les fonctions sont aussi stockées dans des fichiers ".m" (mfiles). Le fichier doit porter le même nom que la fonction. La syntaxe pour écrire une fonction est la suivante :

Listing 11 – Définition d'une fonction

```
function [resultat_1, ... , resultat_n] = nom_de_fonction (argument_1,..., argument_p)
 % contenu de la function ici
```

Les instructions peuvent utiliser les variables $argument_1, \ldots, argument_p$, et doivent attribuer une valeur aux variables $resultat_1, \ldots, resultat_n$.

Exemple : mafonction.m

Listing 12 – fichier : mafonction.m

```
function [somme , produit , minimum] = mafonction(a,b)
  %Ceci servira d'aide.
  %Si on tape help mafonction, ce sont les premieres lignes
  %de commentaires de la fonction qui sont affichees.
  %
  % [somme, produit, minimum] = mafonction(a,b)
  % renvoie la somme le produit et le minimum de a et b
  % ceci ne fait plus partie des commentaires d'aide
  somme = a+b;
  produit = a*b;
  % on peut mettre des structures de controle
  if (a<b)
    toto = a;
    % on peut definir de nouvelles variables comme toto.
  else
    toto = b;
  end
  minimum = toto;
```

A l'exécution, on obtient :

```
>>  [r1, r2, r3] = mafonction(5,9);
```

```
>>  help mafonction
```

Ceci servira d'aide. Si on tape help mafonction, ce sont les premières lignes de commentaires de la fonction qui sont affichées.

[somme, produit, minimum] = mafonction(a,b)

renvoie la somme le produit et le minimum de a et b

Pour plus d'informations : **`helpwin function`**

Matlab offre aussi d'autres possibilités pour écrire des fonctions

— Une première approche utilise la commande **`inline`** :

```
≫  f = inline('x*x','x')
```

Faites

```
≫ help inline
```

pour avoir plus d'information

— La seconde approche définit directement un *handle* (pointeur) sur la fonction :

```
≫  f = @(x) (x*x)
```

Cette approche offre comme avantage la possibilité de figer certains arguments d'une fonction :

```
≫  f = @(t,x,y) (t * x + cost(t) * y)
≫  y = 1; g = @(t,x) f(t,x,y);
```

On définit ainsi une fonction `g(t,x) = t * x + cost(t).`

4.1 Structuration de données

Matlab comme d'autres langages procéduraux comme le C, offre la possibilité de stocker dans une variable des des variables de types différents

4.1.1 Structures de données

Structure à la "C"

Matlab comme d'autres langages procéduraux comme le langage C, offre la possibilité de regrouper plusieurs variables pas nécessairement de même type dans une seule. Pour cette fin Matlab, on peut utiliser la commande **`struct`**. Taper **`help struct`** pour plus amples informations.

```
≫  p = struc('prenom','Jules', 'nom','Mandel', 'age', 30 )
```

Pour accéder à un champ quelconque, utiliser l'opérateur "." comme dans le langage C.

```
≫  p.prenom
≫  fprintf(1,'Mr %s %s a %d ans \n', p.prenom, p.nom, p.age);
```

Il est aussi possible de modifier la valeur d'un champ :

```
≫ p.age = 28 ;
```

Il est toujours possible d'étendre toute structure de manière dynamique. A titre d'exemple, ajoutons une fonction d'affichage à notre structure :

```
≫ p.affiche = @(a) fprintf(1,'Mr %s %s a %d ans \n', a.prenom, a.nom, a.age);
≫ p.affiche(p);
```

Pour afficher tous les champs de notre objet structure, entrer la commande :

```
≫ fieldnames(p)
```

Remarquer ici, que contrairement aux langages comme le C, nous ne définissons pas un nouveau type : nous travaillons seulement sur une instance pas sur un type (ce type n'étant connu qu'implicitement).

Ainsi, pour définir une nouvelle variable du même type que p ci-dessus, il faut opérer une copie (profonde) de p puis modifier les valeurs des champs.

```
≫  q = p; p.nom = 'Justin'; p.age = 29;
```

Il est préférable de fournir une fonction d'initialisation des champs.

Aller plus loin dans cette direction nous entraînera sur le sentier de la programmation orientée objets qui sort du cadre du présent cours. Signalons néanmoins que dans ses versions récentes Matlab propose des outils pour une programmation orientée objets.

Collection d'objets hétérogènes (i.e de types différents)

Stocker une collection d'objets de types différents est une pratique courante dans une programmation avancée. Il généralement difficile de le réaliser dans la plupart des langages de programmation à typage statique comme sont le C, C++ etc., mais pas sous Matlab qui offre pour cette fin un outil appelé **`cell`**. Entrer **`help cell`** pour plus amples informations.

```
>> a = cell(2,1);
>> a{1,1} = eye(3);
>> a{2,1} = 3:-1:0;
```

4.2 Gestion des librairies ou structuration des composants physiques

Problèmes de chemin d'accès

Matlab ne pourra exécuter un script ou une fonction que s'il sait dans quels répertoires rechercher les fichier **`.m`** (*mfiles*) correspondants. Par défaut, Matlab se cantonne à ses répertoires propres et le répertoire courant. Pour savoir quel est le répertoire courant, utilisez la commande pwd (print working directory).

```
>> pwd
```

Pour changer de répertoire, utilisez la commande cd :

```
>> cd mfiles (pour descendre dans le sous répertoire mfiles)
```

Si vous ne souhaitez pas changer de répertoire, mais désirez ajouter un répertoire à la liste des répertoire dans lesquels Matlab recherchera les scripts et fonctions, utilisez la commande addpath :

```
>>  addpath('./utils')
```
(pour ajouter le sous-répertoire utils)

5 Bon à Savoir

Pour chaque thème, vous trouverez soit un script Matlab commenté contenant plusieurs exemples, soit une page contenant quelques explications ainsi que des exemples. Téléchargez ces scripts, essayez les, modifiez les,.....

5.1 L'aide de Matlab

(indispensable si vous voulez économiser votre temps).

5.2 La vectorisation

Listing 13 – Fichier : vect.m

```
clear all

% Avec Matlab, il est possible d'éviter la programmation de nombreuses
% boucles en utilisant des notations et des fonctions vectorisées.

%Exemple 1 : on souhaite élever toutes les valeurs d'un vecteur  a au carré
a = [1:20];

%non vectorisé
```

```
b1 = zeros(size(a));
for i=1:length(a)
  b1(i) = a(i)^2;
end

%vectorisé
b2 = a.^2;
% la notation .^2 signifie qu'on applique l'opération ^2 élément par élément
% à la matrice/vecteur a.
% non seulement c'est plus concis et plus lisible, mais cette approche est
% aussi plus rapide. En effet Matlab utilise des routines
% pré-compilées pour effectuer les opératopns vectorisées.

%mesurons le temps de calcul pour un gros vecteur
% NB: La commande tic lance le chronomètre, la commande toc le stoppe et
% renvoie le temps écoulé depuis tic.

% on efface les variables :
clear a b1 b2 i

%non vectorisé:
a = [1:100000];
disp('Temps sans vectorisation :');
tic
b1 = zeros(size(a));
for i=1:length(a)
  b1(i) = a(i)^2;
end
toc

%vectorisé :
disp('Temps avec vectorisation:')
tic
b2 = a.^2;
toc

% La ligne b1 = zeros(size(a)) n'est pas inutile :
% avec cette ligne on alloue a priori la mémoire dont on aura besoin pour
%stocker la matrice b1. De cette façon Matlab n'est pas obligé, à chaque
% itération, de modifier la taille de b1 quand on rajoute un nouvel élément.
% Si on avait pas alloué à priori la matrice b1, on aurait obtenu des
% performances encore moins bonnes :

%a = [1:30000];
a = [1:100];
disp('Temps sans vectorisation et sans allocation de la mémoire à priori:');
disp('un peu de patience c''est très long')
tic
for i=1:length(a)
  b3(i) = a(i)^2;
end
toc

% vous voilà convaincu (j'espère) qu'il vaut mieux éviter les boucles FOR (ou WHILE),
% avec Matlab

%Exemple 2: Faire la combinaison linéaire de 2 lignes d'une matrice
clear all;
%A: (10*20)
A = rand(10,20);
%On veut additionner 3 fois a prelière ligne à la deuxième ligne, sauf le
% premier et le dernier élément de la ligne

[l,c] = size(A);
%sans vectorisation
for(i=2:c-1)
  A(2,i) = A(2,i)+3*A(1,i);
end

%avec vectorisation
A(2,2:c-1) = A(2,2:c-1) + 3*A(1,2:c-1);

%Exemple 3 : à chaque élément d'un vecteur on additionne celui qui le
%    précède dans le vecteur et celui sui le suit, sauf pour le premier
```

```
%    element pour lequel on ne considère que l'élément qui le suit, et le
% dernier element, pour lequel on ne considère que l'avant-dernier élément.

% exemple : [1 2 3 4]--> [(1+2) (1+2+3) (2+3+4) (3+4)]
clear all;

a = [1 2 3 4 5 6 7 8 9];

prec = 0;
for i = 1:(length(a)-1)
  nouveau = a(i)+prec+a(i+1);
  prec = a(i);
  a(i) = nouveau;
end
a(i+1) = a(i+1)+prec;

a

% avec vectorisation
a = [1 2 3 4 4 5 6 7 8]
n = length(a);

a = a + [a(2:n) 0] + [0 a(1:n-1)]
```

Lorsqu'on écrit des fonctions devant servir d'arguments pour des fonctions de Matlab (comme les solveurs d'edo), il est conseiller de les écrire sous forme vectorielle (c'est-à-dire de sorte qu'elles puissent retourner un vecteur si en entrée elles ont reçu un vecteur) on fait alors usage des opérateurs "." :

```
>> f = inline('x * x','x')
```

aura pour version vectorielle

```
>> f = inline('x .* x','x')
```

Une attention doit être faite lors du positionnement de l'opérateur "." . Afin d'éviter des erreurs on peut demander à Matlab de vectoriser l'expression pour nous. On utilise alors la commande **`vectorize`** *qui n'opère que sur des fonctions inline* comme ci-dessous :

Si l'on écrit :

```
>>  f = inline('x*cos(x) + x*y','x,y')
```

La commande suivante affichera une erreur :

```
>>  f([1;2],[1;2])
```

Par contre si l'on avait écrit :

```
>>  f = vectorize(inline('x*cos(x) + x*y','x,y'))
```

Matlab aurait introduit des "." là où il faut.
Il est aussi possible de vectoriser une fonction *inline* sans recourir à son expression :

```
>>  f = inline('x*cos(x) + x*y','x,y')
>>  f = vectorize(f)
```

5.3 Les graphiques 2D

Listing 14 – Fichier : graph2d.m

```
 % initialisation
clear all

% la commande plot permet d'afficher un certains nombres de points en 2D
% suivant les options, ces points sont soit représentés par un symbole, soit
% reliés par des lignes.

x = [1 2 2.5 2 3]
y = [2 1 3 4 0]
```

```
% la commande figure ouvre une nouvelle fenetre
figure
plot(x,y);
title('trait plein');
figure
plot(x,y,'r')
title('trait plein rouge');
figure
plot(x,y,'r:');
title('pointillé rouge');
figure
plot(x,y,'m-.');
title('autres options')
disp('pause')
pause;
% la commande close all ferme toutes les fenetres graphiques
close all

% on peut controler les fenêtres en sauvant la valeur renvoyée par la
% commande figure.

f1 = figure;
plot(x,y,'o');
f2 = figure;
plot(x,y,'rx');

pause;
% on ferme juste la deuxième fenêtre :
close(f2);

% si on affiche quelque chose de nouveau dans la fenêtre f1, on perd le
% contenu précédent :

plot(x,y,'r-.');

% pour garder le contenu de f1, il faut utiliser la commande hold

hold on

plot(x,y,'o');
title('Lignes et ronds,...');
% on a superposé deux graphes;
xlabel('Abscisse et le symbole \alpha');
ylabel('Ordonnée');

% on peut aussi changer les limites des axes :
axis([-1 4 -0.5 5]);

disp('maintenant, la suite,...');
pause;

close all;
clear all;

% on va afficher la fonction sinus sur [0 10]. Pour cela on va prendre
% beaucoup de points sur cet intervalle, qu'on va relien entre eux.
f1 = figure;
title('figure 1')
x = [0:0.1:10];
y = sin(x);

plot(x,y);

% on va superposer la fonction sin(x)/(x+1)
hold on;
plot(x,y./(x+ones(size(x))),'r');

% on crée une aure figure pour la tangente du sinus de x
f2 = figure;
title('figure 2');
plot(x,tan(y),'m');

% on retourne à la figure 1 pour afficher le cosinus du sinus de x
figure(f1);
```

```
plot(x,cos(y),'m-.');

disp('la suite,...')
pause;
close all;
clear all;

% on va faire plusieurs sous graphes,...
x = [-1:0.01:1];
f1 = figure;

subplot(2,2,1), plot(x,x,'r');
subplot(2,2,2), plot(x,sin(x));
subplot(2,2,2), hold on;
subplot(2,2,2), plot(x,cos(x),'r');
subplot(2,2,3), plot(x,tan(x));
subplot(2,2,4), plot(x,x.^2,'m');
subplot(2,2,4), xlabel('coucou');
subplot(2,2,4), ylabel('hello');
subplot(2,2,4), title('titre');

% on peut aussi faire des diagrammes loglog, ou semilog,...
figure;
clear x y;
x = [1:1000];
y = x.^2;

loglog(x,y);
title('un graphique loglog');

figure;
semilogx(x,y);
title('un graphique log en x');
% semilogy existe aussi.
```

5.4 Les polynomes

Listing 15 – Fichier : polynomes.m

```
% dans matlab, un polynome est représenté par le vecteur de ses
% coefficients, commençant par le degré le plus élevé.
% par exemple, soit le polynome : P(x) = 3x^2-5x+2, il sera représenté par
% le vecteur p :

p = [3 -5 2]

% pour évaluer le polynome, on utilise la fonction polyval : P(5) = ?

polyval(p,5)

% polyval accepte aussi des vecteurs de points à évaluer :

x = [-1:0.1:2];
y = polyval(p,x);

% cela permet de faire des graphes facilement :

plot(x,y);
% on aurait pu écrire directement plot([-1:0.1:2],polyval(p,[-1:0.1:2]))

hold on;
% commande pour conserver le graphe et superposer les affichages suivants

% la commande roots permet de retrouver les racines du polynome

racines = roots(p)

plot(racines,zeros(size(racines)),'ro');
% on aurait pu mettre plot(racines,polyval(p,racines),'ro');
```

```
% les racines complexes peuvent aussi être retrouvées
% x^2+x+1
p2 = [1 1 1]
racines2 = roots(p2)

% la fonction poly permet de créer un polynome à partir de ces racines :
% x^2-1 = (x+1)*(x-1)
p3 = poly([1 -1])

% la fonction polyder calcule la dérivée d'un polynome :
polyder(p2)

%la fonction conv multiplie deux polynomes :
p4 = conv(p2,p3)

% la fonction deconv divise des polynome :
deconv(p4,p2)

%la fonction polyfit permet de calculer le polynome d'interpolation passant
%    par un ensemble de points, ou bien de calculer le polynomes
%    d'approximation au sens des moindres carrés.

% on calcule le polynome de degré 3 passant exactement par 4 points définis
% par x1 et y1
x1 = [1 2 3 4]
y1 = [1 -1 2 0]
p5 = polyfit(x1,y1,3)
figure
plot(x1,y1,'ro');
hold on
plot([0:0.1:5],polyval(p5,[0:0.1:5]));
title('polynome d''interpolation - approximation');
xlabel('Abscisse')
ylabel('Ordonnée')

%on va maintenant calculer la droite (polynome de degré 1) qui approxime le
%    mieux les points (au sens des moindres carrés )

p6 = polyfit(x1,y1,1);
plot([0:0.1:5],polyval(p6,[0:0.1:5]),'m');

legend('données','polynome d''interpolation','droite d''approximation');

% pour plus d'informations : helpwin polyfun

helpwin polyfun
```

6 Résolution d'équations différentielles ordinaires.

6.1 Introduction

Les modèles mathématiques des sciences et techniques se présentent très souvent sous la forme de systèmes d'équations différentielles ordinaires, qui lient des fonctions (inconnues) du temps à leurs dérivées temporelles. En ajoutant des conditions aux limites à ces équations, on peut alors calculer ces fonctions inconnues. Par exemple, la loi de Newton permet de trouver les équation qui régissent le mouvement en chute libre d'un corps représenté par une masse ponctuelle, dans un champ de gravité constant. Si on connaît la position et la vitesse initiale du corps, on peut alors résoudre les équations du mouvement et obtenir la position et la vitesse du corps comme des fonctions du temps.

Dans de nombreux cas, ces équations sont trop complexes pour pouvoir être résolues de façon analytique et l'unique possibilité est d'essayer d'approximer les fonctions inconnues au moyen de méthodes numériques. L'idée de base consiste à ne chercher la valeur des fonctions qu'en un certain nombre de points : le problème est discrétisé.

Un certain nombre de ces méthodes numériques ont été implémentées dans le logiciel Matlab et permettent de résoudre relativement facilement un grand nombre de problèmes. Ce document constitue une introduction à l'utilisation des méthodes implémentées dans Matlab. Une utilisation efficace et avertie de ces méthodes suppose un minimum de connaissance des concepts sur lesquelles elles se basent.

6.2 Solution numérique d'EDO avec Matlab

Pour résoudre numériquement une EDO avec Matlab (ou un système d'EDOs) il faut d'abord la ramener sous la forme :

$$\frac{du}{dt} = g(t, u)$$

où u est le vecteur des fonctions inconnues. Sa dérivée temporelle est exprimée par la fonction g, qui est une fonction du temps et de u.

Il faut ensuite écrire une fonction (sous la forme d'un fichier .m) qui calcule g en fonction de t et de u. Par exemple :

```
function udot = mafonctiong(t,u)
```

on calcule les dérivées des éléments de u en fonction de u et t si u a plusieurs composantes, c'est un vecteur colonne !

```
udot = ......................;
```

On appelle une des méthodes numériques de Matlab pour résoudre l'équation. Plusieurs méthodes sont implémentées, et il vous faut choisir la fonction à appeler :

```
ode23, ode45, ode23s, ode15s, ode113,.....
```

Ces fonctions correspondent à des méthodes numériques différentes, qui sont adaptées à différents types d'équations différentielles. La syntaxe de base pour appeler ces méthodes est (par exemple pour ode45) :

```
[t u] = ode45('mafonctiong',tspan,u0);
```

La fonction ode45 résoudra l'équation différentielle décrite par **`mafonctiong`** pour l'intervalle de temps décrit par le vecteur **`tspan`**, avec u0 comme condition initiale pour u. La fonction renvoie deux résultats : un vecteur (colonne) t qui donne les temps auxquels la méthode numérique et une matrice u où chaque ligne représente l'approximation de $u(t)$ au temps spécifié à la ligne correspondante du vecteur t .

La variable tspan permet de contrôler les temps auxquels on a une approximation de u(t). Si tspan est de la forme [t_0 t_{max}] le vecteur t sera déterminé automatiquement par la méthode numérique utilisée. Si **`tspan`** est de la forme [t_0 t_1 t_2 ... t_{max}], le vecteur t sera égal à $tspan$.

6.3 Exemple

On désire résoudre numériquement l'équation différentielle décrivant le mouvement d'un pendule amorti :

$$\ddot{x} = -\frac{\eta}{m}\dot{x} - \frac{k}{m}x$$

où la position du pendule est donnée par x , m est la masse du pendule, k la constante de raideur du ressort (linéaire) et η le coefficient de frottement (linéaire).

Dans une première étape, on va transformer cette équation du second ordre en un système de deux équations du premier ordre. A cet effet, on pose donc : $u = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$

on obtient donc l'équation différentielle suivante pour u : $\begin{pmatrix} \dot{u}_1 \\ \dot{u}_2 \end{pmatrix} = \begin{pmatrix} u_2 \\ -\frac{\eta}{m}u_2 - \frac{k}{m}u_1 \end{pmatrix}$

La fonction décrivant l'équation différentielle sera donc :

Listing 16 – Fichier : oscillateur.m

```
function udot = oscillateur(t,u)
  % on initialise le vecteur colonne udot
  udot = zeros(2,1);
  %on calcule les dérivées
  udot(1) = u(2);
  udot(2) = -0.5*u(2) -1.5*u(1);
  % eta/m = 0.5 et k/m = 1.5, par exemple.
```

Comme conditions initiales, on va poser qu'en $t = 0s$, le pendule est à sa position d'équilibre $(x(0) = 0)$ mais possède une vitesse de $2m/s$. On a donc : **`u0 = [0 2]`**.

Pour résoudre l'équation différentielle entre de $t = 0s$ jusque $t = 20s$ on va lancer par exemple :

```
>> [t u] = ode45('oscillateur',[0 20],[0 2]);
```

Si on affiche la position et la vitesse du pendule en fonction du temps, on obtient :

```
>> plot(t,u(:,1));
>> plot(t,u(:,2),'r-.');
>> xlabel('temps (s)');
>> title('Pendule amorti');
>> legend('Position (m)','Vitesse (m/s)');
```

7 A Essayer

7.1 Enoncés

Exercices

Exercice 1

Écrire une fonction "monexp" qui calcule l'exponentielle d'un réel
à partir du développement de Taylor de la fonction exponentielle.

Solution : monexp.m

Comment modifier cette fonction pour que, si elle reçoit une matrice,
la fonction calcule simultanément l'exponentielle de tous les éléments
de la matrice. (La fonction renvoie alors une matrice).

Solution : monexp2.m ou monexp3.m

La fonction monexp3 est bien moins performante que monexp2. Pourquoi ?
Pour vous convaincre de la différence de performance, vous pouvez tester
ces fonctions sur de grosses matrices (ici 30*30) et mesurer les
temps de calcul :

```
>> tic; monexp2(rand(30)); toc
elapsed_time =
0.0070

>> tic; monexp3(rand(30)); toc
elapsed_time =
0.3224
```

Exercice 2

Ecrire une fonction "fibo" qui prend comme arguument un naturel "n" et qui
calcule les n premiers termes de la suite de Fibonacci.
Cette suite F(n) est définie comme: F(n) = F(n-1)+F(n-2), avec F(1)=F(2)=1.

Solution : fibo.m

Exercice 3

Ecrire une fonction "monsort" qui prend un vecteur de nombres réels comme
argument et qui renvoie, comme résultat, ce même vecteur avec
les éléments triés par ordre croissant. Comparez votre programme
(résultats et performances) avec la fonction **sort** de Matlab.

Exercices

7.2 Scripts pour les exercices

Listing 17 – Fichier : monexp.m

```
function result = monexp(x)

 % le premier terme de la série est 1
 % exp(x) = 1 + x + (x^2 / 2!) + (x^3 / 3!) + ....
 somme = 1;

 % le terme en x^1 est x
 i=1;
 terme =x;

 % on va additionner les termes successifs de la serie jusqu'a ce que
 % le fait d'additionner un terme ne change pas le resultat. (le terme est
 % trop petit par rapport a la somme des termes deja calcules.
 while(somme ~= somme + terme)

   % on met la somme des termes a jour.
   somme = somme+terme;

   % on calcule le terme en x^i a partir du terme en x^(i-1)
   i = i+1;
   terme = terme*x/i;

 end

 % on renvoie le resultat
 result = somme;
```

Listing 18 – Fichier : monexp2.m

```
function result = monexp2(x)
% cette fonction accepte un argument x du type matrice
% elle renvoie la matrice des exponentielles des entrées de la matrice
% fournie
% essayez monexp2([0 4 1]);

% le premier terme de la série est 1 -->  une matrice de 1
% exp(x) = 1 + x + (x^2 / 2!) + (x^3 / 3!) + ....
% Première modification, on doit initialiser la somme comme une matrice de 1
somme = ones(size(x));

% le terme en x^1 est x
i=1;
terme =x;

% on va additionner les termes successifs de la série jusqu'à ce que
% le fait d'additionner un terme ne change pas le résultat. (le terme est
% trop petit par rapport à la somme des termes déjà calculés.
```

```
while(somme ~= somme + terme)

  % on met la somme des termes à jour.
  somme = somme+terme;


  % Deuxième modification, lors du calcul du terme suivant, on doit calculer
  % simultanément tous les termes pour tous les éléments de la matrice.
  % il faut effectuer la multiplication "terme à terme" et utiliser
  % l'opérateur ".*" au lieu de "*"
  % [a^i b^i] .* [a b] --> [a^(i+1) b^(i+1)]
  %on calcule le terme en x^i à partir du terme en x^(i-1)
  i = i+1;
  terme = terme.*x/i;
  %           ".*"  ici !

end

% on renvoie le résultat
result = somme;
```

Listing 19 – Fichier : monexp3.m

```
 function result = monexp3(x)
% cette fonction accepte un argument x du type matrice
% elle renvoie la matrice des exponentielles des entrées de la matrice
% fournie
% essayez monexp2([0 4 1]);

% modification : On effectue une boucle sur tous les éléments de la matrice.


result = zeros(size(x));

% boucle sur les lignes de x
for(k=1:size(x,1))

  % boucle sur les colonnes de x
  for(l=1:size(x,2))

    % le premier terme de la série est 1
    somme = 1;

    % le terme en x(k,l)^1 est x(k,l)
    i=1;
    terme =x(k,l);

    % on va additionner les termes successifs de la série jusqu'à ce que
    % le fait d'additionner un terme ne change pas le résultat. (le terme est
    % trop petit par rapport à la somme des termes déjà calculés.
    while(somme ~= somme + terme)

      % on met la somme des termes à jour.
      somme = somme+terme;

      % on calcule le terme en x^i à partir du terme en x^(i-1)
      i = i+1;
      terme = terme*x(k,l)/i;

    end

    % on renvoie le résultat
    result(k,l) = somme;

  end % boucle sur k
end % boucle sur l
```

Listing 20 – Fichier : fibo.m

```
function suite = fibo(n)
  % programme simple

  % on verifie les donnees
  if(n<=0)
    error('n ne peut pas etre negatif');
  end

  %on alloue la memoire
  suite = zeros(1,n);

  % on calcule la suite
  if(n==1)
    suite(1) = 1;
  else

    suite(1:2) = [1 1];

    % si n==2, la boucle n'est pas effectuee !
    for(i=3:n)
    suite(i) = suite(i-1)+suite(i-2);
  end

end
```