Master Optimisation/AMS/MSV (M2)
Université Paris–Sud
AN-EDP

Stage de rentrée
Centre d'Orsay
Année 2017 – 2018

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

Thème - 1 *Basic notions on Matlab*

Contents

1 Matlab Environment

`Matlab` is a practical development tool for students in Master 2. It offers several libraries (as toolbox) and an easy to handle integrated development environment.

In the Matlab environment, testing new algorithms is greatly simplified since one is spared with details related to linear algebra (matrix manipulation, inversion of linear system, computation of eigenvalues) as well as those related to solving nonlinear and ordinary differential equations.

It is also possible to do advanced developments in Matlab (e.g. validation of prototypes), thanks to its profiling and debugging functionalities as well as its good plotting library.

Unfortunately, Matlab offers a dynamic typing and interpreted language. This can lead to less efficient codes. However possibility for interfacing with compiled and statically typed languages like FORTRAN, C, C++,. . . are offered.

Although the syntax offered by Matlab is simple and supports imperative programming, it is advised to program in Matlab with a vectorial execution in mind.

The present document is just a brief and basic description of Matlab, and the reader is encouraged to seek for other functionalities that Matlab can provide.

2 Matlab as scriptural language

2.1 Access the help

In order to seek for help on a specific Matlab function, just enter the command:
```
≫ help function_name
```
Example:

```
>> help det
```
Try also:
```
>> lookfor det
>> helpwin
```

2.2 Create and modify matrices. Access to elements and memory management.

2.2.1 Create a matrix

You can create a matrix A explicitly:
```
>> A = [1 2 3 ; 4 5 6]
```
The matrix is delimited by brackets. The elements of the matrix are entered rows by rows. A semicolon is used to separate the rows of the matrix. A vector can be defined in the same way.
```
>> B = [1 5]
```
To access an element of the matrix we specify in parenthesis, separated by a comma, the row and the column of the desired element:
```
>>  A(2,3)
```
For scalars, there is no need to specify neither the row nor the column:
```
>>  F(1,1)
>>  F(1)
>>  F
```

Remark 2.2.1.

- It is also possible to generate special matrices:
  see the Matlab commands **`eye,ones,zeros,diag,tril,triu,sparse, ....`**
- In order to visualize the sparsity pattern of a matrix use the command **`spy`**.

2.2.2 Modifing a matrix

It is possible to modify an element of a matrix:

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

If we modify a non existent element of a matrix, the matrix is enlarged until the element can exist:

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

2.2.3 Memory management

Since it is easy to create new variables in the Matlab environnement, if no care is exercised, the workspace can quickly get crowded with unused variables. The command **`who`** gives the list of variables in the workspace. The command **`whos`** gives the same list but with additional informations on the nature and the size of the variable.

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

To clear a variable and free the memory used by it, enter the command:

```
clear name_of_the_variable
>> clear F
>> who
```

The command **`clear all`** delete all variables in the workspace. It is not possible to undo the clear command.

2.2.4 Display

To prevent the result of a command from being displayed, just put a semicolon a the end of the command:

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

No result is displayed. Yet the command has been executed.

```
>>  T
```

2.3 Elementary operations on matrices.

For this subsection, and for illustration, we are still going to use the matrices defined in previous sections.

2.3.1 Operations on scalars

Matlab allows to perform classical operations on scalars:

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

Classical functions such as sin, cos, log, ... are also available. (try helpwin ops, ou helpwin elfun)

2.3.2 Operations on matrices

Operations $+, -, *, /$ are also available for matrices. If no ambiguity exists for addition and subtraction, care must be exercise for operations such as multiplication, division, power,.... Examples :

```
>>  a*A
```

The matrices must have compatible dimensions.

2.3.3 Solving linear systems

One way of solving the linear system $A * x = C$, is to compute the inverse of A :

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

But this approach may be inaccurate if the matrix A is badly conditioned. Moreover, computing the inverse of a matrix requires much more operations than it takes to solve the linear system. The best way to proceed th Matlab is to use the backslash operator ($\setminus$). Then Matlab looks for the must appropriate direct method to solve the system (LU decomposition of the matrix followed by solving two triangular linear systems)

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

Let us mention that this approach (with backslash) make use of direct methods for linear systems (full or sparse). In order to use iterative methods of Krylov type (known to be less memory consuming), just select the appropriate one in the available list: **`pcg, cgs, bicg, bicgstab, gmres, ...`**. Use the help to access the function description:

```
>> help pcg
```

2.3.4 Concatenation and extraction

It is possible to create new matrices by concatenating existent matrices. The syntax is similar to that use to create a matricx from scalars.

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

To extract the sub-matrix made of the second row of A and columns 3, 2 and 1. Enter command:

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

2.4 Colon operator ":"

The ":" (colon) operator is an operator that can help generate a sequence of numbers. It is also very useful for extracting sub-matrices, managing iteration loops (see section control structures), ...

2.4.1 Generate a sequence of numbers

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

2.4.2 Extracting sub-matrix

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

Read : A(all rows, second column). This notation is equivalent to:

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

2.4.3 Drawing a function

A simple way to visualize the graph of a function is to evaluate that function at a number of points and to connect those points by straight lines. Let us visualize the function $\sin(x)$ for x going from 0 à 12.

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

At the first line, values from 0 to 12 spaced by 0.05 are generated. Then, the function sinus is evaluated at those points. Finally those points are displayed connected by lines.

2.4.4 Learn more "with the help"

If you wish to access the help about the operator ":", enter

```
>> help colon
```

Also try : `help linspace , help logspace`

3 Matlab as imperative language

The Matlab language would not have much interest if it were not possible to perform tests, implementing loops, etc., as in the majority of programming languages. The help concerning the control structures can be accessed with:

```
≫ helpwin lang
```

3.1 Boolean operations

The boolean values ”true” and ”false” are respectively identified in Matlab by 0 and 1. The boolean operators in Matlab are described in the help topic **relop** (helpwin relop). Available boolean operators are :

Listing 1: Boolean Operators

```
> , >= , < , <= , == (équal), ~= (different), & (and), | (or), ~ (not), xor (exclusif or).
```

Example :

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

Warning : When writing expressions which combines arithmetic and boolean operations, make sure to properly use the parentheses.

3.2 Control structures

Instruction IF

The instruction IF is similar to that of many other programming languages `(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
```

If the expression *expression1* is true, then the commands *statements1* are executed, if not, if the expression *expression2* is true then the commands *statements2* are executed, finally, if neither *expression1* nor *expression2* are true, then it is the commands *statements3* which are executed. The ELSEIF and ELSE clauses are optional.

Warning : A frequent mistake is to write a THEN after *expression1*. Recall that instruction THEN does not exist in Matlab.

Instruction WHILE

Listing 5: Instruction WHILE

```
while expression
    statements
end
```

The structure WHILE is a powerful structure that can help repeat a sequence of instructions provided a specific condition is satisfied :

Unless *expression* is false, the command *statements* will be executed. As illustration, let us compute the sum of square of integers from 1 to 11.

Listing 6: Instruction WHILE applied

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

And without a loop :

$\gg$ `sum([1:11].^2)`

Be careful not to create an infinite loop. e.g.

Listing 7: Instruction WHILE applied

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

This loop will never stop. It is always possible to stop a running Matlab command by using the following keyboard : $[CTRL] + C$ (combination of keys Ctrl et C).

Instruction FOR

Listing 8: Instruction FOR

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

If n is the length of the vector representing *expr*, the commands in *statement* will be executed n times. At the first execution *variable* will have value expr(1), ... and so on until the last execution where its value will be expr(n). As an illustration, let us compute the product of odd integers from 1 to 11.

Listing 9: Instruction FOR applied

```
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
```

And without the loop :
≫ `prod([1:2:11])`

4 Matlab as structural language

4.1 Scripts and functions

Matlab would have a very limited utility if it were not possible to write scripts and functions:

A **`Script`** in Matlab is a set of instructions, written in a file, which can be executed all at once.

A **`Function`** in Matlab is a set of instructions, taking some input arguments and returning one or more results.

SCRIPTS ou PROCEDURES

Scripts int Matlab are stored in files with extension ".m". They contain a suite of Matlab instructions written as if they were entered in Matlab command window. To run a script, just enter in Matlab command window, the name of the script's file without the ".m" extension.

`Example : monscript.m`

Listing 10: File: 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
```

To execute this script, just enter the following command:
≫ `monscript`

A Script in Matlab can access variables that were defined just before the script is launched. And variables that are defined inside the script will remain accessible to the user after the script is executed.

For further informations enter the command : ≫ `helpwin script`

FUNCTIONS

Functions in Matlab are also stored in ".m" files (mfiles). *The file must have the same name as the function*. The syntax for defining a function is as follows:

Listing 11: Function definition

```
 function [result_1, ... , result_n] = fonction_name (argument_1,..., argument_p)
  % contenu de la function ici
```

Instructions inside the function can make use of variables $argument_1, \ldots, argument_p$, and must assign a value to variables $result_1, \ldots, result_n$.

`Example : 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;
end
```

At execution we get:
`>>  [r1, r2, r3] = mafonction(5,9);`
The command :
`>>  help mafonction`
Will display the help. It will consist of the first lines of comments at the header of the function definition.
`[somme, produit, minimum] = mafonction(a,b)`
Will return the sum, the product and the minimum of a and b. For further informations : **`helpwin function`**

Matlab also offers other ways to define functions:

- The first approach uses the command **`inline`** :
  `>>  f = inline('x*x','x')`
  For further informations enter
  `>> help inline`

- The second approach uses the handle to the just defined function :
  `>>  f = @(x) (x*x)`
  This approach is more advantageous since it offers the possibility to bind some arguments which generates new functions:
  `>>  f = @(t,x,y) (t * x + cost(t) * y)`
  `>>  y = 1; g = @(t,x) f(t,x,y);`
  This will define the function `g(t,x) = t * x + cost(t)`.

4.2 Data Structures

C like struct

Matlab like most other procedural languages such as the C programming language offers the possibility to store in a single variable, several other variables of not necessary the same type. For this purpose, Matlab offers the command **`struct`**. Type **`help struct`** for further informations.

`>>  p = struct('firstname','Jules', 'lastname','Mandel', 'age', 30 )`
To access any field use the "." operator like in C programming language.
`>>  p.firstname`
`>>  fprintf(1,'Mr %s %s is %d years old \n', p.firstname, p.lastname, p.age);`
It is possible to change the value of a field
`>> p.age = 28 ;`

It is always possible to extend any struct dynamically. Let us add a display function in our struct :

```
≫ p.disp = @(a) fprintf(1,'Mr %s %s is %d years old \n', a.firstname, a.lastname,
a.age);
≫ p.disp(p);
```

To list all the fields of our struct object, enter the following command

```
≫ fieldnames(p)
```

Note that on contrary to language such as the C language, we are not defining a new type in the language: we are just working on an instance of that implicit type. So in order to define a new variable of the same type as p, we need to perform a deep copy and change the value of all fields.

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

It is better to furnish an initialization function.

Moving deeper in this direction will bring us directly in the domain of objects oriented programming, which is beyond the scope of the present class. However let us mention that recent versions of Matlab are equipped with tools that allow for more advanced object oriented programming.

Collection of objects of heterogeneous types

Storing collections of variables of different types is also something common in advanced programming. This is something usually difficult to achieve in most statically typed programming language such as C, C++ etc. But not in Matlab which offers for this purpose a tool call **`cell`**. Type **`help cell`** for further information.

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

4.3 Library management or physical components management

Path acces problems

Matlab would execute a script or function only if it knows in which directories it should look for the corresponding mfiles. By default, Matlab looks in its directories as well as in the current directory. To check which is the current directory, use the command **`pwd`** (print working directory).

```
pwd
```

In order to change the directory, use the command **`cd`**:

```
≫ cd mfiles (to go down the subdirectory mfiles)
```

If, without changing the directory, you wish to add a directory to the list of directories where Matlab should look for scripts and functions, use the command **`addpath`**:

≫ `addpath('./utils')` (to add the subdirectory utils)

5 Miscellaneous

For each topic, you will find either a commented Matlab script containing several examples, or a page with some explanations and examples. Download these scripts, try them, modify them, ...

5.1 Help in Matlab

(essential if one wishes to save time).

5.2 Vectorization

5.2.1 Illustration

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)]
```

5.2.2 Enable tool for automatic vectorizing *inline* functions

When writing functions that will serve as arguments to Matlab functions (such as ode45,...), it is advised to write them in vectorial forms (that is they should return a vector if the input is a vector). This is achieved by using the operator ".":
The vectorial form of

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

should be

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

Care must be exercise when placing the operator "." . In order to avoid unexpected mistakes, we can ask Matlab to vectorize the expression for us. To this end, we use the command **`vectorize`** *This command operates only on inline functions* as illustrated below:

if write :

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

the following command will issue and error:

```
≫  f([1;2],[1;2])
```
On contrary if one had written:
```
≫  f = vectorize(inline('x*cos(x) + x*y','x,y'))
```
Matlab would have introduced some "." where needed.
It is also possible to vectorize an *inline* function without knowledge of its expression:
```
≫  f = inline('x*cos(x) + x*y','x,y')
≫  f = vectorize(f)
```

5.3 Graphics in 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 Polynomials

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
```

5.5 Try this

5.5.1 Subjects

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

5.5.2 Scripts for exercises

Listing 16: Fichier: monexp.m

```
function result = monexp(x)

  % le premier terme de la serie 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;
end
```

Listing 17: **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;
end
```

Listing 18: **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
end
```

Listing 19: 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
end
```