Débuter avec MATLAB

Stéphane Balac

Centre de Mathématiques
INSA de Lyon

Table des matières

Introduction

Ce document est un guide d'initiation à MATLAB. MATLAB est un programme interactif de calcul scientifique utilisable pour la résolution numérique de nombreux problèmes mathématiques ou appliqués. En outre, MATLAB dispose de potentialités graphiques importantes.

L'objectif de ce document est de permettre au débutant de rapidement se familiariser avec MATLAB. Aussi, seules les fonctionnalités les plus courantes de MATLAB sont présentées (aucune référence aux *Toolbox* n'est faite par exemple). De même, en général, les commandes MATLAB ne sont pas présentées de manière exhaustive. Seule l'utilisation la plus courante de la commande est mentionnée. **Cette documentation a été rédigée en prenant comme support la version 5.1 de MATLAB**. Toutefois dans la mesure où nous nous limitons aux fonctionnalités les plus courantes de MATLAB, la plupart des commandes décrites sont communes aux versions antérieures et resteront compatibles avec les versions ultérieures. Les commandes sont généralement présentées suivies d'un exemple. Le lecteur est invité à tester par lui même les exemples proposés (et des variantes). Au sein d'un même paragraphe, les différents exemples sont souvent liés. Il est donc recommandé d'exécuter les exemples dans l'ordre où ils apparaissent. Par contre d'un paragraphe à l'autre, il peut être nécessaire de « nettoyer » l'espace de travail en tapant `clear all` dans la fenêtre de contrôle MATLAB. La présentation qui est faite correspond à une utilisation de MATLAB sur station de travail Unix, mais devrait être très facilement transposable aux autres systèmes.

MATLAB est abréviation de MATrix LABoratory. MATLAB est avant tout un programme de calcul matriciel. L'exemple suivant est révelateur à ce sujet : si `A` et `B` sont deux variables définies comme étant des matrices, l'instruction `A*B` calcule le produit matriciel des matrices `A` et `B`. D'autre part avec MATLAB, un scalaire est considéré comme une matrice à un élément et un vecteur est considéré comme une matrice à une ligne (ou une colonne). Bien qu'un peu déroutant au départ, cette approche s'avère en pratique d'une grande efficacité.

Nous supposons que le lecteur est familié des notions d'algèbre linéaire élémentaire, notamment matriciel. Un exemple de matrice que nous utiliserons fréquemment dans ce document correspond au carré magique. L'expression carré magique désigne tout tableau carré dont les cases renferment des nombres entiers disposés de telle sorte que les sommes des nombres inscrits sur chaque ligne, sur chaque colonne et chaque diagonale soient toutes égales. On pourra consulter au sujet des carrés magiques le chapitre 2 de l'ouvrage *Histoire d'algorithmes* [1] : « plus souvent utilisés comme talismans que comme objet de recherche mathématique, les carrés magiques étaient étroitement liés à la vie quotidienne. Ainsi, il était fait appel à eux pour faciliter les accouchements, soigner les morsures d'animaux vénéneux ou encore guérir de maladie. Il est donc naturel qu'ils ne se rencontrent pas que dans les livres mais aussi sur des tableaux, sous forme de gravure sur des monuments, des médailles ...». Comme exemple bien connu, on peut citer le carré magique d'ordre 4 qui apparaît sur une gravure sur bois de la Renaissance d'Albrecht Dürer (Mélancolie, 1415).

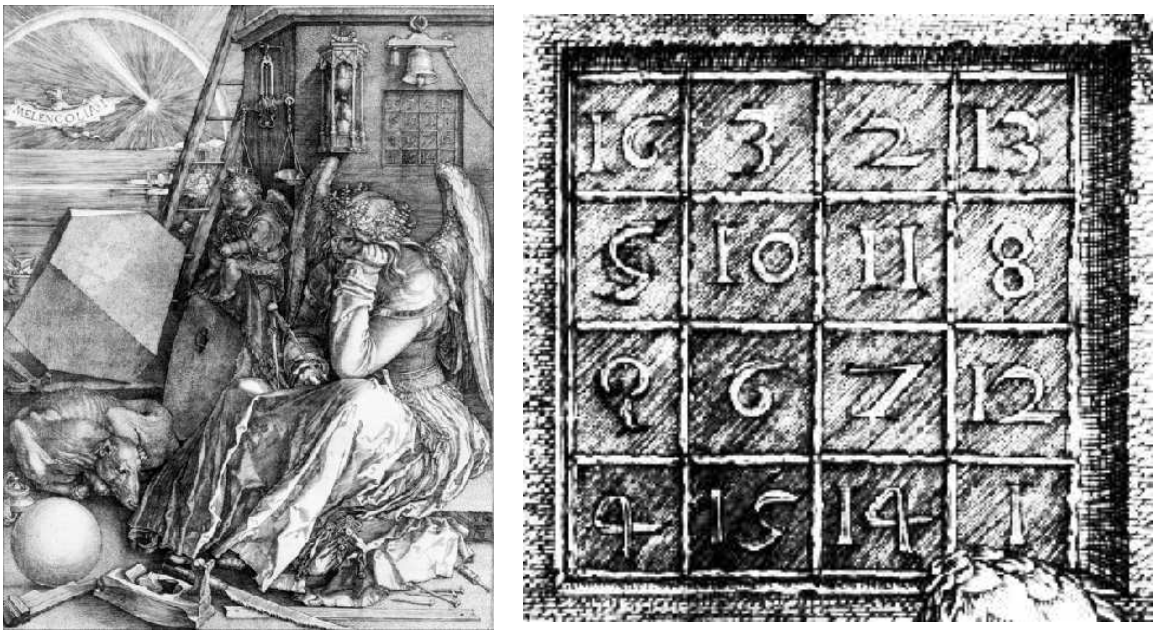

FIG. 1 – Mélancolie par Albrecht Dürer (1415). Un carré magique d'ordre 4 apparaît au dessous de la cloche.

Par convention, les commandes MATLAB apparaissent dans le texte dans une fonte « `true-type` ». Les exemples sont eux aussi présentés dans une fonte « `true-type` ». Dans un souci de gain de place certains exemples sont présentés sur deux colonnes.

Ce document est disponible aux formats pdf et postscript auprès de l'auteur. Une version hypertexte (HTLM) de ce document est également disponible.

Ce document est destiné à évoluer, aussi toute erreur ou ambigüité relevée, toute remarque ou avis peut être communiqué à l'adresse suivante : `Stephane.Balac@voila.fr`.

Brest, août 1999

Cette version du document a bénéficié de corrections suggérées par Olivier Mazet du Centre de Mathématiques de l'INSA de Lyon.

Lyon, janvier 2001

Stéphane Balac
Centre de Mathématiques
INSA de Lyon
F-69621 Villeurbanne Cedex

1 Présentation et généralités

1.1 Une session MATLAB

Pour lancer MATLAB[1], commencez par ouvrir une fenêtre de commande (commandtool) puis déplacez-vous dans votre répertoire (directory) de travail. Tapez la commande `matlab`. Voici ce que vous devez voir apparaître :

```
[unix prompt] matlab

                    < M A T L A B (R) >
          (c) Copyright 1984-97 The MathWorks, Inc.
                    All Rights Reserved
                     Version 5.1.0.421
                        May 25 1997


Commands to get started: intro, demo, help help
Commands for more information: help, whatsnew, info, subscribe

>>
```

Notez le prompt MATLAB (»») qui indique que MATLAB attend des instructions. Voici un exemple de session MATLAB :

```
>> A = [ 1 3; 4 2 ]
A =
     1     3
     4     2
>> A*A
ans =
    13     9
    12    16
```

```
>> quit

 16 flops.

[unix prompt]
```

Dans cette session on a défini la matrice

$$A = \begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}$$

et l'on a calculé son carré.

Chaque ligne d'instructions doit se terminer par un retour chariot. La commande pour quitter MATLAB est `quit`. On veillera dans la mesure du possible à ne pas lancer simultanément plusieurs sessions MATLAB.

1.2 L'espace de travail

Comme tout langage de programmation MATLAB permet de définir des données variables. Les variables sont définies au fur et à mesure que l'on donne leurs noms (identificateur) et leurs valeurs numériques ou leurs expressions mathématiques. MATLAB ne

[1] La présentation qui est faite correspond à une utilisation de MATLAB sur station de travail Unix, mais devrait être très facilement transposable aux autres systèmes.

nécessite pas de déclaration de type ou de dimension pour une variable. Les variables sont stockées dans l'espace de travail (ou workspace) et peuvent être utilisées dans les calculs subséquents.

Pour obtenir la liste des variables actives de l'espace de travail on dispose des commandes `who` et `whos`. La commande `who` affiche le nom des variables actives. La commande `whos` donne plus d'informations : le nom, la taille du tableau (nombre de lignes et de colonnes) associé, l'espace mémoire utilisé (en Bytes) et la classe des données (principalement `double array` s'il s'agit d'un tableau de valeurs réelles ou complexes et `char` s'il s'agit d'un tableau de caractères).

La commande `clear` permet de nettoyer l'espace de travail : toutes les variables sont détruites. Il est possible de ne détruire qu'une partie des variables en tapant `clear nom-var` où `nom-var` est le nom de la (ou des) variable(s) à détruire.

```
>> x=2*pi/3; y=sin(x); z=cos(x);
>> A = [ 1 3; 4 2 ]; B = A*A;
>> t = 'bonjour';
>> who

Your variables are:

A         B         t         x         y         z

>> whos
  Name      Size         Bytes  Class

  A         2x2             32  double array
  B         2x2             32  double array
  t         1x7             14  char array
  x         1x1              8  double array
  y         1x1              8  double array
  z         1x1              8  double array

Grand total is 18 elements using 102 bytes

>> clear x y t
>> whos
  Name      Size         Bytes  Class

  A         2x2             32  double array
  B         2x2             32  double array
  z         1x1              8  double array

Grand total is 9 elements using 72 bytes

>> clear
>> who
>>
```

Il est possible de sauvegarder une session MATLAB MATLAB dans un fichier pour une utilisation ultérieure. L'instruction `save nom-fic` enregistre toutes les variables de l'espace de travail dans le fichier `nom-fic.mat`. Si aucun nom de fichier n'est précisé, le fichier par défaut est `matlab.mat`. Il est possible de ne sauver qu'une partie des variables (par exemple seulement la variable contenant le résultat d'un calcul) en utilisant l'instruction `save nom-fic nom-var` où `nom-var` est le nom de la (ou des) variable(s) à sauvegarder. Attention, seul le contenu des variables est sauvegardé et non pas l'ensemble des instructions effectuées durant la session. Pour ramener dans l'espace de travail les variables sauvegardées dans le fichier `nom-fic.mat`, taper `load nom-fic`.

Dans l'exemple qui suit on calcule le sinus et le cosinus de $2\pi/3$ dans les variables `y` et `z`. On sauve ces résultats dans un fichier `toto.mat`. Après avoir quitté MATLAB on peut vérifier que le fichier `toto.mat` existe bien mais qu'il ne s'agit pas d'un fichier ascii. Si l'on relance MATLAB et que l'on tape `load toto`, on peut vérifier que les variables `y` et `z` existent bien et ont les valeurs précédentes.

```
>> x=2*pi/3, y=sin(x), z=cos(x)
x =
    2.0944
y =
    0.8660
z =
   -0.5000
>> save toto y z
>> quit

 6 flops.

[unix prompt] ls
toto.mat
[unix prompt] more y.mat
MATLAB 5.0 MAT-file, Platform: SOL2, Created on: Mon Jul  5 20:42:21 1999
     ‘ MI      8   e     e       d     ‘   ‘ ‘ ‘z          ¿ßÿÿÿÿÿü

[unix prompt] matlab

>> load toto
>> who

Your variables are:

 y         z

>> y
y =
    0.8660
>> z
z =
```

```
   -0.5000
>> x
??? Undefined function or variable 'x'.

>>
```

La commande `diary` permet de sauvegarder l'ensemble d'une session dans un fichier ascii pour un traitement ultérieur (insertion dans un document, impression, ...). Le fichier de sauvegarde par défaut a pour nom `diary`. On provoque la sauvegarde dans le fichier `nom-fic`, par l'instruction `diary nom-fic`. Attention, c'est la commande `diary` qui déclenche le début de sauvegarde ; il est impossible de sauvegarder la partie de la session précédent son appel. Pour arrêter la sauvegarde, taper `diary off`.

```
>>  x=2*pi/3; y=sin(x);
>> diary toto
>>  z=cos(x)
z =
   -0.5000
```

```
>> diary off
>> t = tan(x)
t =
   -1.7321
>>
```

Si vous éditez le fichier `toto`, voici ce que vous verrez :

```
>>  z=cos(x)
z =
   -0.5000
>> diary off
```

et rien de plus ...

1.3 Obtenir de l'aide

Dans une session MATLAB, il est possible d'obtenir une aide en ligne sur une commande en tapant `help nom-commande`. Par exemple,

```
>> help diary

 DIARY Save text of MATLAB session.
    DIARY file_name  causes a copy of all subsequent terminal input
    and most of the resulting output to be written on the named
    file. DIARY OFF suspends it. DIARY ON turns it back on.
    DIARY, by itself, toggles the diary state.

    Use the functional form of DIARY, such as DIARY('file'),
    when the file name is stored in a string.

>>
```

Attention, les commandes MATLAB doivent être tapées en *minuscules* pour être reconnues, même si elles figurent en majuscules dans l'aide en ligne.

On peut également obtenir de l'aide par le biais de la commande `doc` qui donne accès à une documentation complète au format HTML. Pour quitter cette documentation, cliquer sur Exit Program dans le menu File du navigateur.

La commande `lookfor` permet de rechercher un mot-clé parmi les lignes de commentaires en entête des fonctions MATLAB (ces lignes sont celles affichées par la commnande `doc`). L'instruction `lookfor motclé` recherche le mot-clé `motclé` dans la première ligne de commentaire de toutes les fonctions MATLAB. L'instruction `lookfor motclé -all` recherche le mot-clé dans toutes les lignes de commentaires en entête des fonctions. Attention le mot-clé doit être en anglais, les commentaires des fonctions MATLAB étant rédigés en anglais. On peut utiliser une phrase comme mot-clé. Il faut alors l'écrire entre guillemets (`' '`). La commande `lookfor` retourne le nom la fonction MATLAB (ou des fonctions) où le mot-clé figure dans la première ligne de commentaires. Elle retourne également la ligne de commentaires où figure le mot-clé. Si le mot-clé n'a été trouvé dans aucune ligne de commentaires, MATLAB rend la main sans rien afficher.

```
>> lookfor determinant
DET    Determinant.
>> lookfor 'tridiagonal matrix'
LESP   Tridiagonal matrix with real, sensitive eigenvalues.
POISSON Block tridiagonal matrix from Poisson's equation.
TRIDIAG Tridiagonal matrix (sparse).
TRIDIEIG  Find a few eigenvalues of a tridiagonal matrix.
TRIDISOLVE  Solve A*x = b where A is a square, symmetric tridiagonal matrix.
>> lookfor papillon
>>
```

1.4 Syntaxe d'une ligne d'instructions

Si une instruction MATLAB est suivie d'un point virgule, le résultat de cette instruction n'est pas affiché. Pour ré-afficher un résultat contenu dans une variable, il suffit de taper le nom de la variable. Le résultat de la dernière instruction exécutée peut être rappelé par la commande `ans` :

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

```
>> ans
ans =
    91    67    67
    67    91    67
    67    67    91
>>
```

Plusieurs instructions MATLAB peuvent figurer sur une même ligne. Il faut alors les séparer par une virgule ou par un point virgule. D'autre part, si une commande est trop longue pour tenir sur une ligne, il est possible de poursuivre sur la ligne suivante en terminant la ligne par 3 points (...).

```
>> B = [ 1 3; 4 2 ]; B*B
ans =
```

```
    13     9
    12    16
>> x = 1 + 2 + 3 + 4 + 5 + 6 ...
+7 + 8 + 9 + 10
x =
    55
>>
```

Si la syntaxe de l'instruction soumise est erronée ou si vous demandez à MATLAB d'exécuter une instruction illégale (qui n'a pas de sens mathématique par exemple), vous obtiendez un message d'erreur. Ce message vous indique les sources d'erreurs possibles et doit vous permettre de corriger rapidement votre erreur.

```
>> A + B
??? Error using ==> +
Matrix dimensions must agree.
>> C = [ 1 2 3; 4 5]
??? Number of elements in each row must be the same.
>> whose
??? Undefined function or variable 'whose'.
>>
```

1.5 Gestion des fichiers du répertoire de travail

Un certain nombre de commandes permettent de gérer les fichiers du répertoire de travail. La commande `dir` donne la liste des fichiers du répertoire de travail. La commande `cd` permet de changer de répertoire de travail. La commande `type` permet d'afficher le contenu d'un fichier. La commande `delete` permet de détruire un fichier. Ces commandes s'utilisent de la même manière que les commandes correspondantes d'Unix (ou du DOS). Enfin la commande `edit` permet d'ouvrir un éditeur de texte. Le choix de l'éditeur a été effectué au moment de l'installation de MATLAB sur votre machine, il vous est donc impossible de choisir votre éditeur favori. Il est également possible d'exécuter des commandes Unix à partir de MATLAB en faisant précéder la commande d'un point d'exclamation (`!`).

Si vous avez effectué les exemples du paragraphe 1.2, deux fichiers toto et toto.mat existent dans votre répertoire de travail. Voici alors quelques exemples d'utilisation des commandes de gestion des fichiers.

```
>> dir
 .       ..      toto    toto.mat
>> edit toto
>> delete toto
>> !ls
toto.mat
>>
```

2 Types de données et variables

2.1 Particularités de MATLAB

Comme tout langage de programmation MATLAB permet de définir des données variables. Une variable est désignée par un identificateur qui est formé d'une combinaison de lettres et de chiffres. Le premier caractère de l'identificateur doit nécessairement être une lettre. Attention, MATLAB différencie majuscules et minuscules ! Ainsi `X33` et `x33` désignent deux variables distinctes. Les variables sont définies au fur et à mesure que l'on donne leurs noms (identificateur) et leurs valeurs numériques ou leurs expressions mathématiques. L'utilisation de variables avec MATLAB ne nécessite pas de déclaration de type ou de dimension. Le type et la dimension d'une variable sont déterminés de manière automatique à partir de l'expression mathématique ou de la valeur affectée à la variable. Une variable peut être de type réel, complexe, chaîne de caractères ou logique.

Pour MATLAB toute variable est considérée comme étant un tableau d'éléments d'un type donné. MATLAB différencie trois formes particulières de tableaux. Les *scalaires* qui sont des tableaux à une ligne et une colonne. Les *vecteurs* qui sont des tableaux à une ligne ou à une colonne. Les *matrices* qui sont des tableaux ayant plusieurs lignes et colonnes. Une variable MATLAB est donc toujours un tableau que l'on appelle variable scalaire, vecteur ou matrice suivant la forme du tableau.

2.2 Les types de données

2.2.1 Les 4 types de données MATLAB

Les trois principaux types de variables utilisés par MATLAB sont les types réel, complexe et chaîne de caractères. Il n'y a pas de type entier à proprement parler. Le type logique est associé au résultat de certaines fonctions. Signalons qu'il est inutile (impossible) de déclarer le type d'une variable. Ce type est établi automatiquement à partir des valeurs affectées à la variable. Par exemple les instructions `x = 2 ; z = 2+i ; rep = 'oui' ;` définissent une variable `x` de type réel, une variable `z` de type complexe et une variable `rep` de type chaîne de caractères.

```
>> clear
>> x = 2; z = 2+i; rep = 'oui';
>> whos
  Name      Size         Bytes  Class

  rep       1x3              6  char array
  x         1x1              8  double array
  z         1x1             16  double array (complex)

Grand total is 5 elements using 30 bytes
>>
```

Comme on ne définit pas de manière explicite le type d'une variable, il est parfois utile de pouvoir le déterminer. Cela est possible grâce aux commandes `ischar`, `islogical` et `isreal`. `ischar(x)` retourne 1 si `x` est de type chaîne de caractères et 0 sinon. `islogical(x)` retourne 1 si `x` est de type logique et 0 sinon. La commande `isreal(x)` est à utiliser avec discernement : elle retourne 1 si `x` est réel ou de type chaîne de caractères et 0 sinon (`x`

est complexe à partie imaginaire non nulle ou n'est pas un tableau de valeurs réelles ou de caractères).

```
>> ischar(rep)
ans =
     1
>> ischar(x)
ans =
     0
>> isreal(z)
ans =
     0
>> isreal(x)
ans =
     1
>> isreal(rep)
ans =
     1
>>
```

2.2.2 Le type complexe

L'unité imaginaire est désignée par `i` ou `j`. Les nombres complexes peuvent être écrits sous forme cartésienne $a + ib$ ou sous forme polaire re^{it}. Les différentes écritures possibles sont `a+ib`, `a+i*b`, `a+b*i`, `a+bi` et `r*exp(it)` ou `r*exp(i*t)` avec `a`, `b`, `r` et `t` des variables de type réel. Les commandes `imag`, `real`, `abs`, `angle` permettent de passer aisément de la forme polaire à la forme cartésienne et réciproquement. Si `z` est de type complexe, les instructions `imag(z)` et `real(z)` retournent la partie imaginaire et la partie réelle de `z`. Les instructions `abs(z)` et `angle(z)` retournent le module et l'argument de `z`.

On fera attention au fait que les identificateurs `i` et `j` ne sont pas réservés. Aussi il est possible que des variables de noms `i` et `j` aient été redéfinies au cours d'un calcul antérieur et soient toujours actives. Si c'est la cas, on peut soit détruire ces deux variables (`clear i j`), `i` et `j` redeviennent alors l'unité imaginaire, soit ré-affecter à `i` ou à `j` la valeur unité imaginaire par l'instruction `i=sqrt(-1)`. On se méfiera donc des boucles d'indices `i` et `j` dans lesquelles on manipule des variables de type complexe. On fera également attention à ne pas laisser d'espace autour de l'unité imaginaire afin d'éviter de mauvaises interprétations des données dans certains cas. Comparez par exemple,

```
>> z = [1+i, 2, 3i]
z =
   1.0000 + 1.0000i   2.0000                  0 + 3.0000i
>> y = [1+i, 2, 3 i]
y =
   1.0000 + 1.0000i   2.0000            3.0000              0 + 1.0000i
>>
```

2.2.3 Le type chaîne de caractères

Une chaîne de caractères est un tableau de caractères. Une donnée de type chaîne de caractères (`char`) est représentée sous la forme d'une suite de caractères encadrée d'apostrophes simples (`'`). Une variable de type chaîne de caractères étant interprétée comme un tableau de caractères, il est possible de manipuler chaque lettre de la chaîne en faisant référence à sa position dans la chaîne. La concaténation de chaînes de caractères s'effectue selon les règles de manipulation des tableaux, voir le paragraphe 2.3.

L'exemple suivant présente différentes manipulations d'une chaîne de caractères.

```
>> ch1 = 'bon'
ch1 =
bon
>> ch2 = 'jour'
ch2 =
jour
>> whos
  Name      Size         Bytes  Class

  ch1       1x3              6  char array
  ch2       1x4              8  char array

Grand total is 7 elements using 14 bytes

>> ch = [ch1,ch2]
ans =
bonjour
>> ch(1), ch(7), ch(1:3)
ans =
b
ans =
r
ans =
bon
>> ch3 = 'soi';
>> ch = [ch(1:3), ch3, ch(7)]
ans =
bonsoir
>>
```

Si une chaîne de caractères doit contenir le caractère apostrophe (') celui-ci doit être doublé dans la chaîne (ainsi pour affecter le caractère apostrophe (') à une variable on devra écrire '''', soit 4 apostrophes).

```
>> rep = 'aujourd'hui'
??? rep = 'aujourd'hui
                    |
Missing operator, comma, or semi-colon.

>>  rep = 'aujourd''hui'
rep =
aujourd'hui
>>  apos = ''''
apos =
'
>>
```

La chaîne de caractères vide s'obtient par 2 apostrophes ''. La commande `isempty` permet de tester si une variables de type chaîne de caractères est vide ou non. La commande

`strcmp` permet de tester si deux chaines de caractères sont égales ou non.

2.2.4 Le type logique

Le type logique (logical) possède 2 formes : 0 pour faux et 1 pour vrai. Un résultat de type logique est retourné par certaines fonctions ou dans le cas de certains tests.

Dans l'exemple qui suit on considère une variable `x` contenant la valeur 123 et une variable `y` définie par l'expression mathématique $y = exp(log(x))$. On teste si les variables `x` et `y` contiennent les mêmes valeurs. La variable `tst` est une variable de type logique qui vaut 1 (vrai) les valeurs sont égales et 0 (faux) sinon. Suivant la valeur de `tst`, on affiche la phrase `x est egal a y` ou la phrase `x est different de y`. Dans l'exemple proposé, compte-tenu des erreurs d'arrondis lors du calcul de $exp(log(123))$, la valeur de la variable `y` ne vaut pas exactement 123. On pourra également considérer le cas où $x = 12$.

```
>> x = 123; y = exp(log(x));
>> tst = ( x==y );
>> if tst, disp('x est egal a y '), else disp('x est different de y '), end
x est different de y
>> whos
  Name      Size         Bytes  Class

  tst       1x1              8  double array (logical)
  x         1x1              8  double array
  y         1x1              8  double array

Grand total is 3 elements using 24 bytes

>> format long
>> x, y, tst
x =
   123
y =
     1.229999999999999e+02
tst =
     0
>> format, clear
>>
```

2.3 Les vecteurs

2.3.1 Définir un vecteur

On définit un vecteur ligne en donnant la liste de ses éléments entre crochets (`[ ]`). Les éléments sont séparés au choix par des espaces ou par des virgules. On définit un vecteur colonne en donnant la liste de ses éléments séparés au choix par des points virgules (`;`) ou par des retours chariots (touche Entrée/Enter). On peut transformer un vecteur ligne `x` en un vecteur colonne et réciproquement en tapant `x'` (`'` est le symbole de transposition).

Il est inutile de définir la longueur d'un vecteur au préalable. Cette longueur sera établie automatiquement à partir de l'expression mathématique définissant le vecteur ou à partir des données. On peut obtenir la longueur d'un vecteur donné grâce à la commande `length`.

Un vecteur peut également être défini « par blocs » selon la même syntaxe. Si par exemple x_1, x_2 et x_3 sont trois vecteurs (on note `x1`, `x2` et `x3` les variables MATLAB correspondantes), on définit le vecteur $X = (x_1 \mid x_2 \mid x_3)$ par l'instruction `X = [x1 x2 x3]`.

```
>> x1 = [1 2 3], x2 = [4,5,6,7], x3 = [8; 9; 10]
x1 =
     1     2     3
x2 =
     4     5     6     7
x3 =
     8
     9
    10
>> length(x2), length(x3)
ans =
     4
ans =
     3
>> whos
  Name      Size          Bytes  Class

  x1        1x3              24  double array
  x2        1x4              32  double array
  x3        3x1              24  double array

Grand total is 10 elements using 80 bytes

>> x3'
ans =
     8     9    10
>> X = [x1 x2 x3']
X =
     1     2     3     4     5     6     7     8     9    10
>>
```

Les éléments d'un vecteur peuvent être manipulés grâce à leur indice dans le tableau. Le k^e élément du vecteur `x` est désignée par `x(k)`. Le premier élément d'un vecteur a obligatoirement pour indice 1. En pratique ceci impose de faire des translations d'indices si par exemple on souhaite définir une suite $x_n, n = 0, \cdots, N$. Le terme x_0 de la suite correspondra à l'élément `x(1)` du vecteur et le terme x_N à l'élément `x(N+1)`. Il est possible de manipuler plusieurs éléments d'un vecteur simultanément. Ainsi les éléments k à l du vecteur `x` sont désignés par `x(k :l)`. On peut également manipuler facilement les éléments d'un vecteur dont les indices sont en progression arithmétique. Ainsi si l'on souhaite extraire les éléments $k, k+p, k+2p, \cdots, k+Np = l$ on écrira `x(k :p :l)`. Plus généralement, si `K` est un vecteur de valeurs entières, `X(K)` retourne les éléments du vecteur `X` dont les indices sont les éléments du vecteur `K`.

Reprenons l'exemple précédent.

```
>> X(5)
ans =
     5
>> X(4:10)
ans =
     4     5     6     7     8     9    10
>> X(2:2:10)
ans =
     2     4     6     8    10
>> K = [1 3 4 6]; X(K)
ans =
     1     3     4     6
>>
```

Il est très facile de définir un vecteur dont les composantes forment une suite arithmétique. Pour définir un vecteur `x` dont les composantes forment une suite arithmétique de raison h, de premier terme a et de dernier terme b, on écrira `x = a :h :b`. Si $a - b$ n'est pas un multiple de h, le dernier élément du vecteur `x` sera $a + Ent((a-b)/h)\ h$ où Ent est la fonction partie entière. La commande `linspace` permet de définir un vecteur `x` de longueur N dont les composantes forment une suite arithmétique de premier terme a et de dernier terme b (donc de raison $(a-b)/N$). Les composantes du vecteur sont donc *lin*éairement e*spac*és. La syntaxe est `x = linspace(a,b,N)`.

```
>> x = 1.1:0.1:1.9
x =
  Columns 1 through 7
    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000    1.7000
  Columns 8 through 9
    1.8000    1.9000
>> x = 1.1:0.2:2
x =
    1.1000    1.3000    1.5000    1.7000    1.9000
>> x = linspace(1.1,1.9,9)
ans =
  Columns 1 through 7
    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000    1.7000
  Columns 8 through 9
    1.8000    1.9000
>>
```

2.3.2 Vecteurs spéciaux

Les commandes `ones`, `zeros` et `rand` permettent de définir des vecteurs dont les éléments ont respectivement pour valeurs 0, 1 et des nombres générés de manière aléatoire.

`ones(1,n)`	:	vecteur ligne de longueur n dont tous les éléments valent 1
`ones(m,1)`	:	vecteur colonne de longueur m dont tous les éléments valent 1
`zeros(1,n)`	:	vecteur ligne de longueur n dont tous les éléments valent 0
`zeros(m,1)`	:	vecteur colonne de longueur m dont tous les éléments valent 0
`rand(1,n)`	:	vecteur ligne de longueur n dont les éléments sont générés de manière aléatoire entre 0 et 1
`rand(m,1)`	:	vecteur colonne de longueur m dont les éléments sont générés de manière aléatoire entre 0 et 1

2.4 Les matrices

2.4.1 Définir une matrice

On a déjà vu que l'on définissait la matrice

$$A = \begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}$$

en tapant `A = [ 1 3 ; 4 2 ]`.

D'une façon générale, on définit une matrice en donnant la liste de ses éléments entre crochets. Signalons que MATLAB admet d'autres façons d'écrire les matrices. Les éléments d'une ligne de la matrice peuvent être séparés au choix par un blanc ou bien par une virgule (,). Les lignes quant à elles peuvent être séparées au choix par le point-virgule (;) ou par un retour chariot. Par exemple, on peut aussi écrire la matrice A de la manière suivante,

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

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

```
>> A = [1,3
        4,2]
A =
     1     3
     4     2
>>
```

Un élément d'une matrice est référencé par ses numéros de ligne et de colonne. `A(i,j)` désigne le i^{e}élément de la j^{e}ligne ligne de la matrice `A`. Ainsi `A(2,1)` désigne le premier élément de la deuxième ligne de `A`,

```
>> A(2,1)
ans =
      4
>>
```

La commande `size` permet d'obtenir les dimensions d'une matrice `A` donnée. On peut soit obtenir de manière séparée le nombre de lignes et de colonnes par les instructions

`size(A,1)` et `size(A,2)` respectivement, soit obtenir le nombre m de lignes et le nombre n de colonnes par l'instruction `[m,n] = size(A)`.

On peut construire très simplement une matrice « par blocs ». Si `A`, `B`, `C`, `D` désignent 4 matrices (aux dimensions compatibles), on définit la matrice blocs

$$K = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

par l'instruction `K = [ A B ; C D]`.

Voici un exemple de construction par blocs de la matrice

$$B = \left(\begin{array}{ccc|cc|c} 35 & 1 & 6 & 26 & 19 & 24 \\ 3 & 32 & 7 & 21 & 23 & 25 \\ 31 & 9 & 2 & 22 & 27 & 20 \\ \hline 8 & 28 & 33 & 17 & 10 & 15 \\ 30 & 5 & 34 & 12 & 14 & 16 \\ \hline 4 & 36 & 29 & 13 & 18 & 11 \end{array}\right)$$

```
>> A11 = [35 1 6; 3 32 7; 31 9 2];
>> A12 = [26 19; 21 23; 22 27];
>> A21 = [ 8 28 33; 30 5 34];
>> A22 = [17 10; 12 14];
>> B11 = [ A11 A12; A21 A22 ]
B11 =
    35     1     6    26    19
     3    32     7    21    23
    31     9     2    22    27
     8    28    33    17    10
    30     5    34    12    14
>> B12 = [ 24 25 20 15 16]';
>> B = [ B11 B12];
>> B21 = [ 4 36 29 13 18 11];
>> B = [ B ; B21]
B =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
>>
```

2.4.2 Matrices spéciales

Certaines matrices se construisent très simplement grâce à des commandes dédiées. Citons les plus utilisées :

`eye(n)`	:	la matrice identité dans $\mathbb{R}^{n,n}$
`ones(m,n)`	:	la matrice à `m` lignes et `n` colonnes dont tous les éléments valent 1
`zeros(m,n)`	:	la matrice à `m` lignes et `n` colonnes dont tous les éléments valent 0
`rand(m,n)`	:	une matrice à `m` lignes et `n` colonnes dont les éléments sont générés de manière aléatoire entre 0 et 1

Signalons que si les entiers m et n sont égaux on peut se contenter de ne spécifier qu'une seule valeur de dimension : `ones(n)` est la matrice carrée de dimension n dont tous les éléments valent 1. Mentionnons enfin la commande `magic(n)` qui permet d'obtenir une matrice magique de dimension n.

```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
>> ones(3,2)
ans =
     1     1
     1     1
     1     1
>> zeros(2)
ans =
     0     0
     0     0
>> rand(2,3)
ans =
    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919
>> magic(6)
ans =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
>>
```

2.4.3 Manipuler des matrices

Le symbole deux-points (:) permet d'extraire simplement des lignes ou des colonnes d'une matrice. Le j^{e} vecteur colonne de la matrice `A` est désigné par `A( :,j)`. C'est simple, il suffit de traduire le symbole deux-points (:) par « tout ». Ainsi `A( :,j)` désigne toutes les lignes et la j^{e} colonne de la matrice `A`. Bien entendu, la i^{e} ligne de la matrice `A` est désignée par `A(i, :)`.

```
>> A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
>>  A(1,:)
ans =
    17    24     1     8    15
>>  A(:,2)
ans =
    24
     5
     6
    12
    18
>>
```

Si l'on souhaite échanger les colonnes 2 et 3 de la matrice A une première possibilité consiste à exécuter :

```
>> v = A(:,2); A(:,2) = A(:,3); A(:,3) = v;
A =
    17     1    24     8    15
    23     7     5    14    16
     4    13     6    20    22
    10    19    12    21     3
    11    25    18     2     9
>>
```

On peut également extraire plusieurs lignes ou colonnes simultanément. Si `J` est un vecteur d'entiers, `A( :,J)` est la matrice issue de `A` dont les colonnes sont les colonnes de la matrice `A` d'indices contenus dans le vecteur `J`. De même `A(J, :)` est la matrice issue de `A` dont les lignes sont les lignes de la matrice `A` d'indices contenus dans le vecteur `J`.

D'une façon plus générale, il est possible de n'extraire qu'une partie des éléments des lignes et colonnes d'une matrice. Si `L` et `C` sont deux vecteurs d'indices, `A(L,C)` désigne la matrice issue de la matrice `A` dont les éléments sont les `A(i,j)` tels que `i` $\in$ `L` et `j` $\in$ `C`.

```
>> A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
>> L = [1 3 5]; C = [3 4];
>> A(L,C)
ans =
     1     8
    13    20
    25     2
>> A(1:2:5,3:4)
ans =
     1     8
    13    20
    25     2
>>
```

Dans la dernière instruction, on a utilisé la forme spéciale permettant de définir un vecteur dont les composantes sont en progression arithmétique, voir le paragraphe 2.3.

Une seconde possibilité pour échanger les lignes 2 et 3 de la matrice `A` consiste à exécuter :

```
>> J = [1 3 2 4]; A = A(:,J)
A =
    17     1    24     8    15
    23     7     5    14    16
     4    13     6    20    22
    10    19    12    21     3
    11    25    18     2     9
```

```
>>
```

Il existe des commandes MATLAB permettant de manipuler globalement des matrices. La commande `diag` permet d'extraire la diagonale d'une matrice : si `A` est une matrice, `v=diag(A)` est le vecteur composé des éléments diagonaux de `A`. Elle permet aussi de créer une matrice de diagonale fixée : si `v` est un vecteur de dimension n, `A=diag(v)` est la matrice diagonale dont la diagonale est `v`.

```
>> A=eye(3); diag(A)
ans =
     1
     1
     1
>> v=[1:3]
v =
     1 2 3
```

```
>> diag(v)
 ans =
     1 0 0
     0 2 0
     0 0 3
>>
```

On n'est pas obligé de se limiter à la diagonale principale ...La commande `diag` admet un second paramètre k pour désigner la k^{e}sur-diagonale (si $k > 0$) ou la k^{e}sous-diagonale (si $k < 0$).

```
>> A = [4 5 6 7 ; 3 4 5 6
        2 3 4 5; 1 2 3 4]
A =
     4     5     6     7
     3     4     5     6
     2     3     4     5
     1     2     3     4
>> diag(A,1)
ans =
     5
     5
     5
>> diag(A,-2)
ans =
     2
     2
>>
```

On construit à l'aide de la commande `diag` très simplement des matrices tridiagonales. Par exemple la matrice correspondant à la discrétisation par différences finies du problème de Dirichlet en dimension 1 s'obtient ainsi

```
>> N=5;
>> A=diag(2*ones(N,1)) - diag(ones(N-1,1),1) - diag(ones(N-1,1),-1)
A =
     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
>>
```

On dispose également de la commande `tril` permet d'obtenir la partie triangulaire inférieure (l pour lower) d'une matrice. La commande `triu` permet d'obtenir la partie triangulaire supérieure (u pour upper) d'une matrice.

```
>> A = [ 2 1 1 ; -1 2 1 ; -1 -1 2]
A =
     2     1     1
    -1     2     1
    -1    -1     2
>> triu(A)
ans =
     2     1     1
     0     2     1
     0     0     2
```

```
>> tril(A)
ans =
     2     0     0
    -1     2     0
    -1    -1     2
>>
```

Comme pour la commande `diag`, les commandes `triu` et `tril` admettent un second paramètre k. On peut ainsi obtenir la partie triangulaire supérieure (ou inférieure) à partir de la k^{e}diagonale. Ainsi,

```
>> tril(A,-1)
ans =
     0     0     0
    -1     0     0
    -1    -1     0
>> tril(A,1)
ans =
     2     1     0
    -1     2     1
    -1    -1     2
>>
```

On obtient la transposée de la matrice `A` à coefficients réels en tapant `A'`. Si la matrice est à coefficients complexes, `A'` retourne la matrice adjointe de `A`.

```
>> A = [0 1 2; -1 0 1; -2 -1 0]
A =
     0     1     2
    -1     0     1
    -2    -1     0
```

```
>> A'
ans =
     0    -1    -2
     1     0    -1
     2     1     0
```

2.5 La structure sparse

On appelle matrice creuse (le terme anglais est « sparse matrix ») une matrice comportant une forte proportion de coefficients nuls. De nombreux problèmes issus de la physique conduisent à l'analyse de systèmes linéaires à matrice creuse. L'intérêt de telles matrices résulte non seulement de la réduction de la place mémoire (on ne stocke pas les zéros) mais aussi de la réduction du nombre d'opérations (on n'effectuera pas les opérations portant sur les zéros). Par défaut dans MATLAB une matrice est considérée comme pleine (ou « full » en anglais), c'est-à-dire que tous ses coefficients sont mémorisés. Si M est une matrice, la commande `sparse(M)` permet d'obtenir la même matrice mais stockée sous la forme sparse. Si l'on a une matrice stockée sous la forme sparse, on peut obtenir la même matrice stockée sous la forme ordinaire par la commande `full`. Il est possible de visualiser graphiquement la structure d'une matrice grâce à la commande `spy`. Si M est une matrice, la commande `spy(M)` ouvre une fenêtre graphique et affiche sous forme de croix les éléments non-nuls de la matrice. Le numéro des lignes est porté sur l'axe des ordonnées, celui des colonnes en

abscisse. On obtient également le nombre d'éléments non-nuls de la matrice. La commande `nnz` permet d'obtenir le nombre d'éléments non-nuls d'une matrice.

Reprenons l'exemple de la matrice tridiagonale issue de la discrétisation par différences finies du problème de Dirichlet en dimension 1.

```
>> N=5;
>> A=diag(2*ones(N,1)) - diag(ones(N-1,1),1) - diag(ones(N-1,1),-1)
A =
     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
>> nnz(A)
ans =
    13
>> B = sparse(A)
B =
   (1,1)        2
   (2,1)       -1
   (1,2)       -1
   (2,2)        2
   (3,2)       -1
   (2,3)       -1
   (3,3)        2
   (4,3)       -1
   (3,4)       -1
   (4,4)        2
   (5,4)       -1
   (4,5)       -1
   (5,5)        2
>> whos
  Name      Size          Bytes  Class

  A         5x5             200  double array
  B         5x5             180  sparse array
  N         1x1               8  double array

Grand total is 39 elements using 388 bytes
>> spy(A)
>>
```

Pour les très grosses matrices creuses, il n'est bien entendu pas souhaitable d'utiliser une structure pleine (full) pour définir la matrice avant de passer en stockage sparse. La commande `sparse` permet de définir une matrice creuse directement sous la forme sparse. On l'utilise de la façon suivante : `A = sparse(is,js,s,n,m)` pour définir une matrice $A \in \mathbb{R}^{n,m}$ dont les coefficients sont nuls (et donc non mémorisés) sauf les éléments $a_{\texttt{is(l)},\texttt{js(l)}}$ qui valent `s(l)` pour l variant de 1 à L longueur du tableau `s` (`L = length(s)`).

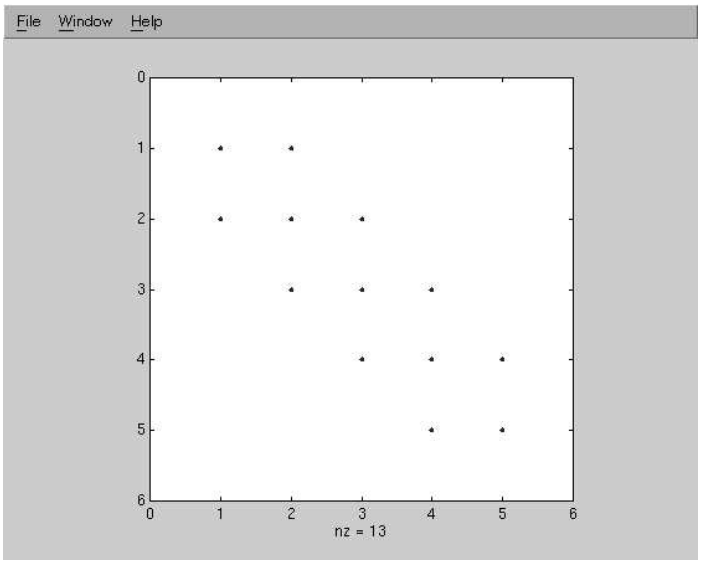

FIG. 2 – Résultat de la commande `spy(B)`.

```
>> N=5;
>> s =[2*ones(1,N), -ones(1,N-1), -ones(1,N-1)]
s =
  Columns 1 through 12
     2     2     2     2     2    -1    -1    -1    -1    -1    -1    -1
  Column 13
    -1
>> is = [1:N,1:N-1,2:N]
is =
  Columns 1 through 12
     1     2     3     4     5     1     2     3     4     2     3     4
  Column 13
     5
>> js = [1:N,2:N,1:N-1]
js =
  Columns 1 through 12
     1     2     3     4     5     2     3     4     5     1     2     3
  Column 13
     4
>> B = sparse(is,js,s,N,N)
B =
   (1,1)        2
   (2,1)       -1
   (1,2)       -1
   (2,2)        2
   (3,2)       -1
   (2,3)       -1
   (3,3)        2
   (4,3)       -1
   (3,4)       -1
```

```
   (4,4)        2
   (5,4)       -1
   (4,5)       -1
   (5,5)        2
>> A = full(B)
A =
     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
>>
```

Les commandes `spdiags`, `speye` et `sprand` permettent de construire des matrices diagonales, identités et des matrices dont les éléments sont des nombres aléatoires avec un stockage « sparse ». Elles s'utilisent de la même manière que leur équivalent `diag`, `eye` et `rand` pour les matrices pleines.

3 Calculer avec MATLAB

3.1 Les constantes

Les principales constantes sont :

`pi`	:	3.1415926535897
`i`	:	$\sqrt{-1}$
`j`	:	$\sqrt{-1}$
`eps`	:	précision numérique relative
`realmin`	:	plus petit nombre à virgule flottante manipulable
`realmax`	:	plus grand nombre à virgule flottante manipulable
`inf`	:	infini. Est obtenu quand on essaie d'évaluer une expression dont le résultat excède `realmax`
`NaN`	:	not-a-number. Est obtenu quand on essaie d'effectuer une opération non-définie comme $0/0$

Les valeurs des constantes `eps`, `realmin` et `realmax` dépendent de la machine sur laquelle MATLAB est installé. Par exemple sur une station SUN Ultra 1 on a `eps`$= 2.2204e-16$, `realmin`$= 2.2251e-308$ et `realmax`$= 1.7977e+308$.

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

```
>> pi = 0; cos(pi)
ans =
     1
>> clear pi
>>  cos(pi)
ans =
     -1
>>
```

3.2 Opérations et fonctions portant sur les scalaires

Il est bien entendu possible d'utiliser MATLAB pour faire de simples additions (`-`:. Si `x` et `y` sont deux variables scalaires de type réel, `x+y`, `x-y`, `x*y` et `x/y` désignent les 4 opérations usuelles entre les valeurs de `x` et `y` dans $\mathbb{R}$. Si `x` et `y` sont deux variables scalaires de type complexe, `x+y`, `x-y`, `x*y` et `x/y` désignent les 4 opérations usuelles entre les valeurs de `x` et `y` dans $\mathbb{C}$. L'exponentiation s'obtient grâce au symbole `^` (la syntaxe est `x^y`).

La commande `rem` donne le reste (remainder) de la division entière de deux entiers (la syntaxe est `rem(m,n)`). Les commandes `lcm(m,n)` et `gcd(m,n)` retournent respectivement

le plus petit multiple commun et le plus grand commun diviseur à deux entiers m et n. La commande `factor(n)` permet d'obtenir les termes de la décomposition en facteurs premiers de l'entier n.

Les fonctions mathématiques incorporées sont :

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

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

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

Les fonctions d'arrondis sont :

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

Les fonctions trigonométriques et hyperboliques sont :

`cos` : cosinus,

`acos` : cosinus inverse (arccos),

`sin` : sinus,

`asin` : sinus inverse (arcsin),

`tan` : tangente,

`atan` : tangente inverse (arctan),

`cosh` : cosinus hyperbolique (ch),

`acosh` : cosinus hyperbolique inverse (argch),

`sinh` : sinus hyperbolique (sh),

`asinh` : sinus hyperbolique inverse (argsh),

`tanh` : tangente hyperbolique (th),

`atanh` : tangente hyperbolique inverse (argth).

3.3 Opérations et fonctions portant sur les vecteurs

Une particularité de MATLAB est de permettre d'effectuer des opérations de manière globale sur les éléments d'un vecteur de type réel ou complexe sans avoir à manipuler directement ses éléments. Si `k` est une variable scalaire et `x` un vecteur, l'instruction `k*x` multiplie tous les éléments de `x` par `k`. Si `x` et `y` sont deux vecteurs de longueur identique, l'instruction `z = x+y` (respectivement `x-y`) définit le vecteur `z` dont les éléments sont `z(i) = x(i) + y(i)` (respectivement `z(i) = x(i) - y(i)`). On obtient un vecteur `z` dont la i^{e}composante est le produit (respectivement le quotient) de la i^{e}composante du vecteur `x` par la i^{e}composante du vecteur `y` en effectuant l'instruction `z = x.*y` (respectivement `z = x./y`). Attention à ne pas oublier le point !

La commande `cross(x,y)` permet de calculer le produit vectoriel des deux vecteurs `x` et `y`. Il n'y a pas de commande dédiée pour calculer le produit scalaire de deux vecteurs. Il s'obtient grâce à l'instruction `sum(x.*y)`.

Les fonctions mathématiques incorporées décrites au paragraphe 3.2 peuvent être utilisées avec un argument qui est un vecteur. La fonction est alors appliquée à tous les éléments du vecteur en même temps.

```
>>  x = [1:10:100]; y=sqrt(x)
y =
  Columns 1 through 7
    1.0000    3.3166    4.5826    5.5678    6.4031    7.1414    7.8102
  Columns 8 through 10
    8.4261    9.0000    9.5394
>>
```

Il existe également quelques fonctions spécifiques aux vecteurs[2] :

`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,

`fliplr(x)` : échange la position des éléments du vecteur `x`.

```
>> x=[3 1 2];
>> sum(x)
ans =
     6
>> prod(x)
ans =
     6
>> max(x)
ans =
     3
```

```
>> min(x)
ans =
     1
>> sort(x)
ans =
     1     2     3
>> fliplr(x)
ans =
     2     1     3
>>
```

Citons enfin quelques fonctions logiques. Les commandes `all`[2]et `any`[2]servent à tester si les éléments d'un vecteur sont nuls ou non. Si `v` est un vecteur de valeurs numériques, `all(v)` retourne vrai (1) si tous les éléments du vecteur sont différents de 0 et faux (0) si au moins un élément vaut 0. `any(v)` retourne vrai (1) si au moins un élément du vecteur est différent de 0 et faux (0) si le vecteur est composé exclusivement de 0.

3.4 Opérations et fonctions portant sur les matrices

Si les opérandes sont des matrices, les opérations `+` (addition), `-` (soustraction), `*` (multiplication), `^` (exponentiation), sont alors les opérations matricielles usuelles. Ainsi `A*B` désigne le produit de la matrice `A` par la matrice `B`, `A+B` désigne la somme de ces deux matrices et `A^2` le carré de la matrice `A`.

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

```
>> C = A*B
C =
     14 14
     32 32
>> C^2
ans =
     644  644
    1472 1472
>>
```

[2]Ces commandes peuvent également être utilisées avec une matrice. Dans ce cas la commande porte sur chaque vecteur colonne de la matrice.

Si les dimensions des matrices `A` et `B` sont incompatibles avec l'opération matricielle, MATLAB renvoi un message d'erreur :

```
>> A+B

??? Error using ==> + Matrix dimensions must agree.
>>
```

En plus des opérations matricielles usuelles, il est possible d'effectuer des opérations entre matrices «élément par élément». Pour cela, il faut faire précéder l'opérateur d'un point (.). Ainsi si `A` et `B` sont 2 matrices de même dimension, on obtient la matrice dont le terme d'indices (i, j) est le produit des deux termes d'indices (i, j) des matrices `A` et `B` par la commande `A.*B`. De même la commande `A.^2` fournit la matrice dont les termes sont les carrés des termes de la matrice `A`. Bien entendu les commandes `A.+B` et `A+B` donnent le même résultat.

```
>> A=[1 2 3; 4 5 6]
A =
    1 2 3
    4 5 6
>> B=[1 2 3; 1 2 3]
B =
     1 2 3
     1 2 3
>> A.*B
ans =
     1 4 9
     4 10 18
>> A.^2
ans =
     1     4     9
    16    25    36
>>
```

Les fonctions matricielles les plus courantes sont :

`det(A)` : renvoie le déterminant de la matrice carrée `A`.

`eig(A)` : renvoie les valeurs propres (eigenvalues) de la matrice carrée `A`. Si l'on souhaite également les vecteurs propres on exécutera `[V,D] = eig(A)` qui renvoie une matrice diagonale `D` formée des valeurs propres de `A` et une matrice `V` dont les vecteurs colonnes sont les vecteurs propres correspondant.

`poly(A)` : renvoie les coefficients du polynôme caractéristique associé à la matrice carrée `A`. On sera vigilant à l'ordre dans lequel sont rangés les coefficients : le premier élément du vecteur est le coefficient du monôme de plus haut degré. Ainsi dans l'exemple suivant il faut lire $p(\lambda) = \lambda^3 - 6\lambda^2 - 72\lambda - 27$,

```
>> A = [1 2 3 ; 4 5 6 ; 7 8 0]; p = poly(A)
p =
     1    -6    -72    -27
```

`inv(A)` : renvoie l'inverse de la matrice carrée `A`.

`rank(A)` : renvoie le rang de la matrice carrée `A`.

`trace(A)` : renvoie la trace de la matrice `A`.

`expm(A)` : renvoie l'exponentielle matricielle de `A`.

On peut obtenir les différentes normes d'une matrice `A` grâce à la commande `norm`.

`norm(A)` : renvoie la norme 2 de la matrice `A`.

`norm(A,2)` : même chose que `norm(A)`.

`norm(A,1)` : norme 1 de la matrice `A`, $\|A\|_1 = \max\limits_{1\leq j\leq n} \sum\limits_{1\leq i\leq n} |a_{ij}|$.

`norm(A,inf)` : norme infini de la matrice `A`, $\|A\|_\infty = \max\limits_{1\leq i\leq n} \sum\limits_{1\leq j\leq n} |a_{ij}|$.

`norm(A,'fro')` : norme de Frobenius de la matrice `A`, $\|A\|_{fro} = \sqrt{\sum\limits_{1\leq i,j\leq n} |a_{ij}|^2}$.

Ces fonctions matricielles incorporée de MATLAB peuvent être utilisées avec un argument qui est une matrice « sparse ». Les exceptions sont les fonctions `rank`, `expm` et `norm` qui nécessitent de passer en stockage `full` (on exécutera donc `rank(full(B))` par exemple).

3.5 Résolution de systèmes linéaires

La commande MATLAB \ (backslash) est la commande générique pour résoudre un système linéaire. L'algorithme mis en œuvre dépend de la structure de la matrice `A` du système. MATLAB utilise dans l'ordre les méthodes suivantes :

- Si `A` est une matrice triangulaire, le système est résolu par simple substitution.
- Si la matrice `A` est symétrique ou hermitienne, définie positive, la résolution est effectuée par la méthode de Choleski.
- Si `A` est une matrice carrée mais n'entrant pas dans les deux cas précédents, une

factorisation LU est réalisée en utilisant la méthode d'élimination de Gauss avec stratégie de pivot partiel.

– Si `A` n'est pas une matrice carrée, la méthode QR est utilisée.

Dans le cas des matrices stockées sous forme sparse, des algorithmes particuliers sont mis en œuvre. Chacune des méthodes précédentes peut être utilisée de manière spécifique grâce aux commandes `chol`, `lu`, `qr`. Il est également possible d'utiliser des méthodes itératives. Les commandes `cgs`, `bicg`, `bicgstab` mettent par exemple en œuvre des méthodes de type gradient conjugué.

```
>> A=[1 2 ;3 4]; b=[1 1]';
>>  x= A\b
x =
    -1
     1
>> A*x
ans =
     1
     1
>>
```

3.6 Les polynômes

Sous MATLAB le polynôme de degré n, $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ est défini par un vecteur `p` de dimension $n+1$ contenant les coefficients $\{a_i\}_{i=0,\ldots,n}$ rangés dans l'ordre décroissant des indices. C'est-à-dire que l'on a `p(1) = a_n`, ..., `p(n+1) = a_0`.

La commande `polyval` permet d'évaluer le polynôme `p` (la fonction polynômiale) en des points donnés. La syntaxe est `polyval(p,x)` où `x` est une valeur numérique ou un vecteur. Dans le second cas on obtient un vecteur contenant les valeurs de la fonction polynômiale aux différents points spécifiés dans le vecteur `x`. Utilisée avec la commande `fplot`, la commande `polyval` permet de tracer le graphe de la fonction polynômiale sur un intervalle $[x_{min}, x_{max}]$ donné. La syntaxe de l'instruction est

`fplot('polyval([ a_n, ..., a_0] , x)' , [x_min , x_max])`.

Voici par exemple comment définir le polynôme $p(x) = x^2 - 1$. Le graphe de la fonction polynômiale est présenté à la figure 3.

```
>> p = [ 1, 0, -1];
>> polyval(p,0)
ans =
    -1
>> polyval(p,[-2,-1,0,1,2])
ans =
     3     0    -1     0     3
>> fplot('polyval([ 1, 0, -1] , x)' , [-3,3]), grid
>>
```

On obtient les racines du polynôme `p` grâce à l'instruction `roots(p)`.

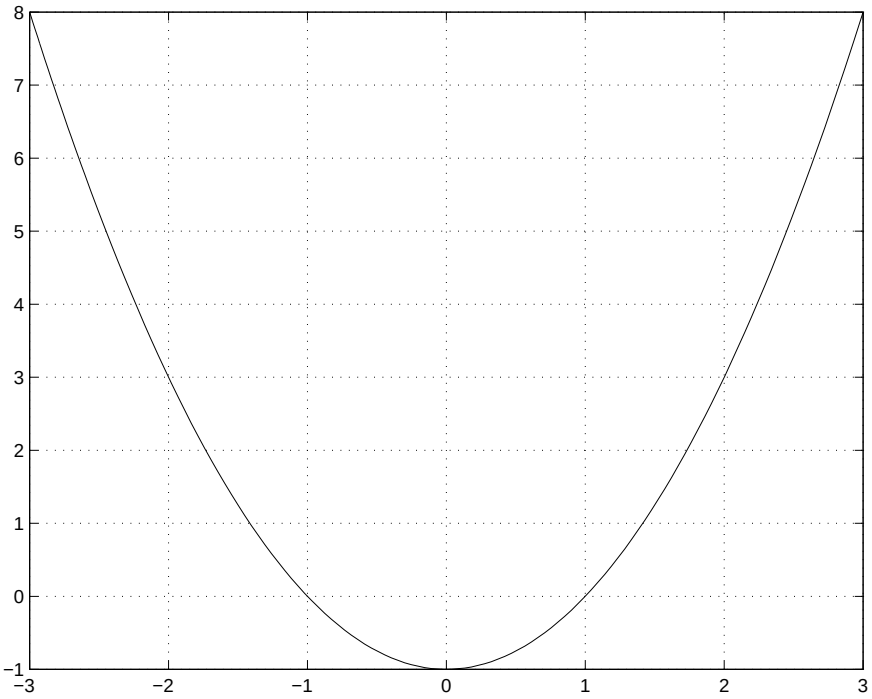

FIG. 3 – Graphe de la fonction polynômiale $p(x) = x^2 - 1$.

L'instruction `poly` permet d'obtenir la représentation canonique $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ d'un polynôme de degré n dont on connait les n racines $\{x_i\}_{i=1,\ldots,n}$. Les coefficients $\{a_i\}_{i=0,\ldots,n}$ sont obtenus sous forme d'un vecteur `p` et sont rangés dans l'ordre décroissant des indices. C'est-à-dire que `p(1) = a_n`, ..., `p(n+1) = a_0`.

```
>> r = roots(p)
r =
   -1.0000
    1.0000
>> poly(r)
ans =
    1.0000    0.0000   -1.0000
>>
```

4 Les entrées-sorties

4.1 Les formats d'affichage des réels

MATLAB dispose de plusieurs formats d'affichage des réels. Par défaut le format est le format court à 5 chiffres. Les autres principaux formats sont :

`format long`	:	format long à 15 chiffres.
`format short e`	:	format court à 5 chiffres avec notation en virgule flottante.
`format long e`	:	format long à 15 chiffres avec notation en virgule flottante.

MATLAB dispose également des formats `format short g` et `format long g` qui utilise la « meilleure » des deux écritures à virgule fixe ou à virgule flottante. On obtiendra tous les formats d'affichage possibles en tapant `help format`.

On impose un format d'affichage en tapant l'instruction de format correspondante dans la fenêtre de contrôle, par exemple `format long`. Pour revenir au format par défaut on utilise la commande `format` ou `format short`.

```
>> pi
ans =
    3.1416

>> format long
>> pi
ans =
   3.14159265358979

>>  format short e
>> pi^3
ans =
   3.1006e+01

>> format short g
>> pi^3
ans =
       31.006

>> format short
>>
```

4.2 Affichage simple, la commande `disp`

La commande `disp` permet d'afficher un tableau de valeurs numériques ou de caractères. L'autre façon d'afficher un tableau est de taper son nom. La commande `disp` se contente d'afficher le tableau sans écrire le nom de la variable ce qui peut améliorer certaines présentations.

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

```
     4    14    15     1
>>
```

On utilise fréquemment la commande `disp` avec un tableau qui est une chaîne de caractères pour afficher un message. Par exemple `disp('Calcul du déterminant de la matrice A')`. On utilise également la commande `disp` pour afficher un résultat. Par exemple `disp(['Le déterminant de la matrice A vaut ', num2str(det(A))])`. On remarque que l'usage de la commande `disp` est alors un peu particulier. En effet un tableau doit être d'un type donné, les éléments d'un même tableau ne peuvent donc être des chaînes de caractères et des valeurs numériques. On a donc recours à la commande `num2str` (« number to string ») pour convertir une valeur numérique en une chaîne de caractères. Par défaut la commande `num2str` affiche 4 décimales mais il est possible de lui spécifier le nombre de décimales souhaité en second paramètre. De même il est possible de lui spécifier un format d'affichage particulier de la valeur numérique ; on consultera la documentation MATLAB pour plus de détails. Attention, si la chaîne de caractères contient une apostrophe il est impératif de doubler l'apostrophe.

4.3 Lecture

La commande `input` permet de demander à l'utilisateur d'un programme de fournir des données. La syntaxe est `var = input(' une phrase ')`. La phrase `une phrase` est affichée et MATLAB attend que l'utilisateur saisisse une donnée au clavier. Cette donnée peut être une valeur numérique ou une instruction MATLAB. Un retour chariot provoque la fin de la saisie. Une valeur numérique est directement affectée à la variable `var` tandis qu'une instruction MATLAB est évaluée et le résultat est affecté à la variable `var`. Il est possible de provoquer des sauts de ligne pour aérer le présentation en utilisant le symbole `\n` de la manière suivante : `var = input(' \n une phrase : \n ')`. Pensez à mettre un point virgule (`;`) à la fin de l'instruction si vous ne souhaitez pas voir s'afficher `var =` .

Sous cette forme il est impossible d'avoir une donnée de type chaîne de caractères dans la mesure où MATLAB essaie d'interpréter cette chaîne de caractères comme une instruction. Si l'on souhaite saisir une réponse de type chaîne de caractères on utilise la syntaxe `var = input(' une phrase ','s')`. Signalons qu'un retour chariot (sans autre chose) initialise la variable `var` au tableau vide `[]`. Voici un exemple d'utilisation de la commande `input` (on suppose que la variable `res` contient une valeur numérique).

```
rep = input(' Affichage du resultat ? o/n [o] ','s');
if isempty(rep), rep = 'o'; end
if rep == 'o' |  rep == 'y'
  disp(['Le resultat vaut ', num2str(res)])
end
```

4.4 Impressions dirigées par format

La commande `sprintf` permet l'impression de variables selon un modèle donné. Un modèle d'édition se présente sous la forme du symbole pourcent (%) suivi d'indications permettant de composer le contenu du champ à imprimer, en particulier sa longueur en nombre de caractères. Le modèle d'édition utilisé par MATLAB est le modèle d'édition du langage C.

La syntaxe de la commande `sprintf` est :

`sprintf(`*format*, *variables*`)`

où

- *variables* est le nom des variables à imprimer suivant le modèle d'édition spécifié dans *format* ;
- *format* est le format d'édition. Il s'agit d'une chaîne de caractères contenant les modèles d'éditions des variables à imprimer.

Modèle d'édition de caractères

Un modèle d'édition de caractères est de la forme $\%Ls$ où % est le symbole de début de format et s le symbole précisant que la donnée est de type chaîne de caractères. L est un entier donnant la longueur total du champ (en nombre de caractères). Par défaut le champ est justifié à droite (si la longueur de la chaîne de caractères est plus petite que la longueur L du champ, des espaces sont insérés après la chaîne de caractères). Le symbole − (moins) juste après le symbole % permet de justifier à gauche. En l'absence de l'entier L la longueur totale du champ est égale au nombre de caractères de la chaîne.

```
>> sprintf('%s', 'il fera beau a Brest')
ans =
il fera beau a Brest
>> temps = 'il fera beau a Brest'; sprintf('%s',temps)
ans =
il fera beau a Brest
>> sprintf('%30s', temps)
ans =
          il fera beau a Brest
>> sprintf('%-30s', temps)
ans =
il fera beau a Brest
>> sprintf('meteo : %s', temps)
ans =
meteo : il fera beau a Brest
>>
```

Modèle d'édition des réels

Un modèle d'édition de réel est de la forme $\%\pm L.Dt$, où % est le symbole de début de format, L est un entier donnant la longueur total du champ (en nombre de caractères, point virgule compris), D est le nombre de décimales à afficher et t spécifie le type de notation utilisée. Par défaut le champ est justifié à droite (si la longueur de la variable est plus petite que la longueur du champ L, des espaces sont insérés à gauche). Le symbole − (moins) permet de justifier à gauche. Le symbole + (plus) provoque l'affichage systématique d'un signe + devant les réels positifs.

Les principales valeurs possibles pour t sont les suivantes :

`d` : pour les entiers

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

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

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

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

```
>> x = pi/3; y = sin(x);
>> sprintf('sin(%8.6f) = %4.2f', x,y)
ans =
sin(1.047198) = 0.87
>> sprintf('sin(%8.6f)  = %4.2E', x,y)
ans =
exp(1.047198)  =  8.66E-01
>>
```

Utilisations particulières

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

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

Il est possible d'utiliser les symboles suivant dans les chaînes de caractères

`\n` : provoque le passage à une nouvelle ligne

`\t` : insère une tabulation horizontale

`\b` : décale l'impression du champ suivant d'un caractère vers la gauche

`\r` : saut horizontal

```
>> z =[]; x = [1:10]; for i=1:length(x), z = [z ,x(i), log(x(i))]; end;
>> s = sprintf('%4.1f | %8.6E \n ', z )
ans =
  1.0 | 0.000000E+00
  2.0 | 6.931472E-01
  3.0 | 1.098612E+00
  4.0 | 1.386294E+00
  5.0 | 1.609438E+00
  6.0 | 1.791759E+00
  7.0 | 1.945910E+00
```

```
  8.0 | 2.079442E+00
  9.0 | 2.197225E+00
 10.0 | 2.302585E+00
>>
```

Si l'on a besoin d'afficher le caractère %, on le doublera **%%** pour qu'il ne soit pas interprété comme le début d'un format.

La commande **fprintf** est l'analogue de **sprintf** pour imprimer de variables selon un modèle donné dans un fichier.

5 Programmer sous MATLAB

5.1 Scripts et fonctions

Il est possible d'enregistrer une séquence d'instructions dans un fichier (appelé un « *M-file* ») et de les faire exécuter par MATLAB. Un tel fichier doit obligatoirement avoir une extension de la forme `.m` (d'où le nom M-file) pour être considéré par MATLAB comme un fichier d'instructions. On distingue 2 types de M-file, les fichiers de scripts et les fichiers de fonctions. Un script est un ensemble d'instructions MATLAB qui joue le rôle de programme principal. Si le script est écrit dans le fichier de nom `nom.m` on l'exécute dans la fenêtre MATLAB en tapant `nom`. Même si l'on ne souhaite pas à proprement parler écrire de programme, utiliser un script est très utile. Il est en effet beaucoup plus simple de modifier des instructions dans un fichier à l'aide d'un éditeur de texte que de retaper un ensemble d'instructions MATLAB dans la fenêtre de commande.

Les fichiers de fonctions ont deux rôles. Ils permettent à l'utilisateur de définir des fonctions qui ne figurent pas parmi les fonctions MATLAB incorporées (« built-in functions ») et de les utiliser de la même manière que ces dernières (ces fonctions sont nommées « fonctions utilisateur »). Ils sont également un élément important dans la programmation d'applications où les fonctions jouent le rôle des fonctions et procédures des langages de programmation usuels.

On définit la fonction `fonc` de la manière suivante :

`function` [$vars_1, \ldots, vars_n$] = `fonc`($vare_1, \ldots, vare_m$)

    séquence d'instructions

où

- $vars_1, \ldots, vars_n$ sont les variables de sortie de la fonction ;
- $vare_1, \ldots, vare_m$ sont les variables d'entrée de la fonction ;
- `séquence d'instructions` est le corps de la fonction.

Le fichier doit impérativement commencer par le mot-clé *`function`*. Suit entre crochets les variables de sortie de la fonction, le symbole `=`, le nom de la fonction et enfin les variables d'entrée entre parenthèses. Si la fonction ne possède qu'une seule variable de sortie, les crochets sont inutiles. Il est impératif que la fonction ayant pour nom `fonc` soit enregistrée dans un fichier de nom `fonc.m` sans quoi cette fonction ne sera pas « visible » par MATLAB.

Dans l'exemple qui suit, on définit une fonction `modulo` qui calcule la valeur de `a` modulo `n` en prenant pour système de résidus $1, \cdots, n$ au lieu de $0, \cdots, n-1$ (système de résidus considéré par la fonction incorporée `mod`). Les lignes qui suivent doivent être enregistrées dans un fichier de nom `modulo.m`.

```
function [r,q] = modulo(a,n)

% Calcule la valeur de a modulo n en prenant pour systeme de residus
% 1, ... , n au lieu de 0, ... , n-1.
%
% appel : [r,q] = modulo(a,n)
%
% Arguments de sortie :
%  r : le residu
%  q : le quotient
```

```
q = floor(a./n);
r = a - n*q;

% si le reste de la division entiere vaut 0, le residu vaut par convention n
if r == 0, r = n; end
```

Les lignes précédées du symbole `%` sont des lignes de commentaire. Les lignes de commentaire situées entre la ligne `function ...` et la 1re ligne d'instructions sont affichées si l'on demande de l'aide sur la fonction `modulo`.

```
>> help modulo

 Calcule la valeur de a modulo n en prenant pour systeme de residus
 1, ... , n au lieu de 0, ... , n-1.

 appel : [r,q] = modulo(a,n)

 Arguments de sortie :
  r : le residu
  q : le quotient

>>
```

L'appel d'une fonction utilisateur s'effectue de la même façon que l'appel de n'importe quelle fonction MATLAB :

```
>> b = 10 ; m = 4;
>> [r,q] = modulo(b,m)
r =
     2
q =
     2
>> modulo(10,5)
ans =
     5
>>
```

Remarques :

Il n'y a pas de mot-clé (par exemple end) pour indiquer la fin de la fonction. La fonction est supposée se terminer à la fin du fichier. Il est toutefois possible de provoquer un retour au programme appelant dans le corps de la fonction grâce à la commande `return`.

On ne peut écrire qu'une seule fonction par fichier (qui doit porter le nom de cette fonction). Toutefois dans la version 5 de MATLAB existe la notion de « sous-fonction ». Une sous-fonction est une fonction écrite dans le même fichier qu'une autre fonction

(dite principale) et qui ne sera utilisable que par cette fonction principale (une sous-fonction ne peut pas être appelée par un autre sous-programme que la fonction principale).
Si le fichier ne commence pas par le mot-clé `function` on a tout simplement écrit un script !

La gestion des variables d'entrée et de sortie est très souple sous MATLAB. Si l'on n'est intéressé que par le résidu et pas par le quotient, on peut se contenter de ne mettre qu'une seule variable de sortie, `v = modulo(10,4)`. Dans cet appel la variable `v` contiendra le résidu (la première variable de sortie). Par contre, même si l'on ne souhaite recueillir que le quotient, on est obligé d'effectuer un appel de la forme `[r,q] = modulo(10,4)` et donc de définir une variable inutile. Aussi, d'une manière générale, il est bon de ranger les variables de sortie par ordre « d'importance ». Il est également possible d'appeler une fonction donnée avec moins de variables d'entrée que le nombre indiqué pour la définition de la fonction (il faut bien entendu que le corps de la fonction soit programmé de sorte de prévoir cette éventualité). Il existe deux fonctions MATLAB utiles pour gérer cette situation : `nargin` qui retourne le nombre de variables d'entrée utilisés lors de l'appel et `nargout` qui retourne le nombre de variables de sortie prévues lors de l'appel. Voici un petit exemple venant illustrer ces possibilités.

```
function [A,rang] = matale(T,m,n)

% Construit une matrice A de m lignes et n colonnes ayant des elements
% entiers generes de maniere aleatoire entre 0 et T.
% Calcule le rang de la matrice si l'appel est effectue avec 2 arguments
% de sortie.
% Si la matrice est carree, le parametre n peut etre omis.
%
% Appels:
% [A,r] = Matale(T,m,n)
% [A,r] = Matale(T,m)
%     A = Matale(T,m,n)
%     A = Matale(T,m)

if nargin == 2
  A = fix(T*rand(m));
else
  A = fix(T*rand(m,n));
end

if nargout == 2
 rang = rank(A);
end
```

Dans cet exemple, on gère les variables d'entrée de la fonction de sorte de ne pas avoir besoin de donner lors de l'appel le nombre de lignes et de colonnes si la matrice est carrée. On gère aussi les variables de sortie afin de ne pas calculer le rang de la matrice si aucune variable de sortie pour le résultat n'est prévue lors de l'appel.

```
>> [A,r] = matale(20,3,4)
A =
    16    13    13    10
    10    16     7    14
     4     0    16     8
r =
     3
>> [A,r] = matale(20,3)
A =
    12     0    18
     5    14     9
     3     8     8
r =
     3
>> A = matale(20,3)
A =
     8     7     2
    17    16     4
     1     0     3
>>
```

Un point important concerne la gestion des variables entre le programme principal (ou le workspace) et les fonctions de l'utilisateur. Toutes les variables définies à l'intérieur d'une fonction sont des variables locales à cette fonction. La communication avec des variables du programme principal (ou du « workspace ») ou avec des variables d'autres fonctions se fait uniquement par les variables d'entrée et sortie de la fonction. Une alternative existe toutefois : il est possible de déclarer certaines variables comme des *variables globales*. Une variable globale peut être partagée entre un programme principal et plusieurs fonctions sans qu'il soit besoin de la spécifier parmi les variables d'entrée-sortie des différentes fonctions. On déclare une variable globale grâce au mot clé `global`. Par exemple pour déclarer la variable `numex` globale on écrit `global numex`. Attention, la déclaration `global numex` doit être reprise dans chaque fonction utilisant `numex` comme variable.

5.2 Opérateurs de comparaison et opérateurs logiques

Les opérateurs de comparaison sont :

`==`	:	égal à (`x == y`)
`>`	:	strictement plus grand que (`x >y`)
`<`	:	strictement plus petit que (`x <y`)
`> =`	:	plus grand ou égal à (`x` $\geq$ `y`)
`< =`	:	plus petit ou égal à (`x` $\leq$ `y`)
`~=`	:	différent de (`x ~= y`)

Les opérateurs logiques sont :

`&`	:	et (`x & y`)
`|`	:	ou (`x | y`)
`~`	:	non (`~ x`)

Les opérateurs de comparaison et les opérateurs logiques sont utilisés essentiellement dans les instructions de contrôle, voir le paragraphe 5.3.

5.3 Instructions de contrôle

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

5.3.1 Boucle FOR : parcours d'un intervalle

Une première possibilité pour exécuter une séquence d'instructions de manière répétée consiste à effectuer une boucle pour les valeurs d'un indice, incrémenté à chaque itération, variant entre deux bornes données. Ce processus est mis en œuvre par la « boucle for ».

Syntaxe :

for `indice` = *borne_inf : borne_sup*

séquence d'instructions

end

où

- `indice` est une variable appelée l'indice de la boucle ;
- *borne_inf* et *borne_sup* sont deux constantes réelles (appelées paramètres de la boucle) ;
- *séquence d'instructions* est le traitement à effectuer pour les valeurs d'indices variant entre *borne_inf* et *borne_sup* avec un incrément de 1. On parle du corps de la boucle.

Interprétation :

Si *borne_inf* est plus petit ou égal à *borne_sup*, le traitement *séquence d'instructions* est exécuté ($borne\_sup - borne\_inf + 1$) fois, pour les valeurs de la variable *indice* égales à $borne\_inf, borne\_inf + 1, \cdots, borne\_sup - 1, borne\_sup$. Si *borne_inf* est strictement plus grand que *borne_sup*, on passe à l'instruction qui suit immédiatement l'instruction de fin de boucle (*end*).

Remarque :

L'indice de boucle ne prend pas nécessairement des valeurs entières. D'autre part il n'est pas nécessaire que l'indice de la boucle apparaisse dans le corps de la boucle ; par contre il est interdit de modifier sa valeur s'il apparaît. Il est possible d'imbriquer des boucles mais elles ne doivent pas se recouvrir. On peut utiliser un incrément (pas) autre que 1 (valeur par défaut). La syntaxe est alors *borne_inf* : *pas* : *borne_sup*. Le pas peut être négatif. Attention a bien gérer la borne supérieure ! Voici un exemple (idiot) venant illustrer les possibilités de variations de l'indice de la boucle

```
>> for r=1.1:-0.1:0.75
     disp(['r = ', num2str(r)]);
   end
r = 1.1
r = 1
r = 0.9
r = 0.8
>>
```

Voici un exemple d'utilisation d'une boucle pour calculer $n!$[3]

```
>> n = 4;
>> nfac = 1;
>> for k = 1:n
     nfac = nfac*k;
   end
>> nfac
nfac =
    24
>>
```

5.3.2 Boucle WHILE : tant que . . . faire

Une seconde possibilité pour exécuter une séquence d'instructions de manière répétée consiste à effectuer une boucle tant qu'une condition reste vérifiée. On arrête de boucler dès que cette condition n'est plus satisfaite. Ce processus est mis en œuvre par la « boucle while ».

Syntaxe :

`while` *expression logique*

séquence d'instructions

`end`

où

- *expression logique* est une expression dont le résultat peut être vrai ou faux ;
- *séquence d'instructions* est le traitement à effectuer tant que *expression logique* est vraie.

Interprétation :

Tant que *expression logique* est vraie le traitement *séquence d'instructions* est exécuté sous forme d'une boucle. Lorsque *expression logique* devient faux, on passe à l'instruction qui suit immédiatement l'instruction de fin de boucle (*`end`*).

Remarque :

expression logique est en général le résultat d'un test (par exemple `i < Imax`) ou le résultat d'une fonction logique (par exemple `all(x)`). Il est impératif que le traitement de la *séquence d'instructions* agisse sur le résultat de *expression logique* sans quoi on boucle indéfiniment `(-:`.

Voici comment calculer $n!$[3]avec une boucle while :

```
>> n = 4;
>> k = 1; nfac = 1;
```

```
>> while k <= n
     nfac = nfac*k;
     k = k+1;
   end
>> nfac
nfac =
    24
>>
```

5.3.3 L'instruction conditionnée IF

On a parfois besoin d'exécuter une séquence d'instructions seulement dans le cas où une condition donnée est vérifiée au préalable. Différentes formes d'instruction conditionnée existent sous MATLAB.

L'instruction conditionnée la plus simple a la forme suivante :

Syntaxe :

`if` expression logique

séquence d'instructions

`end`

où

- *expression logique* est une expression dont le résultat peut être vrai ou faux ;
- *séquence d'instructions* est le traitement à effectuer si *expression logique* est vraie.

Interprétation :

la *séquence d'instructions* n'est exécutée que si le résultat de l'évaluation de l'*expression logique* est vraie (c'est-à-dire vaut 1). Dans le cas contraire on exécute l'instruction qui suit le mot clé *`end`*. Dans le cas où l'*expression logique* est vraie, après exécution de la *séquence d'instructions* on reprend le programme à l'instruction qui suit le mot clé *`end`*.

Remarque :

Contrairement à certains langages de programmation, il n'y a pas de mot clé « then » dans cette instruction conditionnée. Notez également que la marque de fin de bloc conditionné est le mot clé `end` et non pas « endif ».

Il existe une séquence conditionnée sous forme d'alternatives :

Syntaxe :

`if` expression logique

séquence d'instructions 1

`else`

séquence d'instructions 2

[3] Le lecteur attentif sait calculer $n!$ plus simplement ...Comme par exemple en exécutant `prod([1 :n])`

```
end
```

où

- *expression logique* est une expression dont le résultat peut être vrai ou faux ;
- *séquence d'instructions 1* est la séquence d'instructions à exécuter dans le cas où *expression logique* est vraie et *séquence d'instructions 2* est la séquence d'instructions à exécuter dans le cas où *expression logique* est faux.

Interprétation :

Si *expression logique* est vraie la *séquence d'instructions 1* est exécutée, sinon c'est la *séquence d'instructions 2* qui est exécutée. Le déroulement du programme reprend ensuite à la première instruction suivant le mot clé **`end`**.

Il est bien entendu possible d'imbriquer des séquences d'instructions conditionnées (au sens où la séquence d'instruction conditionnée contient des séquences d'instructions conditionnée). Pour une meilleure lisibilité, il est recommandé d'utiliser des indentations afin de mettre en évidence l'imbrication des séquences d'instructions conditionnées.

Il est possible d'effectuer un choix en cascade :

Syntaxe :

```
if expression logique 1
    séquence d'instructions 1
elseif expression logique 2
    séquence d'instructions 2
        ⋮
elseif expression logique N
    séquence d'instructions N
else
    séquence d'instructions par défaut
end
```

Interprétation :

Si *expression logique 1* est vraie la *séquence d'instructions 1* est exécutée et le programme reprend ensuite à la première instruction suivant le mot clé **`end`**, sinon si *expression logique 2* est vraie la *séquence d'instructions 2* est exécutée et le programme reprend ensuite à la première instruction suivant le mot clé **`end`**, *etc.* Si aucune des expressions logiques 1 à N n'est vraie alors *séquence d'instructions par défaut* est exécutée.

Remarque :

Attention à ne pas laisser d'espace entre else et if ; le mot clé est **`elseif`**.

On utilise fréquemment un choix en cascade lors d'initialisation de données. Par exemple, on initialise une matrice `A` en fonction de la valeur d'une variable `numex` (numéro d'exemple) de la manière suivante :

```
if numex == 1
 A = ones(n);
elseif numex == 2
 A = magic(n);
elseif numex == 3 | numex == 4
 A = rand(n);
else
 error('numero d''exemple non prevu ...');
end
```

5.3.4 Choix ventilé, l'instruction `switch`

Une alternative à l'utilisation d'une séquence d'instructions conditionnées pour effectuer un choix en cascade existe. Il s'agit de l'instruction `switch`.

Syntaxe :

switch var

case cst_1 ,

séquence d'instructions 1

case cst_2 ,

séquence d'instructions 2

⋮

case cst_N ,

séquence d'instructions N

otherwise

séquence d'instructions par défaut

end

où

- `var` est une variable numérique ou une variable chaîne de caractères ;
- `cst_1`, ..., `cst_N`, sont des constantes numérique ou des constantes chaîne de caractères ;
- *séquence d'instructions i* est la séquence d'instructions à exécuter si `var==cst_i`.

Interprétation :

Si la variable `var` est égale à l'une des constantes `cst_1`, ..., `cst_N`, (par exemple `cst_i`) alors la séquence d'instructions correspondante (ici *séquence d'instructions i*) est exécutée. Le programme reprend ensuite à la première instruction suivant le mot-clé *end*. Si la

variable `var` n'est égale à aucune des constantes la *séquence d'instructions par défaut* est exécutée.

Remarque :

La variable `var` doit bien entendu être du même type que les constantes `cst_1`, ..., `cst_N`. Il n'est pas nécessaire de prévoir un cas par défaut (bien que cela soit préférable). S'il n'y a pas de cas par défaut et si la variable `var` n'est égale à aucune des constantes `cst_1`, ..., `cst_N`, alors le programme continue à la première instruction suivant le mot-clé ***end***.
Si la séquence d'instructions à exécuter est la même pour différents cas il est possible de les regrouper. La syntaxe est alors,

```
case { cst_k , cst_l , ...}
```

séquence d'instructions commune

Reprenons l'exemple où l'on souhaite initialiser une matrice `A` en fonction de la valeur prise par une variable numérique `numex` (numéro d'exemple). En utilisant un choix ventilé on obtient :

```
switch numex
  case 1,
    A = ones(n)
  case 2,
    A = magic(n);
  case {3,4},
    A = rand(n);
  otherwise
    error('numero d''exemple non prevu ...');
end
```

Voici un exemple de choix ventilé portant sur une variable de type chaîne de caractères.

```
rep = input('Votre reponse (oui, non, chepas) :');

switch rep
  case {'oui','o'},
    disp('bravo ...');
  case {'non','n'}
    disp('perdu ...');
  case 'chepas'
    disp('c''est pourtant facile ...');
end
```

5.3.5 Interruption d'une boucle de contrôle

Il est possible de provoquer une sortie prématurée d'une boucle de contrôle. L'instruction `break` permet de sortir d'une *boucle for* ou d'une *boucle while*. L'exécution se poursuit alors séquentiellement à partir de l'instruction suivant le mot clé ***end*** fermant la boucle. En cas de boucles imbriquées, on interrompt seulement l'exécution de la boucle intérieure contenant l'instruction `break`.

L'instruction `return` provoque un retour au programme appelant (ou au clavier). Les instructions suivant le `return` ne sont donc pas exécutées. L'instruction `return` est souvent utilisée conjointement avec une instruction conditionnée par exemple pour tester dans le corps d'une fonction si les paramètres d'entrée ont les valeurs attendues.

L'instruction `error` permet d'arrêter un programme et d'afficher un message d'erreur. La syntaxe est `error(' message d''erreur ')`. L'instruction `warning` permet d'afficher un message de mise en garde sans suspendre l'exécution du programme. La syntaxe est `warning(' message de mise en garde ')`. Il est possible d'indiquer à MATLAB de ne pas afficher les messages de mise en garde d'un programme en tapant `warning off` dans la fenêtre de commandes. On rétablit l'affichage en tapant `warning on`.

On peut ainsi améliorer la fonction `matale` de la manière suivante :

```
function [A,rang] = matale(T,m,n)
% Construit une matrice A de m lignes et n colonnes ayant des elements
% entiers generes de maniere aleatoire entre 0 et T.
% Calcule le rang de la matrice si l'appel est effectue avec 2 arguments
% de sortie.
% Si la matrice est carree, le parametre n peut etre omis.
%
% Appels:
% [A,r] = Matale(T,m,n)
% [A,r] = Matale(T,m)
%     A = Matale(T,m,n)
%     A = Matale(T,m)

% si la fonction est appelee avec un nombre d'arguments d'entree
% different de 2 ou 3, on arrete tout ...
if nargin ~= 2 &  nargin ~= 3,
   error(' La fonction matale doit avoir 2 ou 3 arguments d''entree ');
end

if nargin == 2
  A = fix(T*rand(m));
else
  A = fix(T*rand(m,n));
end

if nargout == 2
  rang = rank(A);
  if nargin == 2,
    rangx = m;
  else
```

```
    rangx = min([m,n]);
  end
  if rang ~= rangx, warning(' Le rang n''est pas maximum '); end;
end
```

On obtient alors les messages suivants :

```
>> A = matale(3);
??? Error using ==> matale
 La fonction matale doit avoir 2 ou 3 arguments d'entree

>> A = matale(20,3)
A =
     8    18     8
    12    14    18
    15     3    18
>> [A,r] = matale(20,3)

Warning:  Le rang n'est pas maximum
> In /home0/maths/balac/DOCMATLAB/matale.m at line 34
A =
     1     4     3
    10    15    11
     3    12     9
r =
     2
>>
```

La commande `pause` permet d'interrompre l'exécution du programme. L'exécution normale reprend dès que l'utilisateur enfonce une touche du clavier. L'instruction `pause(n)` suspend l'exécution du programme pendant `n` secondes.

5.4 Un exemple complet

Une technique de construction de carrés magiques d'ordre impair a été proposée par Manuel Moschopoulos au début du XIV^{e}siècle. Cette technique est décrite dans [1].

Notons $l(x)$ le numéro de la ligne et $c(x)$ le numéro de la colonne du carré sur lequel se trouve l'entier x. Partant d'un carré d'ordre impair $n = 2k + 1$, la technique de Moschopoulos peut se formuler comme suit :

- Initialisation de l'algorithme en plaçant l'unité dans la case immédiatement au dessous de la case centrale, autrement dit à l'intersection de la $(k+1)^{\text{e}}$colonne et de la $(k+2)^{\text{e}}$ligne :

$$\begin{aligned} l(1) &= k+2; \\ c(1) &= k+1. \end{aligned}$$

- Connaissant la position $(l(x), c(x))$ de l'entier x, on place l'entier $x+1$ suivant les règles suivantes :

si x n'est pas un multiple de n, alors

$$\begin{aligned} l(x+1) &= 1 + l(x) \quad (\text{ modulo } n), \\ c(x+1) &= 1 + c(x) \quad (\text{ modulo } n); \end{aligned}$$

si x est multiple de n, alors

$$\begin{aligned} l(x+1) &= 2 + l(x) \quad (\text{ modulo } n), \\ c(x+1) &= c(x) \quad (\text{ modulo } n). \end{aligned}$$

Dans ces règles pour la prise du modulo, le système de résidus que l'on considère est $1, 2, \ldots, n$ et non pas $0, 1, \ldots, n-1$.

La fonction `magik` met en œuvre la méthode de Moschopoulos.

```
function M = magik(n)
%
% Calcule le carre magique d'ordre n selon la methode
% de Moschopoulous.
%
% Entree:
%  n : ordre du carre (entier impair)
% Sortie:
%  M : le carre magique
%

if rem(n,2) == 0,
  msg = ['la methode de Moschopoulous ne construit que des carres' ...
       ,' d''ordre impair'];
  error(msg)
end

k = (n-1)/2;
l(1) = k+2;
c(1) = k+1;
M(l(1),c(1)) = 1;

for x = 2:n.^2
  [l(x),c(x)] = pos(x-1,l(x-1),c(x-1),n);
  M(l(x),c(x)) = x;
% ou plus simplement M(pos(x,l(x-1),c(x-1))) = x;
end
```

```
function [ly,cy] = pos(x,lx,cx,n)
%
% Retourne la position (indice de ligne ly et indice de colonne cy)
% dans le carre magique d'ordre nde l'entier y = x+1 selon la
```

```
% regle de Moschopoulous.
%
% Entree:
%  n : ordre du carre (entier impair)
%  x : l'entier considere
% lx : indice de ligne de l'entier x dans le carre magique
% cx : indice de colonne de l'entier x dans le carre magique
%
% Sortie:
%  ly : indice de ligne de l'entier y=x+1 dans le carre magique
%  cy : indice de colonne de l'entier y=x+1 dans le carre magique
%

if rem(x,n) == 0
  ly = modulo(2+lx,n);
  cy = modulo(cx,n);
else
  ly = modulo(1+lx,n);
  cy = modulo(1+cx,n);
end
```

Voici quelques exemples d'utilisation de la fonction `magik`. On vérifie que la somme des éléments des différentes lignes ainsi que la somme des éléments des différentes colonnes sont bien constantes. Pour calculer la somme des éléments diagonaux c'est un peu plus compliqué. On remarque que le carré magique construit diffère du carré magique retourné par la fonction MATLAB incorporée `magic`.

```
>> magik(4)
??? Error using ==> magik
la methode de Moschopoulous ne construit que des carres d'ordre impair
>> magik(5)
ans =
    11    24     7    20     3
     4    12    25     8    16
    17     5    13    21     9
    10    18     1    14    22
    23     6    19     2    15
>> sum(magik(5),1)
ans =
    65    65    65    65    65
>>  sum(magik(5),2)
ans =
    65
    65
    65
    65
    65
>> magic(5)
```

```
ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

6 Graphisme

6.1 Gestion des fenêtres graphiques

Une instruction graphique ouvre une fenêtre dans laquelle est affiché le résultat de cette commande. Par défaut, une nouvelle instruction graphique sera affichée dans la même fenêtre et écrasera la figure précédente. On peut ouvrir une nouvelle fenêtre graphique par la commande `figure`. Chaque fenêtre se voit affecter un numéro. Ce numéro est visible dans le bandeau de la fenêtre sous forme d'un titre. Le résultat d'une instruction graphique est par défaut affiché dans la dernière fenêtre graphique ouverte qui est la fenêtre graphique active. On rend active une fenêtre graphique précédemment ouverte en exécutant la commande `figure(`n`)`, où n désigne le numéro de la figure. La commande `close` permet de fermer la fenêtre graphique active. On ferme une fenêtre graphique précédemment ouverte en exécutant la commande `close(`n`)`, où n désigne le numéro de la figure. Il est également possible de fermer toutes les fenêtres graphiques en tapant `close all`.

La figure 4 représente la fenêtre graphique numéro 1 où est affiché le graphe de la fonction cosinus entre 0 et 2π, résultat de la commande `fplot('cos',[0 2*pi])`. L'apparence de la fenêtre graphique peut varier légèrement suivant le système informatique utilisé.

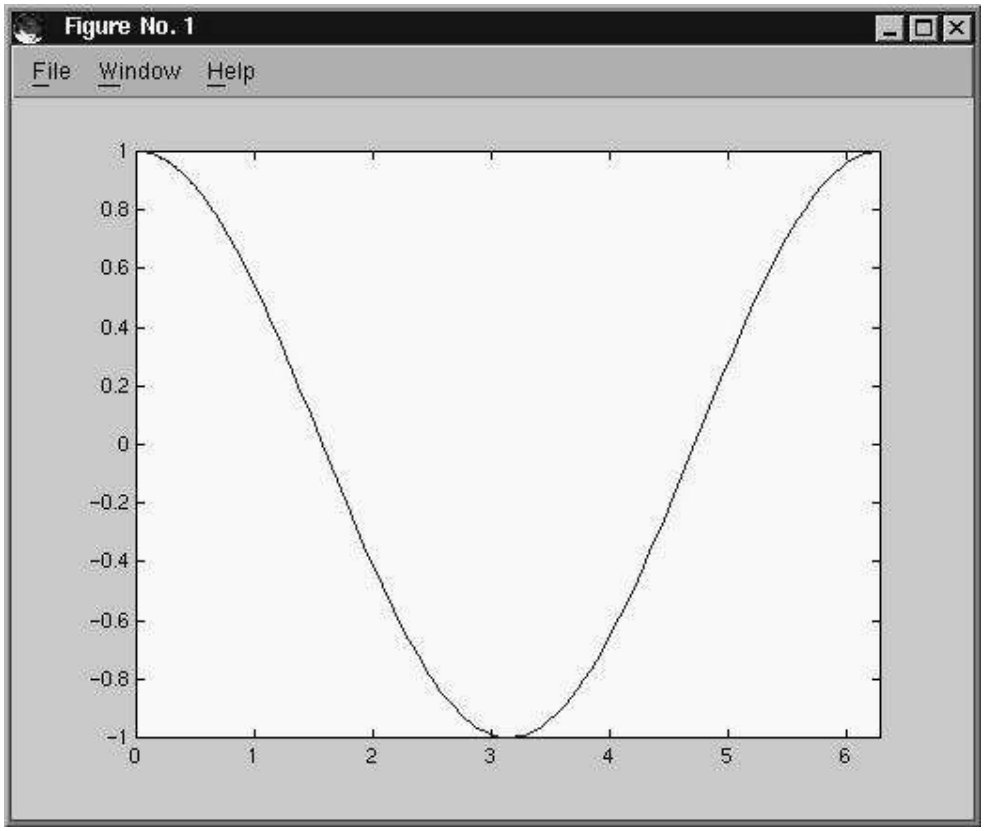

FIG. 4 – Fenêtre graphique et résultat de la commande `fplot('cos',[0 2*pi])`.

6.2 Graphisme 2D

6.2.1 Tracer le graphe d'une fonction ; la commande `fplot`

La commande `fplot` permet de tracer le graphe d'une fonction sur un intervalle donné. La syntaxe est :

```
fplot('nomf', [xmin , xmax])
```

où

- `nomf` est soit le nom d'une fonction MATLAB incorporée, soit une expression définissant une fonction de la variable `x`, soit le nom d'une fonction utilisateur.
- `[xmin , xmax]` est l'intervalle pour lequel est tracé le graphe de la fonction.

Illustrons par des exemples les trois façons d'utiliser la commande `fplot`. On obtient le graphe de la fonction incorporée sinus entre -2π et 2π par l'instruction :

```
fplot('sin',[-2*pi 2*pi]).
```

Pour tracer le graphe de la fonction $h(x) = x\ \sin(x)$ entre -2π et 2π, on peut définir la fonction utilisateur `h` dans le fichier `h.m` de la manière suivante[4] :

```
function y=h(x)

y=x.*sin(x);
```

On obtient alors le graphe de la fonction h par l'instruction :

```
fplot('h', [-2*pi 2*pi]).
```

L'autre façon de procéder est d'exécuter l'instruction[5] :

```
fplot('x*sin(x)', [-2*pi 2*pi]).
```

Dans les deux cas on obtient le dessin représenté à la figure 5.

Il est possible de tracer plusieurs fonctions sur la même figure. Il faut pour cela utiliser la commande `fplot` de la manière suivante :

`fplot('[nom_f1 , nom_f2 , nom_f3]', [x_min , x_max])` où `nomf_f1, nom_f2, nom_f3` est soit le nom d'une fonction MATLAB incorporée, soit une expression définissant une fonction de la variable `x`, soit le nom d'une fonction utilisateur.

Il est également possible de gérer les bornes des valeurs en ordonnées. Pour limiter le graphe aux ordonnées comprises entre les valeurs `y_min` et `y_max` on passera comme second argument de la commande `fplot` le tableau `[x_min , x_max , y_min , y_max ]`.

Voici un exemple dont le résultat est affiché à la figure 6 :

```
>> fplot('[sin(x)/x , cos(x)/x]', [-5, 5, -1, 1])
>>
```

[4] attention de bien lire `x.*sin(x)` et non pas `x*sin(x)`

[5] là on a le choix entre écrire `x.*sin(x)` ou `x*sin(x)`

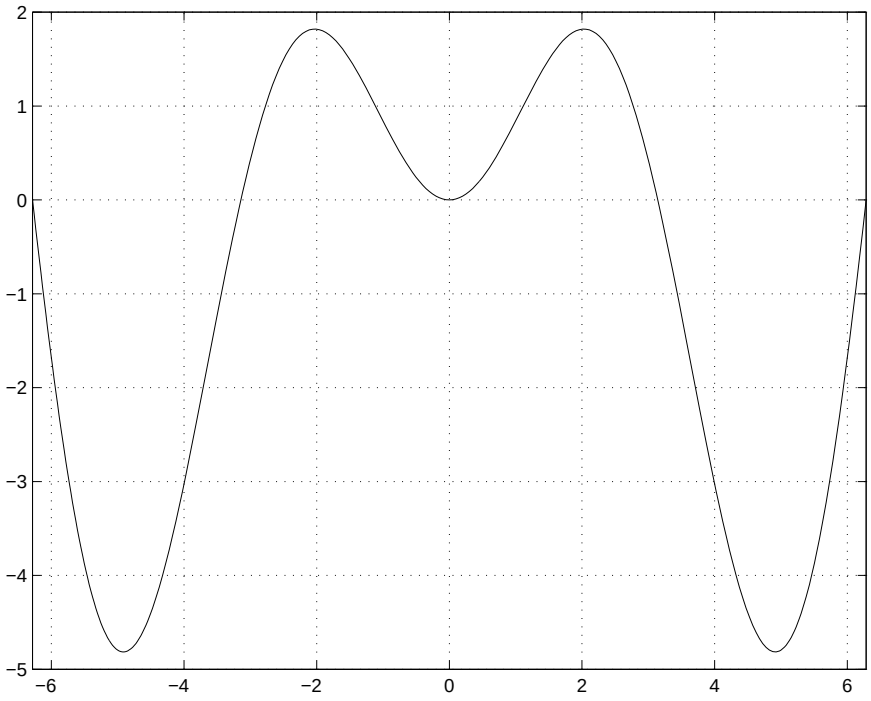

FIG. 5 – Graphe de la fonction $h(x) = x \sin x$ entre -2π et 2π.

On comprend très vite l'intérêt de gérer les bornes des valeurs en ordonnées si l'on exécute la commande `fplot('cos(x)/x', [-5, 5])` pour tracer le graphe de la fonction $\cos(x)/x$ entre -5 et 5. La figure, qui n'est pas vraiment esthétique, n'est pas reproduite ici `(-:`.

6.2.2 La commande `plot`

La commande `plot` permet de tracer un ensemble de points de coordonnées (x_i, y_i) $i = 1, \cdots, N$. La syntaxe est `plot(x,y)` où `x` est le vecteur contenant les valeurs x_i en abscisse et `y` est le vecteur contenant les valeurs y_i en ordonnée. Bien entendu les vecteurs `x` et `y` doivent être de même dimension mais il peut s'agir de vecteurs lignes ou colonnes. Par défaut, les points (x_i, y_i) sont reliés entre eux par des segments de droites.

Voici par exemple une autre façon de tracer le graphe de la fonction $h(x) = x\ \sin(x)$ entre -2π et 2π,

```
>> x=[-2*pi:0.01:2*pi]; y = x.*sin(x);
>> plot(x,y)
>>
```

Essayez aussi

```
>> x=[-2*pi:1:2*pi]; y = x.*sin(x);
>> plot(x,y)
>>
```

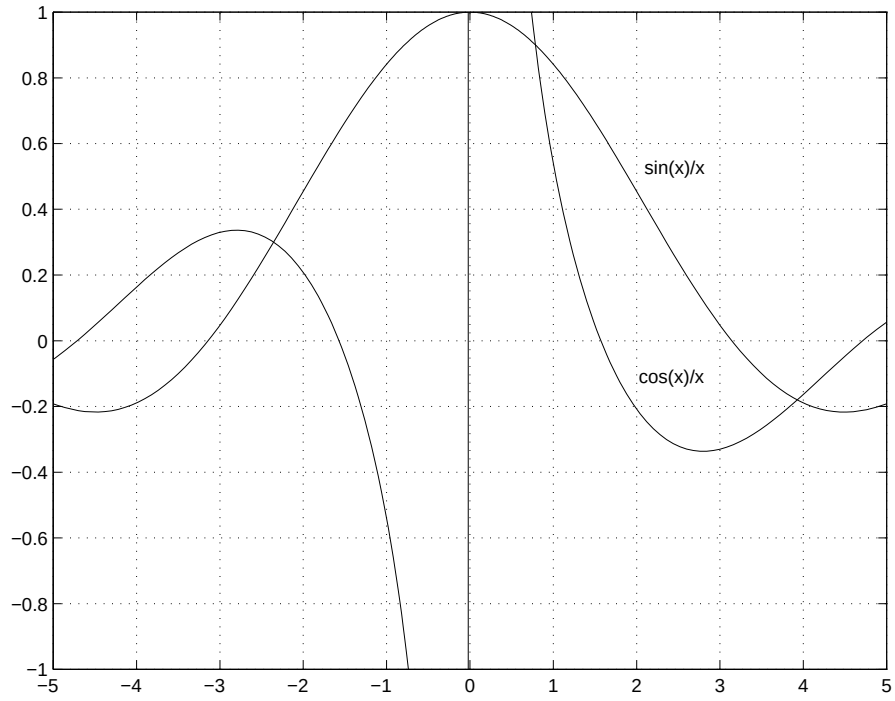

FIG. 6 – Graphe des fonctions $\sin(x)/x$ et $\cos(x)/x$ entre -5 et 5.

Dans cet exemple on a défini un vecteur `x` de valeurs équi-réparties entre -2π et 2π (avec un pas de 0.01 dans le premier cas et de 1 dans le deuxième cas) et on a calculé l'image par la fonction h de ces valeurs (vecteur `y`). On affiche les points de coordonnées (`x(i)`, `y(i)`).

On peut spécifier à MATLAB quelle doit être la couleur d'une courbe, quel doit être le style de trait et/ou quel doit être le symbole à chaque point (x_i, y_i). Pour cela on donne un troisième paramètre d'entrée à la commande `plot` qui est une chaîne de 3 caractères de la forme `'cst'` avec `c` désignant la couleur du trait, `s` le symbole du point et `t` le style de trait. Les possibilités sont les suivantes :

y	jaune	.	point	-	trait plein
m	magenta	o	cercle	:	pointillé court
c	cyan	x	marque x	--	pointillé long
r	rouge	+	plus	-.	pointillé mixte
g	vert	*	étoile		
b	bleu	s	carré		
w	blanc	d	losange		
k	noir	v	triangle (bas)		
		^	triangle (haut)		
		<	triangle (gauche)		
		>	triangle (droit)		
		p	pentagone		
		h	hexagone		

Les valeurs par défaut sont `c = b`, `s = .` et `t = -` ce qui correspond à un trait plein bleu reliant les points entre eux. Il n'est pas obligatoire de spécifier chacun des trois caractères. On peut se contenter d'en spécifier un ou deux. Les autres seront les valeurs par défaut.

La commande `grid` permet d'obtenir un quadrillage de la figure, voir par exemple la figure 4.

Il est possible de tracer plusieurs courbes sur la même figure en spécifiant plusieurs tableaux `x1`, `y1`, `x2`, `y2`, $\cdots$, comme paramètres de la commande `plot`. Si l'on souhaite que les courbes aient une apparence différente, on utilisera des options de couleurs et/ou de styles de traits distincts après chaque couple de vecteurs `x`, `y`.

Voici un exemple dont le résultat est affiché à la figure 7. On trace sur l'intervalle $[-5, 5]$ la fonction $x^2 \cos x$ en trait plein bleu et la fonction $x \cos x$ en trait pointillé rouge.

```
>> x = [-5:0.01:5];
>> y = x.^2.*cos(x); z = x.*cos(x);
>> plot(x,y,'b-',x,z,'r:');
>>
```

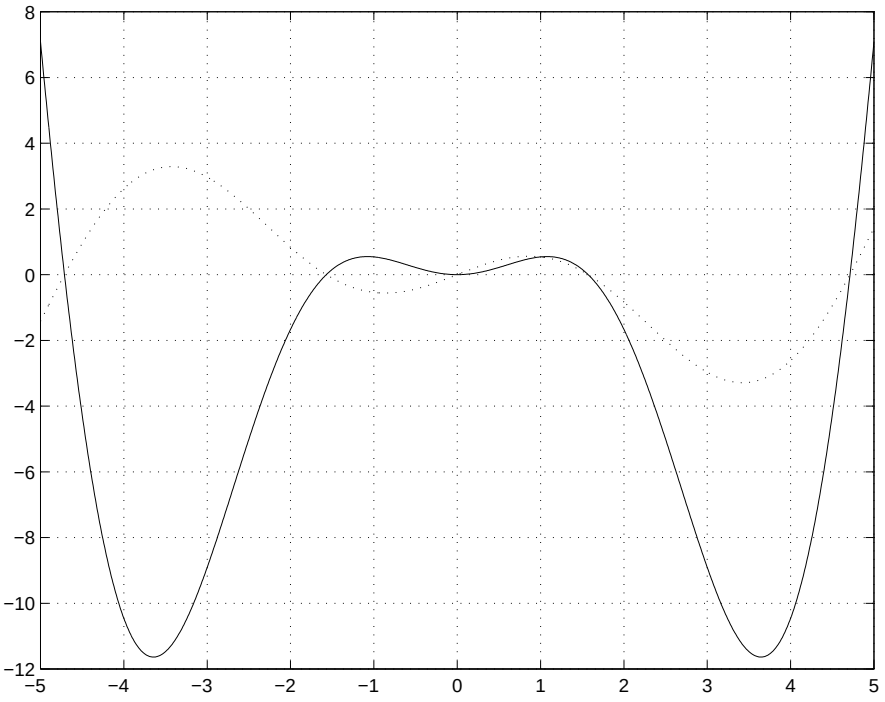

FIG. 7 – Résultat de la commande `plot(x,y,'b-',x,z,'r :')`.

Voici un autre exemple amusant[6].

```
>> N=200;
>> x = rand(1,N); y = rand(1,N);
>> plot(x,y,'bd'), whitebg
>>
```

[6]La commande `whitebg` permet de passer la fenêtre graphique en inverse vidéo

6.2.3 La commande `loglog`

Si `x` et `y` sont deux vecteurs de même dimension, la commande `loglog(x,y)` permet d'afficher le vecteur `log(x)` contre le vecteur `log(y)`. La commande `loglog` s'utilise de la même manière que la commande `plot`.

Voici un exemple dont le résultat est affiché à la figure 8. Quelle est la pente de la droite ?

```
>> x = [1:10:1000]; y = x.^3;
>> loglog(x,y)
>>
```

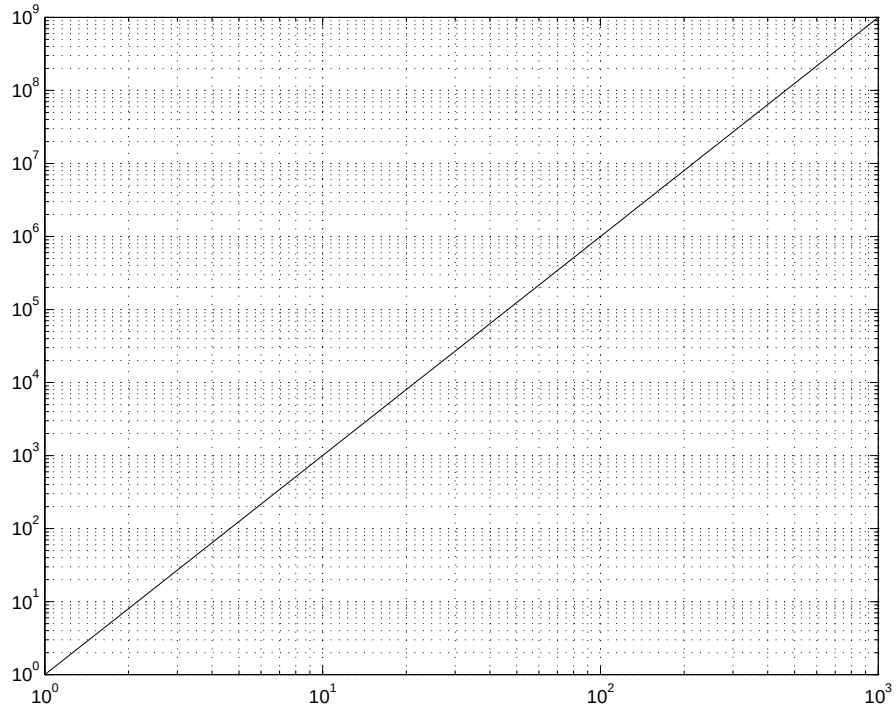

FIG. 8 – Résultat de la commande `loglog(x,y)`.

6.3 Améliorer la lisibilité d'une figure

6.3.1 Légender une figure

Il est recommandé de mettre une légende à une figure. La commande `xlabel` permet de mettre un texte en légende sous l'axe des abscisses. La syntaxe est

```
xlabel(' légende ')
```

pour obtenir le mot `légende` en légende. La commande `ylabel` fait de même pour l'axe des ordonnées.

La commande `title` permet de donner un titre à la figure. La syntaxe est

```
title(' le titre ')
```

pour obtenir comme titre `le titre`.

On peut aussi écrire un texte donné à une position précise sur la figure grâce à la commande `text`. La syntaxe est `text(posx,posy,' un texte ')` où `posx` et `posy` sont les coordonnées du point (dans le système associé au déssin) où doit débuter l'écriture du texte `un texte`. La commande `gtext` permet quant à elle de placer le texte à une position choisie sur la figure à l'aide de la souris. La syntaxe est `gtext(' un texte ')`. Une mire, que l'on déplace en utilisant la souris, apparaît. Il suffit d'un « clic-souris » pour que le texte apparaisse à la position sélectionnée.

Il est possible avec ces commandes d'afficher une valeur contenue dans une variable au milieu de texte. Pour cela on construit un tableau de type chaîne de caractères en convertissant la valeur contenue dans la variable en une chaîne de caractères grâce à la commande `num2str`. Par exemple, supposons que la variable `numex` contienne le numéro de l'exemple traité, disons 5. On obtiendra pour titre de la figure `Exemple numero 5` par l'instruction : `title(['Exemple numero ', num2str(numex)])`.

L'exemple suivant dont le résultat est affiché à la figure 9 illustre l'utilisation des différentes commandes permettant de légender une figure.

```
>> P = 5;
>> t = [0:.01:2];
>> c = 12*exp(-2*t) - 8*exp(-6*t);
>> plot(t,c); grid
>> xlabel('temps en minutes')
>> ylabel('concentation en gramme par litre')
>> title(['evolution de la concentration du produit ', num2str(P), ...
    ' au cours du temps '])
>> gtext('concentration maximale')
>>
```

6.3.2 Afficher plusieurs courbes dans une même fenêtre

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` seront 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`. Le résultat est présenté à la figure 10.

```
>> e = exp(1);
>> figure
>> hold on
>> fplot('exp',[-1 1])
>> fplot('log',[1/e e])
>> plot([-1:0.01:e],[-1:0.01:e])
>> grid
>> hold off
>>
```

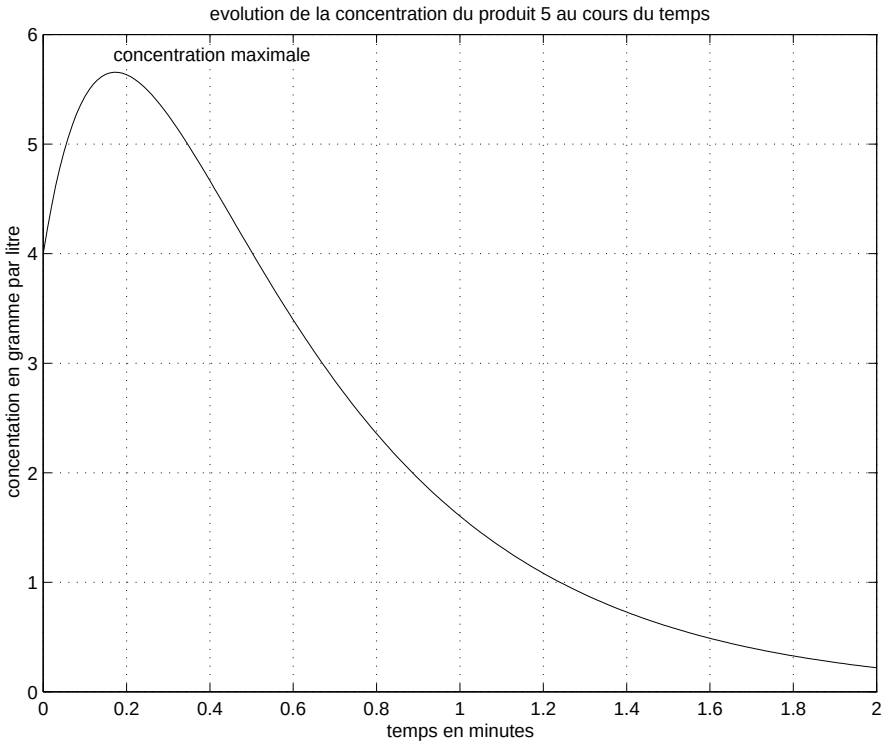

FIG. 9 – Exemple de légendage d'une figure.

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.

Il est possible de décomposer une fenêtre en sous-fenêtres et d'afficher une figure différente sur chacune de ces sous-fenêtres grâce à la commande `subplot`. La syntaxe est

```
subplot(m,n,i)
```

où

- `m` est le nombre de sous-fenêtres verticalement ;
- `n` est le nombre de sous-fenêtres horizontalement ;
- `i` sert à spécifier dans quelle sous-fenêtre doit s'effectuer l'affichage. Les fenêtres sont numérotées de gauche à droite et de haut en bas.

L'exemple suivant dont le résultat apparaît à la figure 11 illustre l'utilisation de la commande `subplot`.

```
>> figure
>> subplot(2,3,1), fplot('cos',[0 4*pi]), title('cosinus'), grid
>> subplot(2,3,2), fplot('sin',[0 4*pi]), title('sinus'), grid
>> subplot(2,3,3), fplot('tan',[-pi/3 pi/3]), title('tangente'), grid
>> subplot(2,3,4), fplot('acos',[-1 1]), title('arc-cosinus'), grid
>> subplot(2,3,5), fplot('asin',[-1 1]), title('arc-sinus'), grid
```

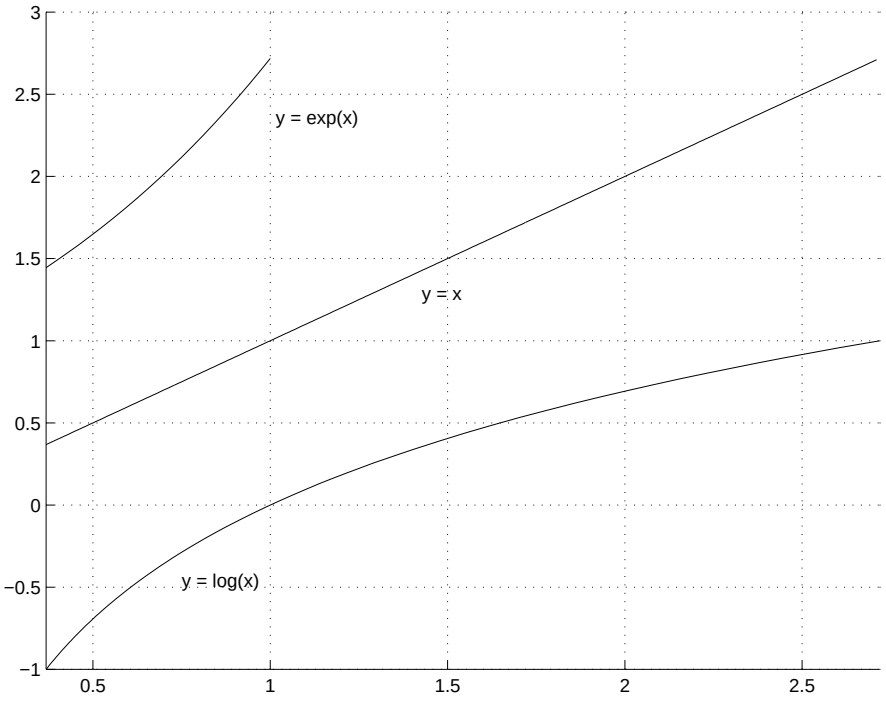

FIG. 10 – Superposition de plusieurs courbes dans la même fenêtre graphique.

```
>> subplot(2,3,6), fplot('atan',[-sqrt(3) sqrt(3)]), title('arc-tangente'), grid
>>
```

6.3.3 Sauvegarder une figure

La commande `print` permet de sauvegarder la figure d'une fenêtre graphique dans un fichier sous divers formats d'images. La syntaxe de la commande `print` est :

```
print -f<num> -d<format> <nomfic>
```

où

- `<num>` désigne le numéro de la fenêtre graphique. Si ce paramètre n'est pas spécifié, c'est la fenêtre active qui est prise en compte.
- `<nomfic>` est le nom du fichier dans lequel est sauvegardée la figure. Si aucune extension de nom n'est donnée, une extension par défaut est ajoutée au nom du fichier en fonction du format choisi (`.ps` pour du PostScript, `.jpg` pour du jpeg, par exemple).
- `<format>` est le format de sauvegarde de la figure. Ces formats sont nombreux. On pourra obtenir la liste complète en tapant `help plot`. Les principaux sont :

`ps`	:	PostScript noir et blanc
`psc`	:	PostScript couleur
`eps`	:	PostScript Encapsulé noir et blanc
`epsc`	:	PostScript Encapsulé couleur
`jpeg`	:	Format d'image JPEG
`tiff`	:	Format d'image TIFF

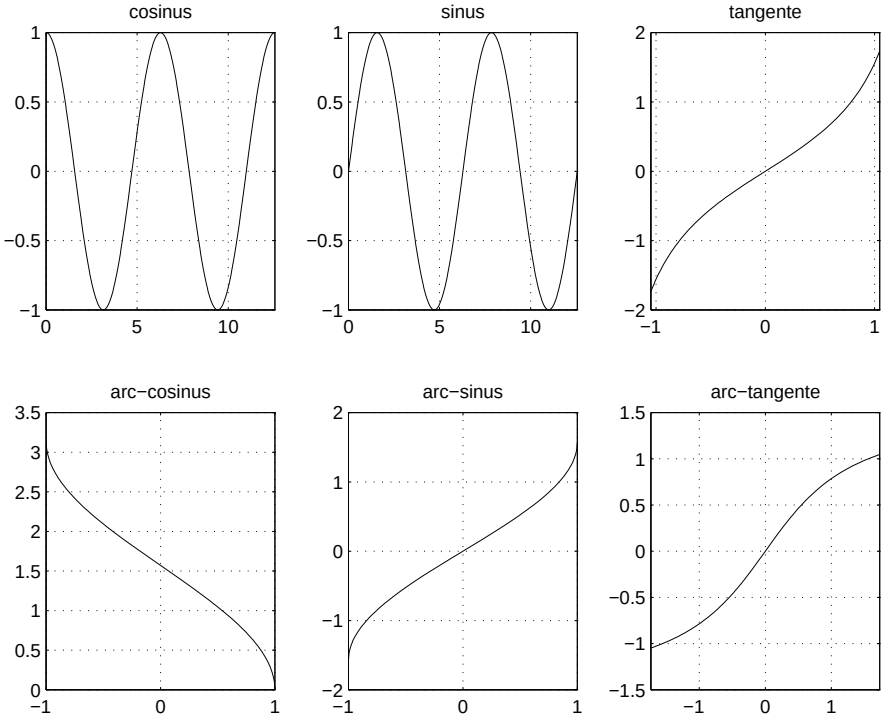

FIG. 11 – Fenêtre graphique décomposée en sous-fenêtres

6.4 Graphisme 3D

6.4.1 Tracer les lignes de niveau d'une fonction de 2 variables

La commande `contour` permet de tracer les lignes de niveau d'une fonction de 2 variables réelles. Cette fonction peut être définie par une expression MATLAB (par exemple `x.*exp(-x.^2-y.^2)`), ou être définie comme une fonction utilisateur. Pour tracer les lignes de niveau de la fonction $g(x, y)$ pour $x \in [x_{min}, x_{max}]$ et $y \in [y_{min}, y_{max}]$ on procède de la manière suivante :

- création d'un maillage, de maille de longueur h, du domaine $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ grâce à la commande `meshgrid`,
  `[X,Y] = meshgrid(x_min :h :x_max, y_min :h :y_max)`.
- Évaluation de la fonction aux nœuds de ce maillage, soit par appel à la fonction utilisateur définissant la fonction,
  `Z = g(X,Y)`
  soit directement en définissant la fonction par une expression MATLAB.
- Affichage des lignes de niveau grâce à la commande `contour`,
  `contour(X,Y,Z)`.

Ainsi pour tracer les lignes de niveau de la fonction $(x, y) \longmapsto xe^{-(x^2+y^2)}$ sur le domaine $[-2, 2] \times [-2, 2]$ en prenant un maillage de maille de longueur $h = 0.2$, on exécute :

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

On peut également écrire une fonction utilisateur `g.m`,

```
function x3 = g(x1,x2)

x3 =  x1.*exp(-x1.^2-x2.^2);
```

et exécuter

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>> Z = g(X,Y);
>> contour(X,Y,Z)
>>
```

Dans les 2 cas on obtient le résultat présenté à la figure 12.

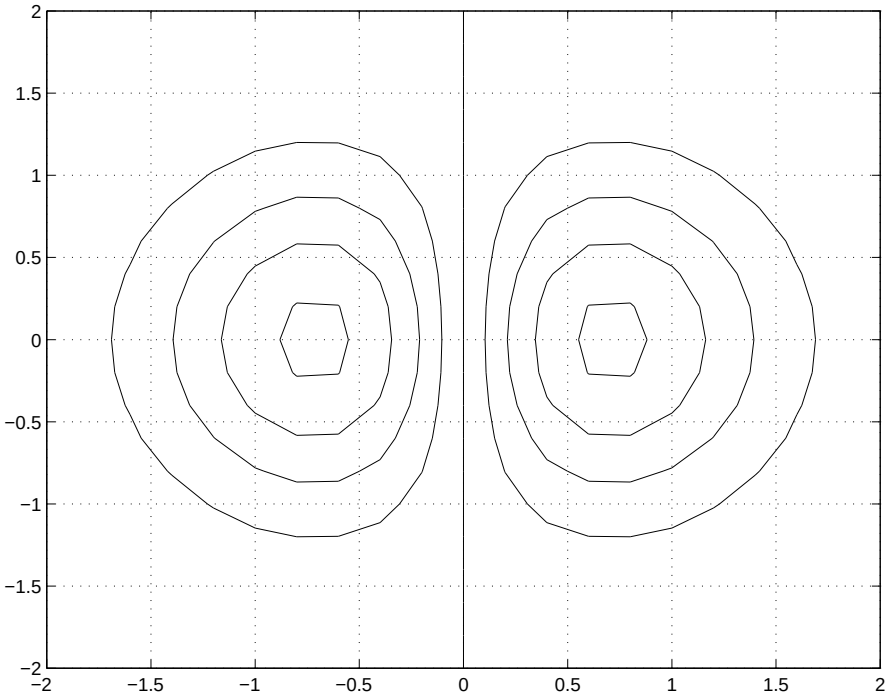

FIG. 12 – Exemple de visualisation des lignes de niveau par la commande `contour`.

Le nombre de lignes de niveau est déterminé de manière automatique à partir des valeurs extrêmes prises par la fonction sur le domaine considéré. Pour imposer le nombre n de lignes de niveau à afficher, il suffit d'appeler la fonction contour avec la valeur n comme quatrième paramètre, `contour(X,Y,Z,n)`.

Il existe deux manières d'afficher les valeurs des lignes de niveau sur la figure. Si l'on souhaite afficher les valeurs pour toutes les lignes de niveau, on utilise la commande `clabel` de la manière suivante :

```
>> [C,h] = contour(X,Y,Z,n)
>> clabel(C,h)
```

Si l'on souhaite afficher uniquement les valeurs de quelques lignes de niveau, on utilise la commande `clabel` de la manière suivante :

```
>> [C,h] = contour(X,Y,Z,n)
>> clabel(C,h,'manual')
```

On peut alors grâce à la souris sélectionner les lignes de niveau pour lesquelles on souhaite afficher la valeur.

Ainsi pour tracer 30 lignes de niveau de la fonction $(x, y) \longmapsto (x-1)^2 + 10\,(x^2 - y)^2$ sur le domaine $[-1, 1] \times [-1, 1]$ et pour choisir à l'aide de la souris les lignes de niveau pour lesquelles l'isovaleur doit être affichée, on exécute :

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>>  Z = (X-1).^2 + 10*(X.^2-Y).^2;
>> [C,h] = contour(X,Y,Z,30);
>> clabel(C,h,'manual')
>>
```

Le résultat est présenté à la figure 13.

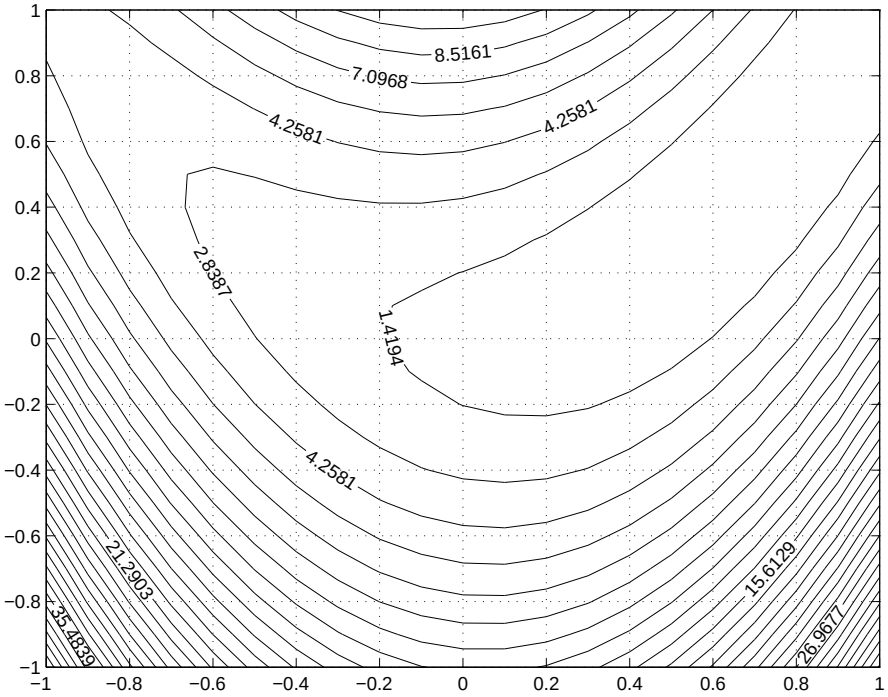

FIG. 13 – Exemple de visualisation des lignes de niveau par la commande `contour`.

Il est possible de modifier la palette des couleurs en utilisant la commande `colormap`. En tapant `help graph3d` dans la fenêtre de contrôle MATLAB, vous obtiendrez toutes les

palettes de couleurs disponibles. A vous de les tester pour obtenir le meilleur effet (-:. La commande `colormap(gray)` permet d'utiliser une palette en dégradé de gris, ce qui est très utile si l'on souhaite une impression de la figure sur une imprimante noir et blanc.

La commande `contourf` s'utilise de la même manière que la commande `contour`. Elle permet d'afficher, en plus des lignes de niveau, un dégradé continu de couleurs qui varie en fonction des valeurs prises par la fonction. La figure 14 présente un exemple d'utilisation de la commande `contourf` obtenu de la manière suivante :

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>> Z = (X-1).^2 + 10*(X.^2-Y).^2;
>> contourf(X,Y,Z,30);
>> colormap(cool);
>>
```

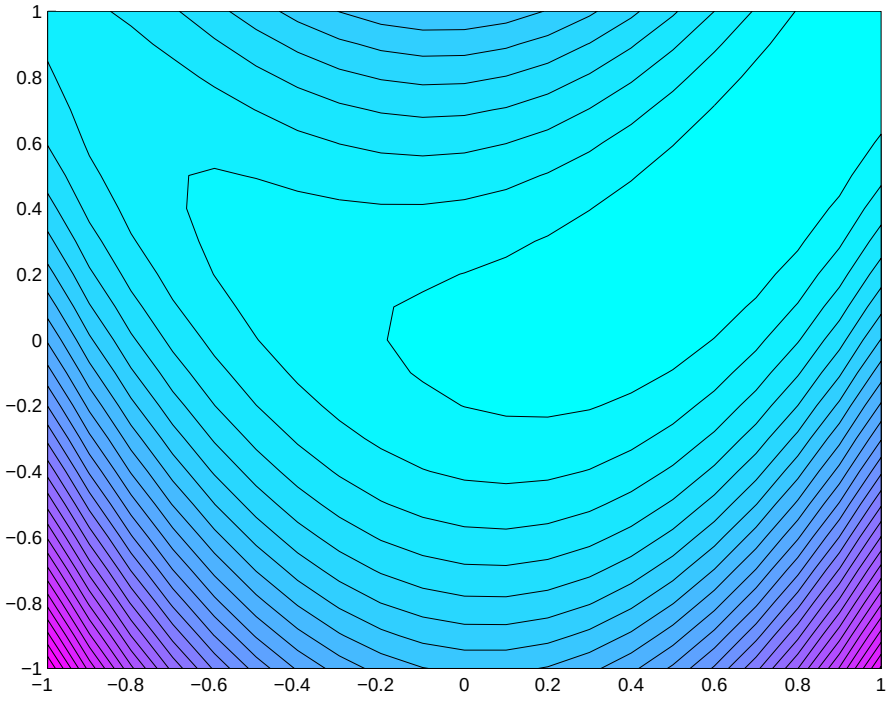

FIG. 14 – Exemple de visualisation des lignes de niveau par la commande `contourf`.

6.4.2 Représenter une surface d'équation $z = g(x, y)$

La commande `mesh` permet de tracer une surface d'équation $z = g(x, y)$. La fonction g peut être définie directement par une expression MATLAB ou être définie comme une fonction utilisateur. Pour tracer la surface d'équation $z = g(x, y)$ pour $x \in [x_{min}, x_{max}]$ et $y \in [y_{min}, y_{max}]$ on procède de la manière suivante :

- création d'un maillage, de maille de longueur h, du domaine $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ grâce à la commande `meshgrid`,
  `[X,Y] = meshgrid(x_min :h :x_max, y_min :h :y_max).`
- Évaluation de la fonction aux nœuds de ce maillage, soit par appel à la fonction utilisateur définissant la fonction,
  `Z = g(X,Y)`
  soit directement en définissant la fonction par d'une expression MATLAB.
- Affichage de la surface grâce à la commande `mesh`,
  `mesh(X,Y,Z).`

Ainsi pour tracer la surface d'équation $z = xe^{-(x^2+y^2)}$ sur le domaine $[-2, 2] \times [-2, 2]$ avec un maillage de maillage de longueur $h = 0.2$, on exécute :

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

Si la fonction est définie comme une fonction utilisateur dans le fichier `g.m`,

```
function x3 = g(x1,x2)

x3 =  x1.*exp(-x1.^2-x2.^2);
```

on exécute :

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>> Z = g(X,Y);
>> contour(X,Y,Z)
>>
```

Dans les 2 cas on obtient le résultat présenté figure 15.

Par défaut les valeurs extrêmes en z sont déterminées automatiquement à partir des extremums de la fonction sur le domaine spécifié. Il est possible de modifier ces valeurs (et également les valeurs extrêmes en abscisses et ordonnées) par la commande `axis` dont la syntaxe est `axis(x_min x_max y_min y_max z_min z_max)`. Si vous n'êtes pas satisfait du résultat, l'instruction `axis auto` restaure la figure originale. La figure 16 montre par exemple le résultat de l'instruction `axis([ -2 2 -2 2 -2 2])`.

Il est également possible de modifier le point de vision grâce à la commande `view`. La commande `view` a deux arguments qui sont l'angle de vision horizontal et l'angle de vision vertical en degré. Par défaut ces angles ont repectivement les valeurs -37.5^o et 30^o. La figure 17 montre par exemple le résultat de l'instruction `view(37.5,30)`.

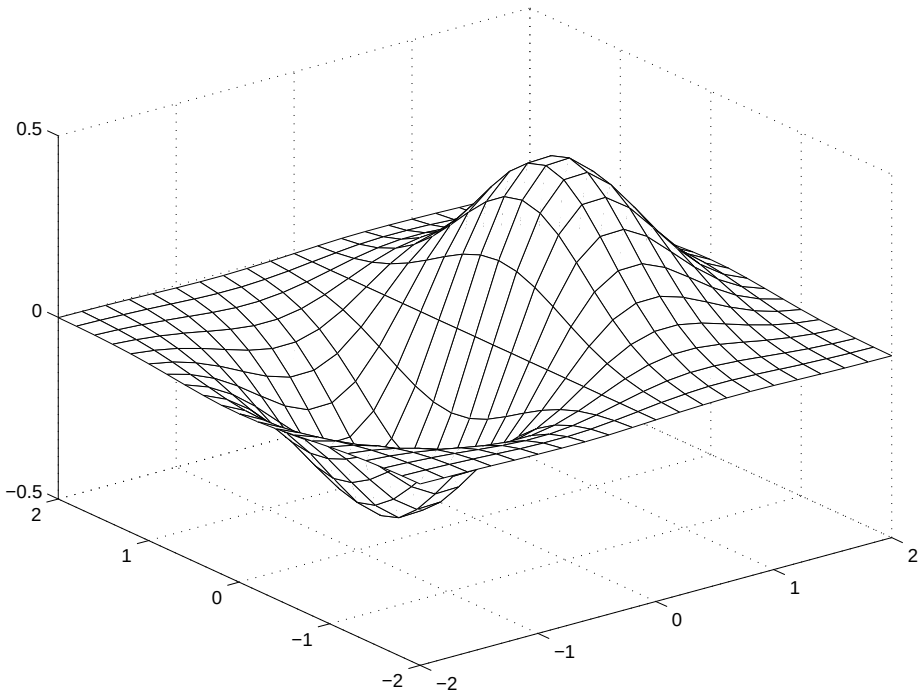

FIG. 15 – Exemple de visualisation d'une surface d'équation $z = g(x, y)$ grâce à la commande `mesh`.

Les commandes `meshc` et `meshz` s'utilisent de la même manière que la commande `mesh`. La commande `meshc` permet d'afficher des lignes de niveau sous la surface dans le plan $z = z_{min}$. La commande `meshz` permet de tracer une boite sous la surface. Un exemple d'utilisation de ces commandes est présenté à la figure 18.

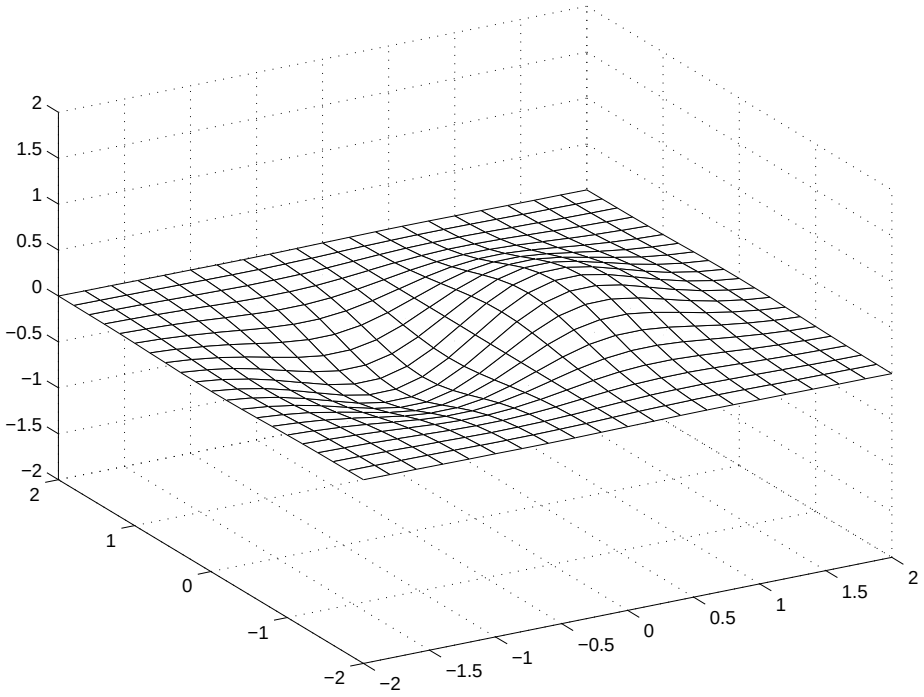

FIG. 16 – Exemple d'utilisation de la commande `axis`

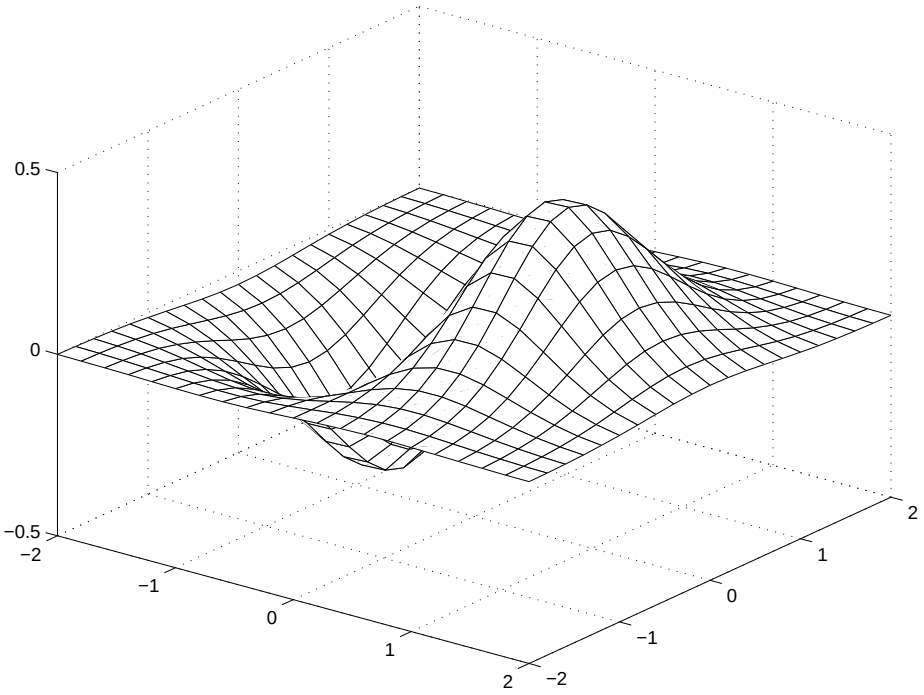

FIG. 17 – Exemple de modification de l'angle de vision par la commande `view`.

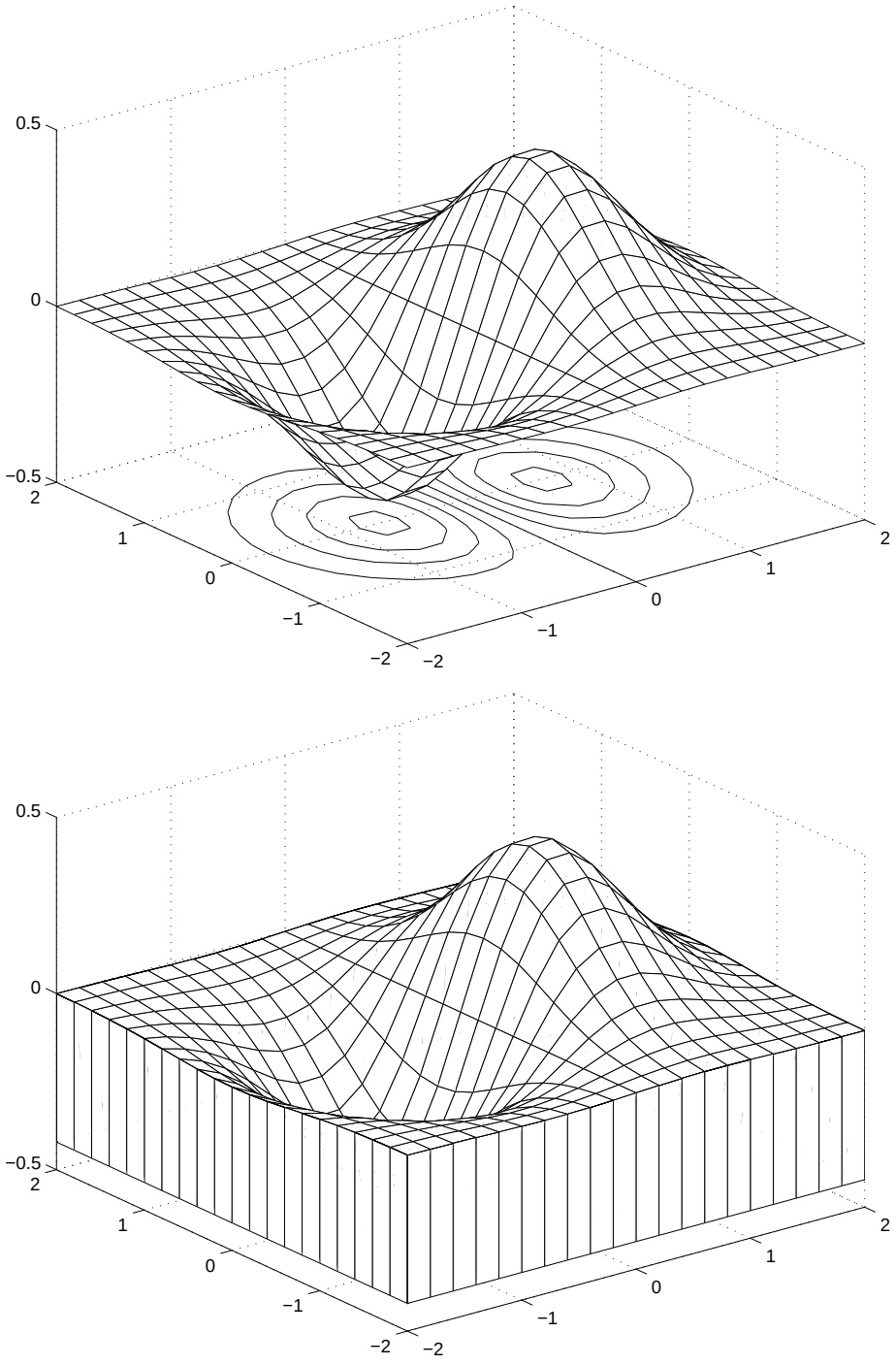

FIG. 18 – Exemple de visualisation d'une surface d'équation $z = g(x, y)$ grâce au commandes `meshc` et `meshz`.

6.4.3 Représenter une surface paramétrée

La commande `surf` permet de tracer une surface paramétrée d'équations,

$$(\mathcal{E}) \qquad \begin{cases} x = g_1(u,v), \\ y = g_2(u,v), \\ z = g_3(u,v). \end{cases}$$

La fonction $G = (g_1, g_2, g_3)$ peut être définie directement par une expression MATLAB ou être définie comme une fonction utilisateur. Pour tracer la surface paramétrée d'équation $(\mathcal{E})$ pour $u \in [u_{min}, u_{max}]$ et $v \in [v_{min}, v_{max}]$ on procède de la manière suivante :

- création d'un maillage de maille de longueur h du domaine de paramétrisation $[u_{min}, u_{max}] \times [v_{min}, v_{max}]$ grâce à la commande `meshgrid`,

  `[U,V] = meshgrid(u_min :h :u_max, v_min :h :v_max).`
- Évaluation de la fonction aux nœuds de ce maillage, soit par appel à la fonction utilisateur définissant la fonction

  `[X,Y,Z] = G(U,V)`

  soit directement en définissant la fonction par d'une expression MATLAB.
- Affichage de la surface grâce à la commande `surf`,

  `surf(X,Y,Z).`

Ainsi pour tracer la surface paramétrée d'équations

$$\begin{cases} x = v\cos u, \\ y = v\sin u, \\ z = 2u. \end{cases}$$

sur le domaine $[0, 2\pi] \times [0, 2]$ avec un maillage de maillage de longueur $h = 0.2$, on exécute :

```
>> [U,V] = meshgrid(0:.2:2*pi, 0:.2:2);
>> X = V.*cos(U);
>> Y = V.*sin(U);
>> Z = 2*U;
>> surf(X,Y,Z)
```

Si la fonction est définie comme une fonction utilisateur dans le fichier `G.m`,

```
function [x1, x2, x3] = G(u,v)

x1 = v.*cos(u);
x2 = v.*sin(u);
x3 = 2*u;
```

on exécute :

```
>> [U,V] = meshgrid(0:.2:2*pi, 0:.2:2);
>> [X,Y,Z] = G(U,V);
>> surf(X,Y,Z)
```

Dans les 2 cas on obtient le résultat présenté figure 19.

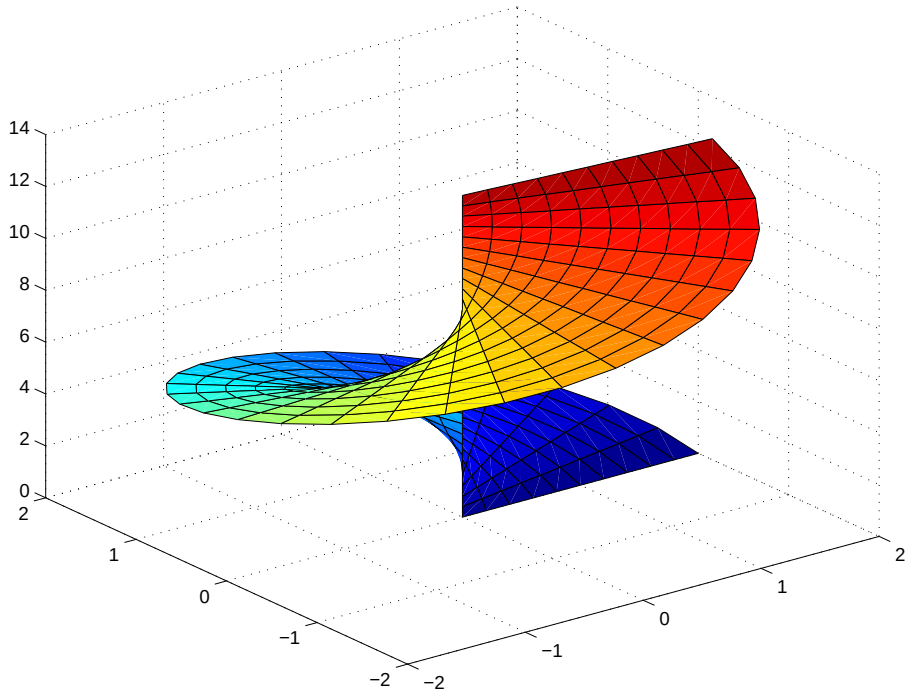

FIG. 19 – Exemple de visualisation d'une surface paramétrée grâce à la commande `surf`.

Références

[1] Jean Luc Chabert, ed. *Histoire d'algorithmes, du caillou à la puce.* Regards sur la science, Belin, 1994.

[2] Kermit Sigmon, 1993. *Matlab Tutorial.*
`http ://www.mines.utah.edu/gg_computer_seminar/matlab/matlab.html` .

Index