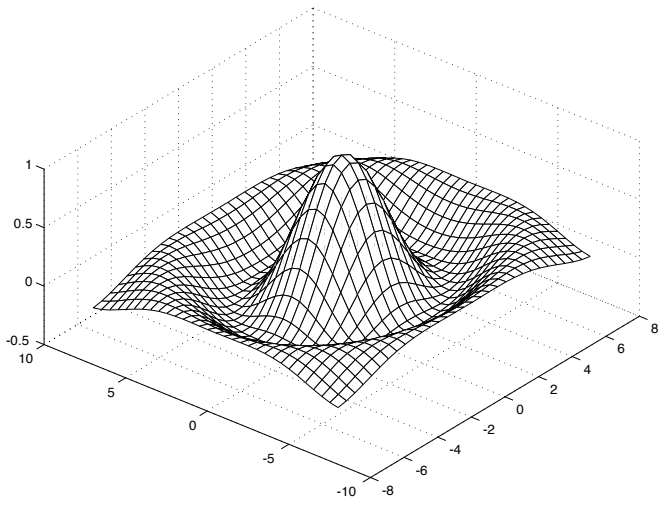
Université Paris-Dauphine
Département MIDO

Introduction à MATLAB



André Casadevall

mars 2013

Table des matières

1

MATLAB

1.1 Qu'est ce que MATLAB ?

MATLAB pour MATtrix LABoratory, est une application qui a été conçue afin de fournir un environnement de calcul matriciel simple, efficace, interactif et portable, permettant la mise en œuvre des algorithmes développés dans le cadre des projets *linpack* et *eispack*.
MATLAB est constitué d'un noyau relativement réduit, capable d'interpréter puis d'évaluer les expressions numériques matricielles qui lui sont adressées :

- soit directement au clavier depuis une fenêtre de commande ;
- soit sous forme de séquences d'expressions ou **scripts** enregistrées dans des fichiers-texte appelés *m-files* (ou fichiers `.m`) et exécutées depuis la fenêtre de commande ;
- soit plus rarement sous forme de fichiers binaires appelés *mex-files* (ou fichiers `.mex`) générés à partir d'un compilateur `C` ou `fortran`.

Ce noyau est complété par une bibliothèque de fonctions prédéfinies, très souvent sous forme de fichiers *m-files*, et regroupés en paquetages ou *toolboxes*. A côté des *toolboxes* requises *local* et *matlab*, il est possible d'ajouter des toolboxes spécifiques à tel ou tel problème mathématique, *Optimization Toolbox*, *Signal Processing Toolbox* par exemple, ou encore des toolboxes crées par l'utilisateur lui-même. Un système de chemin d'accès ou *path* permet de préciser la liste des répertoires dans lesquels MATLAB trouvera les différents fichiers *m-files* utilisés.

1.2 Une session MATLAB

L'interface-utilisateur de MATLAB varie légèrement en fonction de la version de MATLAB et du type de machine utilisée. Elle est constitué d'une fenêtre de commande qui peut être complétée par une barre de menu et pour les versions les plus récentes de plusieurs fenêtres, affichant l'historique de la session, la structure des répertoires accessibles par MATLAB... Avant la première utilisation de MATLAB, il est vivement recommandé (c'est même indispensable dans le cas d'une installation

en réseau) que chaque utilisateur crée un répertoire de travail, `tpMatlab` par exemple, où il pourra enregistrer ses fichiers. Lors de la première session, le chemin d'accès à ce répertoire sera ajouté aux chemins d'accès connus de MATLAB (`MATLABPATH`), soit en utilisant l'item `Set Path` du menu `File`, soit en tapant la commande `addpath` suivie du chemin d'accès au répertoire de travail.

1.2.1 Lancer, quitter MATLAB

Dans l'environnement `unix`, pour lançer MATLAB on tape la commande **matlab** sur la ligne de commande active ; dans les environnements `Windows` ou `MacOs`, il suffit de cliquer sur l'icône de l'application. La fenêtre de commande de MATLAB s'ouvre alors et on tape les commandes ou les expressions à évaluer à droite du prompt ». Le processus d'évaluation est déclenché par la frappe de la touche <*enter*>.

A chaque début session, l'utilisateur indiquera à MATLAB que le répertoire `myMatlab` défini précédemment est le répertoire de travail de la session en tapant la commande `cd` suivie du chemin d'accès au répertoire `myMatlab`.

On quitte MATLAB en tapant `quit` dans la fenêtre de commande ou en sélectionnant `quit` dans le menu `File` de la barre de menu pour les versions `Windows` ou `MacOs`.

1.2.2 Fonctions et commandes

Certaines fonctions de MATLAB ne calculent pas de valeur numérique ou vectorielle, mais effectuent une action sur l'environnement de la session en cours. Ces fonctions sont alors appelées **commandes**. Elles sont caractérisées par le fait que leurs arguments (lorsqu'ils existent) ne sont pas placés entre parenthèses. Les autres fonctions se comportent de façon assez semblable aux fonctions mathématiques et la valeur qu'elles calculent peut être affectée à une variable.

Dans de nombreux cas, fonctions ou commandes peuvent être appelées avec des arguments qui diffèrent soit par leur nombre, soit par leur nature (nombre, vecteur, matrice, ...). Le traitement effectué dépend alors du nombre et de la nature des arguments. Par exemple, nous verrons plus loin que la fonction `diag` appelée avec une matrice pour argument retourne le vecteur constitué par sa diagonale principale ou **vecteur diagonal**. Lorsque cette même fonction est appelée avec un vecteur pour argument, elle retourne la matrice diagonale dont le vecteur-diagonal est le vecteur donné. Aussi une fonction ou une commande n'est pas caractérisée par son seul nom, mais par sa **signature** c'est à dire l'ensemble constitué de son nom et de la liste des types de ses paramètres.

1.2.3 Historique

MATLAB conserve l'historique des commandes. Il est donc possible à l'aides des flèches du clavier de remonter dans la liste des instructions déjà entrées pour retrouver une instruction particulière pour la réutiliser et éventuellement la modifier avant de l'utiliser à nouveau.

1.2.4 Aide en ligne - `help` - `lookfor`

MATLAB comporte un très grand nombre d'opérateurs, de commandes et de fonctions. Tous ne seront pas décrits dans ce document d'autant qu'une aide en ligne efficace peut être utilisée. On peut taper les commandes suivantes :

- `help` permet d'obtenir l'aide de l'aide et donne une liste thématique ;
- `help` *nom de fonction* donne la définition de la fonction désignée et des exemples d'utilisation ;
- `lookfor` *sujet* donne une liste des rubriques de l'aide en ligne en relation avec le sujet indiqué.

Exemple 1.2.1 :

```
>> lookfor min
  minus.m: %-   Minus.
  uminus.m: %-  Unary minus.
  REALMIN Smallest positive floating point number.
```

```
  FLOOR  Round towards minus infinity.
  MIN    Smallest component.
  FMIN   Minimize function of one variable.
  FMINS  Minimize function of several variables.
  COLMMD Column minimum degree permutation.
  GMRES  Generalized Minimum Residual Method.
  QMR    Quasi-Minimal Residual Method
  SYMMMD Symmetric minimum degree permutation.
  . . .
>> help fmin
  FMIN   Minimize function of one variable.
    X = FMIN('F',x1,x2) attempts to return a value of x which is a local
    minimizer of F(x) in the interval x1 < x < x2.  'F' is a string
    containing the name of the objective function to be minimized.

    X = FMIN('F',x1,x2,OPTIONS) uses a vector of control parameters.
    If OPTIONS(1) is positive, intermediate steps in the solution are
    displayed; the default is OPTIONS(1) = 0.  OPTIONS(2) is the termination
    tolerance for x; the default is 1.e-4.  OPTIONS(14) is the maximum
    number of function evaluations; the default is OPTIONS(14) = 500.
    The other components of OPTIONS are not used as input control
    parameters by FMIN.  For more information, see FOPTIONS.

    X = FMIN('F',x1,x2,OPTIONS,P1,P2,...) provides for additional
    arguments which are passed to the objective function, F(X,P1,P2,...)

    [X,OPTIONS] = FMIN(...) returns a count of the number of steps
    taken in OPTIONS(10).

    Examples
        fmin('cos',3,4) computes pi to a few decimal places.
        fmin('cos',3,4,[1,1.e-12]) displays the steps taken
        to compute pi to about 12 decimal places.

    See also FMINS.
```

1.2.5 Liste des functions usuelles - `helpwin`

On obtient la liste des fonction MATLAB usuelles en classées par thème en tapant `helpwin` :

- `helpwin elfun` affiche la liste des fonctions mathématiques élémentaires,
- `helpwin specfun` affiche la liste des fonctions mathématiques avancées,
- `helpwin elmat` affiche la liste des fonctions matricielles élémentaires,

1.2.6 Interaction avec le système d'exploitation

Les commandes et fonctions suivantes permettent à MATLAB d'interagir avec le système d'exploitation de la machine sur laquelle il est utilisé :

- `addpath` ***path*** : ajoute le chemin d'accès (*path*) à la liste des chemins d'accès connus de MATLAB (`MATLABPATH`) ;
- `cd` ou `pwd` : affiche le chemin d'accès au répertoire de travail actuel ;
- `cd` ***path*** : fixe le répertoire de chemin d'accès *path* comme repertoire de travail ;

- `dir` ou `ls` : affiche le contenu du répertoire de travail actuel ;
- `delete` : efface le fichier spécifié (peut être utilisée sous forme de fonction sous la forme : `delete('nomDeFichier')`) ;
- `mkdir` *`path`* : crée le repertoire de chemin d'accès *path* ;
- `rmpath` *`path`* : supprime le chemin d'accès (*path*) du (`MATLABPATH`) ;
- `isdir (`*`path`*`)` : fonction booléenne qui retourne 1 si le chemin d'accès (*path*) est celui d'un répertoire, 0 sinon ;
- `filesep` : variable dont la valeur est celle du symbole séparateur de lignes (dépend du système d'exploitation de la machine utilisée) ;

2

Les “objets" de MATLAB - Listes, vecteurs, tableaux

2.1 Objets et classes de MATLAB

Un *objet* est une abstraction du monde réel (pour MATLAB celui du calcul matriciel), qui caractérisée par :

- des informations structurées ou partie `data` (pour une matrice, par exemple, le nombre de lignes, le nombre de colonnes, la valeur des coefficients . . .) ;
- par un certain comportement défini par des méthodes (pour les matrices, la somme, le produit . . .).

La famille des objets possédant le même type de structure pour la partie `data` et les même méthodes constitue une *classe*.

La **classe fondamentale** de MATLAB est la classe `double` qui modélise les tableaux mono ou bi-dimensionnels de nombres réels ou complexes à la norme IEEE (`double array`). **Les nombres réels ou complexes sont considérés comme des tableaux** 1×1. Cette classe permet également de créer, mais de façon moins naturelle, des tableaux de dimension supérieure à deux.

Les classes suivantes, sont moins fréquemment utilisées :

- la classe `char` modélise les chaînes de caractères (`char array`), un caractère unique étant une chaîne de longueur un ;
- la classe `sparse` modélise les matrices creuses (*i.e.* dont la plupart des éléments sont nuls) réelles ou complexes.

A partir de la version 5, MATLAB a proposé des structures de données complémentaires souvent utilisées dans les objets prédéfinis de MATLAB, les objets graphiques en particulier :

- la classe `structure` modélise les tableaux de “structures" ; les “structures" au sens de MATLAB sont des structure de données assez semblables aux `struct` du `C` pour leur partie `data` du moins (leurs composantes ou *champs* sont accessibles par une notation pointée) ;
- la classe `cell` modélise les tableaux de “cellules" ou `cell array` ; les cellules sont des sortes de conteneurs dans les quels on peut plaçer d'autres objets ; on accède à chacun par une notation indicée.

Dans les versions antérieures à la version 7, **MATLAB ne proposait ni valeurs prédéfinies `true` ou `false` ni classe pour modéliser les booléens**. Depuis la version 7 MATLAB possède une classe `logical` . Comme dans les versions antérieures, `false` est associé à la valeur 0 ; `true` est associé à 1 et par extension, à toute valeur non nulle.

En conclusion, toutes les classes de MATLAB sont associées à des tableaux de structures de données (au sens général) relativeent classiques.

2.2 Valeurs littérales

Le terme de *valeur littérale* désigne les valeurs qu'on peut directement taper au clavier et qui peuvent être alors affectées à une variable.

2.2.1 Nombres

Les nombres réels et entiers (MATLAB ne distingue pas entre réels et entiers) sont écrits sous les formes décimales ou scientifiques usuelles :

$$2,\ 3.214,\ 1.21E33.$$

Les nombres complexes sont écrits sous la forme $a + bi$, comme dans `1+2i`.

Fonctions relatives aux nombres complexes

- `real` et `imag` renvoient respectivement la partie réelle et la partie imaginaire du complexe passé en paramètre,
- `abs` et `arg` renvoient respectivement le module et l'argument du complexe passé en paramètre,
- `conj` renvoie le complexe conjugué du nombre complexe passé en paramètre.

2.2.2 Tableaux de nombres

Les tableaux de nombres réels ou complexes de dimension un ou deux suivent la syntaxe suivante :

- un tableau est délimité par des crochets ;
- les éléments sont entrés ligne par ligne ;
- les éléments appartenant à la même ligne sont séparés par des espaces (ou par des virgules) ;
- les différentes lignes qui **doivent posséder le même nombre d'éléments**, sont séparées par des points-virgule.

Exemple 2.2.1 :

Les tableaux :

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \qquad \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \qquad \begin{array}{cccc} 1 & 2 & 0 & 0 \\ 0 & 2 & 3 & 1 \\ 0 & 0 & 2 & 2 \end{array}$$

s'écrivent sous la forme `[1 2 3 4 ]` `[1; 2; 3; 4 ]` `[1 2 0 0 ; 0 2 3 1 ; 0 0 2 2 ]` :

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

>> [1; 2; 3; 4 ]
  ans =
    1
    2
    3
    4

>> [1 2 0 0 ; 0 2 3 1 ; 0 0 2 2 ]
  ans =
    1  2  0  0
    0  2  3  1
    0  0  2  2
```

Lorsque toutes les lignes ne possèdent pas le **même nombre d'éléments** :

```
>> [1 2 ; 1 2 3]
  ??? Number of elements in each row must be the same.
```

Dans la suite, on appelera :

- **vecteur** un tableau de format $(n, 1)$ *i.e.* ne comportant qu'une seule colonne ;
- **liste** ou encore **vecteur-ligne** un tableau de format $(1, n)$ *i.e.* ne comportant qu'une seule ligne ;
- **tableau** un tableau au sens commun du terme, c'est à dire une structure organisée en lignes et colonnes.

2.2.3 Caractères et chaînes de caractères

On écrit les **caractères** et les **chaînes de caractères** entre apostrophes : `'a', 'toto'` et MATLAB les considère comme des chaînes de caractères de longueur un.

D'autre part, pour MATLAB, chaînes de caractères et liste de caractères sont des objets de même nature :

Exemple 2.2.2 :

La liste de caractères `['a' 'b' 'c' 'd' 'e']` est identique à la chaînes de caractères `['abcde']` :

```
>>['a' 'b' 'c' 'd' 'e']
  ans =
    abcde
```

Mieux encore, `'abcde'` ; `['abc' 'de']` est identique à `'abcde'` :

```
>>['abc' 'de']
  ans =
    abcde
```

Cet exemple donne un idée du rôle des crochets [ ]. Les crochets sont le symbole de l'**opérateur de concaténation** :

- concaténation "en ligne" lorsque le séparateur est un espace ou une virgule ;
- concaténation "en colonne" lorsque le séparateur est un point-virgule comme dans les tableaux de nombres (il est alors nécessaire que les listes de nombres ou de caractères ainsi concaténées possèdent le même nombre d'éléments).

Exemple 2.2.3 :

La liste de caractères `['a' 'b' 'c' 'd' 'e']` est identique à la chaînes de caractères `['abcde']` :

```
>>['abc' ; 'abcd']
  ??? All rows in the bracketed expression must have the same
     number of columns.
```

2.2.4 Cellules et tableaux de cellules - `cell array`

Une cellule est un conteneur dans le quel on peut placer toute sorte d'objets : nombre, chaîne de caractères, tableau et même tableau de cellules. Les tableaux de cellules permettent regrouper dans une même structure des éléments de nature très différente. La syntaxe des tableaux de cellules est voisine de celle des tableaux usuels, les crochets étant remplacés par des accolades.

Exemple 2.2.4 :

```
>> {'paul' 4 ; 'vincent' 7;'...' 0}
  ans =
    'paul'     [4]
    'vincent'    [7]
    '...'        [0]
```

La manipulation des ces objets (sauf lorsqu'on on se limite à des composants qui sont des nombres ou des chaînes de caractères) est un peu plus délicate que celle des tableaux usuels et sera examinée dans un prochain chapitre.

2.3 Variables

Une caractéristique de MATLAB est que les variables n'ont pas à être déclarées, leur nature se déduisant automatiquement de l'objet qui leur est affecté *(cf. exemple 2.3.4 - section 2.3.3)*.

2.3.1 Identificateurs

Les règles de dénomination des variables sont très classiques :
- un identificateur débute nécessairement par une lettre, éventuellement suivie de lettres, de chiffres ou du caractère souligné (`_`) ;
- sa longueur est inférieure ou égale à 31 caractères ;
- dans les identificateurs, les majuscules sont distinguées des minuscules (on dit qu'ils sont *case-sensitive*).

Voici quelques identificateurs prédéfinis :
- `ans` désigne le résultat de la dernière évaluation ;
- `pi` est le nombre $\pi = 3,416\cdots$;
- `eps` désigne l'*"epsilon-machine"*, c'est à dire le nombre $inf\{\epsilon \geq 0$ tels que $1 < 1+\epsilon\}$;
- `inf` désigne l'infini au sens d'une évaluation du type $(1/0)$;
- `NaN` signifie "Not a Number" - peut être le résultat d'une évaluation du type $(0/0)$;
- `i,j` - `i` et `j` représentent tous deux le nombre imaginaire unité $(\sqrt{-1})$ - **attention à ne pas utiliser `i` et `j` comme indices pour accéder aux éléments d'un tableau** ;
- `realmin` désigne le petit nombre réel positif ;
- `realmax` désigne le plus grand nombre réel positif.

2.3.2 Affectation

Le symbole d'affectation de valeur à une variable est le caractère `=`.

Exemple 2.3.1 :

```
>> a =  [1 2 3 4 ]
  a =
    1  2  3  4
>> a = 'abc'
  a =
     abc
```

L'exemple ci-dessus montre bien que dans MATLAB les variables ne sont ni déclarées ni typées.

2.3.3 Espace de travail - *workspace*

L'ensemble des variables et les objets qui leur sont associées constitue l'espace de travail ou *workspace* de la session en cours. Le contenu de cet espace de travail va se modifier tout au long du déroulement de la session et plusieurs commandes ou fonctions permettent de le gérer de façon efficace.

Les commande `who` et `whos`

Ces commandes (le nom d'une commande est contrairement aux fonctions, suivi par la liste non parenthèsée du ou des paramètres) donnent la liste des variables composant l'espace de travail. La commande `who` donne la liste des variables présentes dans l'espace de travail. La commande `whos` retourne une information plus complète comportant pour chaque variable, la dimension du tableau qui lui est associé, la quantité de mémoire utilisée et la classe à laquelle elle appartient.

Exemple 2.3.2 :

On définit les variables `a`, `b` et `c` :

```
>> a = 2 ; b = 'azerty'; c = [1 2 3 ; 5 3 4] ;
  % la partie de la ligne qui suit le symbole % est un commentaire
  % les points-virgules inhibent l'affichage de la valeur des variables
>> who
  Your variables are :
  a      b      c
```

La commande `whos` donne une information plus complète :

```
» whos
 Name  Size  Bytes  Class
 a     1x1   8      double array
 b     1x6   12     char array
 c     2x3   48     double array
Grand total is 13 elements using 68 bytes
leaving 14918672 bytes of memory free
```

On peut également appliquer `whos` avec pour argument une ou plusieurs variables :

Exemple 2.3.3 :

```
» whos b c
 Name  Size  Bytes  Class
 b     1x6   12     char array    Grand total is 12 elements using 60 bytes
 c     2x3   48     double array
leaving 14918960 bytes of memory free.
```

Ce dernier exemple montre bien que le type d'une variable est induit par sa valeur.

Exemple 2.3.4 :

On modifie la valeur de la variable a son type est alors modifié en conséquence :

```
» clear
» a = [1 2 3 4 ] ; whos a
 Name  Size  Bytes  Class
 a     1x4   32     double array
» a = 'abc' ; whos a
 Name  Size  Bytes  Class
 a     1x3   32     char array
```

Les fonctions `size`, `size( ,1)` et `size( ,2)`

La fonction `size` retourne le couple `(nl,nc)` formé du nombre de lignes `nl` et du nombre de colonnes `nc` du tableau associé à la variable donnée comme comme argument.

Exemple 2.3.5 :

On suppose que l'environnement de travail est constitué des trois variables `a`, `b` et `c` de l'exemple précédent. La fonction `size` produit l'affichage suivant :

```
>> size(a)
ans =
  1 1
>> size(b)
ans =
 1   6
```

Pour accéder plus facilement au nombre de lignes et au nombre de colonnes, on peut affecter la valeur retournée par `size` à un tableau à deux éléments `[nl, nc]` :

```
>> size(c)
>> [nl, nc] = size(c)
nl =
  2
nc =
  3
```

Enfin `size( ,1)` et `size( ,2)` permettent l'accès direct au nombre de lignes et au nombre de colonnes d'un tableau :

```
>> size(c,1)
ans =
  2
>> size(c,2)
ans =
  3
```

La fonction `class`

La fonction `class` retourne le nom de la classe à laquelle appartient la variable donné comme comme argument.

Exemple 2.3.6 :

Avec le même espace de travail que dans l'exemple précédent, la fonction `class` produit l'affichage suivant :

```
>> ca = class(a)
ca =
  double
>> cb = class(b)
cb =
  char
```

Les commandes `save`, `load` et `clear` - fichiers `.mat`

Les commandes `save`, `load` et `clear` permettent d'intervenir directement sur l'environnement de travail :

- `save` permet de sauver tout ou partie de l'espace de travail sous forme de fichiers binaires appelés *mat-files* ou fichiers `.mat`, plus précisément :
  - `save` : enregistre la totalité de l'espace de travail dans le fichier *matlab.mat* ;
  - `save` *nom de fichier* : l'espace de travail est enregistré dans le fichier *nom de fichier* ;
  - `save` *nom de variable ... nom de variable* : enregistre les variables indiquées (et les objets qui leurs sont associés) dans un fichier `.mat` qui porte le nom de la première variable ;
  - `save` *nom de fichier nom de variable ... nom de variable* : enregistre les variables dans le fichier dont le nom a été indiqué.
- `load` permet d'ajouter le contenu d'un fichier `.mat` à l'espace de travail actuel ;
- `clear` supprime une ou plusieurs variables (et les objets aux quelles elles font référence) de l'environnement de travail, plus précisément :
  - `clear` sans argument, supprime toutes les variables de l'espace de travail actuel ;

– `clear` *nom de variable ... nom de variable* : supprime les variables indiquées de l'espace de travail.

Exemple 2.3.7 :

Cet exemple illustre les effets de `save`, `load` et `clear`. Tout d'abord on définit trois variables `a`, `b` et `c`

```
>> a = 1; b = 2.5; c = 'hello'
c =
     hello
>> save a b
 % les variables a et b sont enregistrées dans le fichier a.mat créé
 % dans le répertoire de travail
>> clear a b
 % les variables a et b sont supprimées comme le montre l'évaluation
>> a, c
??? Undefined function or variable 'a'
c =
     hello
>> load a
 % on copie les variables du fichier a.mat dans l'espace de travail en cours
>>
>> x = a + b
x =
      3.5
 % les variables a et b sont bien présentes
```

2.4 Listes et vecteurs

Les listes et les vecteurs sont des tableaux de nombres au format particulier : un vecteur est un tableau qui ne comporte qu'une seule colonne ; une liste (ou vecteur-ligne) est un tableau qui ne comporte qu'une seule ligne. MATLAB propose un certain nombre de fonctions qui en simplifient l'usage.

2.4.1 Construction de listes

Valeurs littérales de type liste

Ainsi que nous l'avons déjà vu, on peut définir la valeur d'une liste en donnant la suite de ses éléments séparés par des espaces (ou des virgules) ; la liste est délimitée par des crochets :

Exemple 2.4.1 :

```
>> l1 = [1 3 5 10 ] , l2 = [2, 4]
l1 =
      1 3 5 10
l2 =
      2 4
```

Constructeur de listes

L'expression $(v_i : p : v_f)$ crée une liste dont les éléments constituent une progression arithmétique de valeur initiale v_i, de pas p et dont tous les termes sont inférieurs ou égaux à v_f. Lorsque la valeur du pas est omise comme dans l'expression $(v_i : v_f)$, la valeur du pas est par défaut fixée

à un.

Exemple 2.4.2 :

```
>> l2 = 1 : 4
l2 =
    1  2  3  4
% le pas par défaut vaut 1
>> l3 = 1 : 5.6
l3 =
    1  2  3  4  5
        (puisque 5.0000 + 1 est strictement supérieur à 5.6)
>> l4 = 1.5 : 0.3 : 2.5
l4 =
    1.5000  1.8000  2.1000  2.4000
        (puisque 2.4000 + 0.3 est strictement supérieur à 2.5)
```

Fonctions à valeur liste

- La fonction `linspace(`v_i, v_f, n`)` crée une liste de n nombres uniformément répartis entre les valeurs v_i et v_f ; `linspace(`v_i, v_f, n`)` est équivalent à $\left(v_i : \dfrac{v_f - v_i}{n-1} : v_f\right)$.
- La fonction `ones(`n`)` crée une liste de n éléments tous égaux à 1 ;
- La fonction `zeros(`n`)` crée une liste de n éléments tous égaux à 0.

Exemple 2.4.3 :

```
>> l4 = linspace(0,5, 2, 4)
l4 =
    0,5000  1.0000  1.5000  2.0000
```

Dans la suite *(c.f. 2.5.8)*, on découvrira d'autres fonctions qui permettent de construire des tableaux de format (m, n) quelconque ; ces fonctions permettent en conséquence la construction de listes et de vecteurs particuliers.

2.4.2 Construction de vecteurs

Valeurs littérales de type vecteur

On peut définir la valeur d'un vecteur en tapant **entre deux crochets** la suite de ses éléments séparées par des points-virgule comme on le voit dans l'exemple suivant :

Exemple 2.4.4 :

```
>> v1 = [1 ; 3 ; 5 ]
v1  =
      1
      3
      5
```

Transposition

L'opérateur de transposition est noté `'` (ou `.'` pour les listes et les vecteurs de nombres complexes).

La transposée d'une liste étant un vecteur, pour construire des vecteurs, on peut utiliser les expressions et les fonctions vues pour les listes puis transposer le liste obtenue.

Exemple 2.4.5 :

```
>> v2 =  [1 2 3]'
v2 =
     1
     2
     3
>> v3 = (1.5 : 0.3 : 2.5)'
v3 =
    1.5000
    1.8000
    2.1000
    2.4000
```

On n'oubliera pas les parenthèses nécessaires pour délimiter le constructeur de liste.

2.4.3 Nombre d'éléments d'une liste ou d'un vecteur - `length`

La fonction `size` appliquée à une liste ou à un vecteur retourne, comme pour tous les tableaux, le nombre de lignes et le nombre de colonnes de la liste ou du vecteur. Le nombre de ligne d'une liste est bien évidemment un. La même remarque vaut pour le nombre de colonnes d'un vecteur. Aussi, pour les listes et les vecteurs on utilise de préférence la fonction `length` qui retourne le nombre d'éléments ou **longueur** de la liste ou du vecteur.

Exemple 2.4.6 :

```
>> l = [1 2 3 4] ; length(l)
ans =
     4
>> v = [5 6 7 8 9]' ; length(v)
ans =
     5
```

!!! **Remarque :**

Les exemples précédents montrent que l'on peut écrire sur la même ligne plusieurs expressions à la suite à la condition de les séparer par une virgule ou un point-virgule. La différence entre ces deux séparateurs est que **le résultat de l'évaluation d'une expression suivie d'un point-virgule n'est pas affiché**.

2.4.4 Norme vectorielle - `norm`

La notion de longueur synonyme de nombre d'éléments ne doit pas être confondue avec la notion mathématique de **norme vectorielle**. Les fonctions suivantes permettent de calculer les normes usuelles d'un vecteur ou d'un vecteur-ligne (ou liste) v de $\mathbb{R}^n$:

- `norm(v,p)` $= \left[\sum_{k=1}^{n} |v_k|^p\right]^{1/p}$

- `norm(v)` = `norm(v,2)` = $\left[\sum_{k=1}^{n}(v_k)^2\right]^{1/2}$
- `norm(v,inf)` = $\max_k |v_k|$.

Exemple 2.4.7 :

```
>> l = [1 1 1 1] ; v = l' ;  norm(v)
ans =
     2
>> norm(l)
ans =
     2
```

2.4.5 Accès aux éléments d'une liste ou d'un vecteur - `end`

Soient `s` une liste ou un vecteur non-vide, et k un entier compris entre 1 et la longueur de la liste ou du vecteur considéré ($1 \leq k \leq length(s)$). On accède à l'élément d'indice k de la liste ou du vecteur `s` par `s(k)`, le premier élément de la liste ou du vecteur étant indicé par 1. `s(end)` désigne l'élément de plus grand indice de la liste ou du vecteur.
Attention à ne pas utiliser i ou j qui désignent $\sqrt{-1}$, pour indicer les éléments d'une liste ou d'un vecteur *(c.f. 2.3.1)*.

Exemple 2.4.8 :

```
>> s = [1 3 5 8] ; s(1)
ans =
     1
>> k = 3 ; s(k)
ans =
     5
>> s(end)
ans =
     8
```

L'accès en lecture à un élément d'indice négatif ou dont la valeur est strictement supérieure à la longueur de la liste (ou du vecteur), conduit à une erreur :

Exemple 2.4.9 :

```
>> s = [1 3 5] ; length(s)
ans =
     3
>> s(4)
???  Index exceeds matrix dimensions.
```

Par contre, **il est possible d'affecter une valeur à un élément d'une liste ou d'un vecteur dont l'indice dépasse la longueur de la liste ou du vecteur**. Comme le montre l'exemple suivant, les éléments dont l'indice est compris entre la longueur de la la liste (ou du vecteur) et l'indice donné sont affectés de la valeur 0. La longueur de la liste (ou du vecteur) est alors modifiée en conséquence.

Exemple 2.4.10 :

```
>> s = [1 3 5] ; length(s)
ans =
     3
>> s(6) = 6 ; s
 s =
     1   3   5   0   0   6
>>  length(s)
ans =
     6
```

2.4.6 Extraction de sous-listes ou de sous-vecteurs

Soient `s` une liste (ou un vecteur) non-vide et `lst` une **liste d'entiers dont la valeur est comprise entre** 1 **et la longueur `length(s)` de la liste (ou du vecteur)**, alors `s(lst)` est la liste (ou le vecteur) formé par les éléments de `s` dont l'indice appartient à `lst`.

Exemple 2.4.11 :

```
>> s = [1  3  5  0  0  6] ;
 % on veut extraire de s la sous-liste formée des éléments de rang impair
 % on définit lst par :
lst = 1 : 2 : length(s) ;  % on aurait aussi pu écrire : lst = [1 3 5]
s(lst) =
     1     5     0
```

2.5 Tableaux

2.5.1 Construction de tableaux

Valeurs littérales de type tableau

On a déjà vu que pour définir la valeur d'un tableau (sauf pour les tableaux d'ordre 1), il suffit de concaténer “en colonne” des listes de nombres de **même longueur** :

Exemple 2.5.1 :

Le tableau

```
     1     2     0     0
     0     2     3     1
     0     0     2     1
```

est défini par :

```
>> T = [1 2 0 0 ; 0 2 3 1 ; 0 0 2 1]
T =
     1     2     0     0
     0     2     3     1
     0     0     2     1
```

Concatenation de tableaux - [ ]

L'opérateur [ ] permet la concaténation de tableaux :
- si les tableaux $\{T_k\}_{k=1,2,\cdots,n}$ possèdent le **même nombre de lignes** les expressions équivalentes $[T_1, T_2, \cdots, T_p]$ ou $[T_1 T_2 \cdots T_p]$ créent un tableau :
  - qui a le même nombre de lignes que les tableaux composants ;

  - dont le nombre de colonnes est la somme des nombres de colonnes de chacun des tableaux composants ;
  - qui est obtenu en concaténant “en ligne" les tableaux composants.
- si les tableaux T_k ont le **même nombre de colonnes** l'expression $[T_1; T_2; \cdots; T_p]$ crée un tableau :
  - qui a le même nombre de colonnes que les tableaux composants ;
  - dont le nombre de lignes est la somme des nombres de lignes de chacun des tableaux composants ;
  - qui est obtenu en concaténant “en colonne" les tableaux composants.

Exemple 2.5.2 :

```
>>T1 = [1 2 ; 2 3]
T1 =
     1     2
     2     3
>>T2 = [3 4 ; 6 7]
T2 =
     3     4
     6     7
>> T3 =  [T1 , T2]  ou [T1  T2]
T3 =
     1     2     3     4
     2     3     6     7
>> T4 = [T1 ; T2]
T4 =
     1     2
     2     3
     3     4
     6     7
```

2.5.2 Accès aux éléments d'un tableau - `end`

Soient `T` un tableau, et l et k deux entiers tels que ($1 \leq l \leq$ `size(T,1)`) et ($1 \leq k \leq$ `size(T,2)`). Alors `T(l,k)` désigne l'élément de la ligne l et de la colonne k du tableau `T`.
Utilisé comme indice de ligne, respectivement de colonne, `end` est égal à la plus grande valeur possible pour cet indice.
Attention à ne pas utiliser i ou j qui désignent $\sqrt{-1}$, pour indicer les éléments d'un tableau *(c.f. 2.3.1)*.

Exemple 2.5.3 :

```
>>T = [1 2 0 0 ; 0 2 3 1 ; 0 0 2 4]
T =
     1     2     0     0
     0     2     3     1
     0     0     2     4
>> x = T(2, 3)
x =
     3
>> x = T(2, end)
x =
```

```
     1
>> x = T(end, end)
x =
     4
```

L'accès en lecture à un élément dont les indices seraient négatifs ou dont la valeur serait strictement supérieure au nombre de lignes ou au nombre de colonnes du tableau, conduit à une erreur :

Exemple 2.5.4 :

```
>>x = T(1,5)
???  Index exceeds matrix dimensions
```

Par contre, il est possible **d'affecter une valeur** à un élément d'un tableau dont les indices dépassent le nombre de ligne pour le premier indice, le nombre de colonnes pour le second.
Comme le montre l'exemple suivant, les éléments du tableau dont les indices sont compris entre le nombre de lignes et le nombre de colonnes, et les indices spécifiés, prennent la valeur 0.

Exemple 2.5.5 :

```
>> T = [1 2 0 0 ; 0 2 3 1 ; 0 0 2 1]
  T =
     1     2     0     0
     0     2     3     1
     0     0     2     1
>> T(1,5) = 2
  T =
     1     2     0     0     2
     0     2     3     1     0
     0     0     2     1     0
```

2.5.3 Lignes et colonnes d'un tableau

Soient `T` un tableau et `l` un entier compris entre 1 et la nombre de lignes du tableau. Alors `T(l,:)` désigne la ligne l de `T` et `T(end,:)` désigne la dernière ligne de `T` . De même, si k est un entier compris entre 1 et le nombre de colonnes de `T`, `T(:,k)` désigne la colonne k de `T` et `T(:,end)` désigne la dernière colonne de `T` .

Exemple 2.5.6 :

```
>> T
     1     2     0     0     2
     0     2     3     1     0
     0     0     2     1     0
>> x = T(2, :)
  x =
     0     2     3     1     0
>> y = T(:, 3)
  y =
     0
     3
     2
```

2.5.4 Sous-tableaux et blocs

Soient `T` un tableau, `l` une liste d'entiers compris entre 1 et le nombre de lignes de `T`, et `k` une liste d'entiers compris entre 1 et le nombre de colonnes de `T`. Alors `T(l, k)` est le sous-tableau de `T` formé par les éléments de `T` dont l'indice de ligne appartient à `l` et l'indice de colonne appartient à `k`.

Exemple 2.5.7 :

```
>> T  = [1     2     3     4     5 ; 2     2     3     1     0 ;   3     0     2     1     1]
     1     2     3     4     5
     2     2     3     1     0
     3     0     2     1     1
>> l = [1 2]; k = [1 3 5];
>> T1 = T(l, k)
  T1 =
     1     3     5
     2     3     0
```

`T(l, k)` est un **bloc** lorsque les listes `l` et `k` sont constituées d'entiers consécutifs, ce qui est le cas de `T2` dans l'exemple suivant :

Exemple 2.5.8 :

```
>> T  = [1     2     3     4     5 ; 2     2     3     1     0 ;   3     0     2     1     1]
     1     2     3     4     5
     2     2     3     1     0
     3     0     2     1     1
>> l = [1 2]; k = [1 2 3];

>> T2 = T(l, k)
  T2 =
     1     3     3
     2     2     3
```

2.5.5 Fonction `repmat`

La fonction `repmat` réalise l'opération inverse. Elle permet la création d'un "grand" tableau dont chaque bloc est identique au tableau passé comme premier argument ; les deux autres arguments représentent respectivement le nombre de fois où la matrice-argument est répétée suivant les colonnes, respectivement suivant les lignes :

Exemple 2.5.9 :

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

>> T1 = repmat( T, 2, 3)
  T1 =
     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

2.5.6 Éléments diagonaux d'un tableau - `diag`

Soit `T` un tableau, `diag(T)` retourne le vecteur (**vecteur diagonal** de `T`) formé des éléments de la **diagonale "principale"** de `T`, c'est à dire les éléments de la forme `T(p,p)` où p est un entier compris entre 1 et $\min(\texttt{size(T,1)}, \texttt{size(T,2)})$; remarquons qu'il n'est pas nécessaire que le tableau `T` soit carré.
Plus généralement pour tout entier k compris entre $-\texttt{size(T,1)}$ et `size(T,2)`, on appelle **diagonale de rang** k l'ensemble des éléments de `T` de la forme `T(p, p+k)` où :

- $1 \leq k \leq \texttt{size(T,2)}$ et $1 \leq p \leq \min(\texttt{size(T,1)}, \texttt{size(T,2)} - k)$ (sur-diagonale de rang k)
- $k = 0$ et $1 \leq p \leq \min(\texttt{size(T,1)}, \texttt{size(T,2)})$ (diagonale principale)
- $-\texttt{size(T,1)} \leq k \leq -1$ et $1 - k \leq p \leq \min(\texttt{size(T,1)}, \texttt{size(T,2)} - k)$ (sous-diagonale de rang k-

La fonction `diag(T,k)` retourne le vecteur formé des éléments de la diagonale de rang k de `T`.

Exemple 2.5.10 :

```
>> T = [ 1     2     3     4     5 ;  2     2     0     0     2 ;
                           3     2     3     1     0 ; 4     0     2     1     1]
     1     2     3     4     5
     2     2     0     0     2
     3     2     3     1     0
     4     0     2     1     1

>> v = diag(T) % diagonale "principale"
  v =
     1
     2
     3
     1

>> v = diag(T,1) % sur-diagonale de rang 1
  v =
     2
     0
     1
     1

>> v = diag(T,-2) % sous-diagonale de rang 2
  v =
     3
     0
```

2.5.7 Fonction `tril` et `triu`

Les fonctions **tril** et **triu** extraient respectivement les termes situés sur et au-dessous de la diagonale de rang k, et les termes situés sur et au-dessus de la diagonale de rang k.

Exemple 2.5.11 :

Avec le même tableau `T` que dans l'exemple *2.5.10* ci-dessus :

```
>> tril(T,1)
  ans =
     1     2     0     0     0
     2     2     0     0     0
     3     2     3     1     0
     4     0     2     1     1

>> triu(T,1)
  ans =
     0     2     3     4     5
     0     0     0     0     2
     0     0     0     1     0
     0     0     0     0     1
```

Remarque : !!!

`tril(t,0)` s'écrit aussi `tril(t)`, de même, `triu(t,0)` s'écrit aussi `triu(t)`.

2.5.8 Tableaux particuliers

Les fonction ci-dessous permettent de construire des tableaux correspondant aux matrices usuelles : identité, matrice nulle, ainsi qu'à des matrices-test très utiles pour valider des algorithmes d'analyse matricielle (voir aussi la fonction `gallery`) .

Fonction	Argument	Résultat
`diag(s)`	un vecteur ou une liste `s`	matrice diagonale dont la diagonale est la liste ou le vecteur `s`
`vander(s)`	un vecteur ou une liste `s`	matrice de *Vandermonde* d'ordre `length`(s) engendrée par s
`eye(n)`	un entier n	matrice identité d'ordre n I_n
`eye(n,m)`	deux entiers n et m	sous-matrice (n,m) de $I_{\max(n,m)}$ (1)
`hilb(n)`	un entier n	matrice de *Hilbert* d'ordre n : $h_{i,j} = 1/(i+j-1)$
`invhilb(n)`	un entier n	inverse de la matrice de *Hilbert* d'ordre n
`magic(n)`	un entier n	carré magique d'ordre n
`ones(n)`	un entier n	matrice A carrée d'ordre n telle que $a_{i,j} = 1$
`ones(n,m)`	deux entiers n et m	matrice A de format (n,m) telle que $a_{i,j} = 1$
`pascal(n)`	un entier n	matrice de *Pascal* d'ordre n
`rand(n)`	un entier n	matrice aléatoire carrée d'ordre n
`zeros(n)`	un entier n	matrice nulle d'ordre n
`zeros(n,m)`	deux entiers n et m	matrice nulle de format (n,m) (1)
`zeros(n,1)`	un entier n	vecteur nul de $\mathbb{R}^n$
`wilkinson(n)`	un entier n	matrice de *Wilkinson* d'ordre n

(1) Les fonctions `eye`, `ones` et `zeros` peuvent être appelées avec deux arguments entiers n et m. Le résultat est alors une matrice de format $n \times m$ formée des n premières lignes et des m premières

colonnes de la matrice carrée du même type d'ordre $\max(n, m)$.
D'autre part, on peut remarquer que la fonction `diag` retourne une valeur très différente suivant le type de son argument (cette propriété est appelée *polymorphisme*) :

- lorsque l'argument est un tableau, `diag` retourne le vecteur diagonal du tableau ;
- lorsque l'argument est un vecteur, `diag` retourne une matrice diagonale dont le vecteur diagonal est l'argument avec lequel `diag` est appelé .

Ce petit exemple illustre les deux aspects de la fonction `diag` :

Exemple 2.5.12 :

```
>>  T =
     1     2     3
     1     2     1
     0     1     3

>> d = diag(T))   % d est le vecteur diagonal de T
   d =
     1
     2
     3

>> D = diag(diag(T)) % D est le tableau diagonal dont la diagonale est celle de T
  D =
     1     0     0
     0     2     0
     0     0     3
```

2.6 Fonctions opérant sur les éléments d'un tableau

Les fonctions présentées ci-dessous effectuent des opérations arithmétiques itérativement sur les éléments d'une liste ou d'un vecteur. Appliquées à un tableau, elles effectuent ces mêmes opérations sur les colonnes du tableau (sauf pour `cov`). Ce sont des fonctions extrêmement efficaces *(c.f. 3.6)*.

2.6.1 Fonctions `sum` et `prod`

- Appliquée à une liste ou un vecteur, la fonction `sum` (respectivement `prod`) calcule la somme (respectivement le produit) des éléments la liste ou du vecteur.
- Appliquée à un tableau la fonction `sum` (respectivement `prod`) retourne une liste dont chacun des éléments est la somme (respectivement le produit) des éléments de chaque colonne.

Exemple 2.6.1 :

```
>> s = [5 2 3 1 7] ; p = prod(s) , s = sum(s)
  p =
   210
  s =
    18

>> T = vander([1 2 3])
  T =
     1     1     1
     4     2     1
     9     3     1
```

```
>> p = prod(T)
  p =
    36    6     1

>> s = sum(T)
  s =
    14    6     3
```

2.6.2 Fonctions `max` et `min`

- Appliquée à une liste ou un vecteur, la fonction `max` (respectivement `min`) détermine le plus grand élément (respectivement le plus petit élément) de la liste ou du vecteur et éventuellement la position de cet élément dans la liste ou le vecteur.
- Appliquée à un tableau la fonction `max` (respectivement `min`) retourne la liste des plus grands (respectivement plus petit éléments) de chaque colonne.

Exemple 2.6.2 :

```
>> s = [5 2 3 1 7] ; [ma, ind] = max(s)
  ma =
      7
  ind =
     5

>> [mi, ind] = min(s)
  mi =
      1
  ind =
     4
```

Exemple 2.6.3 :

```
>> T = magic(3) ,  [ma, ind] = max(T)
  T =
     8     1     6
     3     5     7
     4     9     2
  ma =
     8     9     7
  ind =
     1     3     2
```

Pour obtenir la valeur de l'élément maximal du tableau, il suffit d'applique deux fois la fonction `max` :

Exemple 2.6.4 :

```
>> m = max(max(T))
  m =
     9
```

2.6.3 Fonctions statistiques - `mean` et `cov`

- Appliquée à une liste ou un vecteur, la fonction `mean` détermine la moyenne des élément de la liste ou du vecteur.
- Appliquée à une liste ou un vecteur, `cov` détermine la variance des élément de la liste ou du vecteur.

Exemple 2.6.5 :

```
>> s = [5 2 3 1 7] ; m = mean(s)
  m =
    3.6000
>> c = cov(s)
  c =
    5.8000
```

- Appliquée à un tableau la fonction `mean` retourne la liste des moyennes des éléments de chaque colonne.
- Appliquée à un tableau où chaque ligne représente une observation et où chaque colonne correspond à une variable, la fonction `cov` retourne la matrice de covariance des éléments du tableau.

Exemple 2.6.6 :

```
>>T = pascal(3) , m = mean(t)
  T =
     1     1     1
     1     2     3
     1     3     6
  m =
    1.0000    2.0000    3.3333
```

On obtient la moyenne des éléments du tableau par :

```
>> m = mean(mean(T))
  m =
     2.1111
```

Pour la covariance :

```
 >>C = cov(T)
  C =
      0         0         0
      0    1.0000    2.5000
      0    2.5000    6.3333
```

Pour obtenir la variance de chaque colonne sous forme de vecteur-ligne :

```
>> c = diag(cov(T))'
  c =
      0    1.0000    6.3333
```

2.6.4 Fonctions `abs`

La fonction `abs` (nous le verrons d'une façon plus détaillée dans le prochain chapitre) appliquée a un tableau retourne un tableau de même format dont les éléments sont les valeurs absolues des éléments du tableau argument.

Exemple 2.6.7 :

```
>>
  T =
       1     -1      1
      -1     -2      3
       1      3     -6

>> U = abs(T)
    U =
       1      1      1
       1      2      3
       1      3      6
```

2.6.5 Norme matricielle d'un tableau - `norm`

Tout comme les normes vectorielles le font pour les vecteurs ou les listes, les normes "matricielles" donnent une mesure de la taille des éléments d'un tableau. La fonction `norm` permet le calcul des normes matricielles usuelles d'un tableau. Si `T` désigne un tableau de format (m, n) :

- `norm(T)` retourne la plus grande valeur propre de TT^T ;
- `norm(T,1)` $= \max_k \sum_{l=1}^{m} |T(l,k)|$ = `max(sum(abs(T)))`
- `norm(T, 2) = norm(T)`
- `norm(T, 'inf')` $= \max_l \sum_{k=1}^{n} |T(l,k)|$ = `max(sum(abs(T')))`
- `norm(T,'fro')` $= \sqrt{\sum_{k=1}^{n}\sum_{l=1}^{m} T(l,k)^2}$ = `sqrt(sum(diag(T'*T)))` (norme de Frobenius).

Dans les définitions ci-dessus, dans `T'*T` par exemple, `*` est l'opérateur MATLAB du produit matriciel.

2.6.6 Réorganisation des éléments d'un tableau - `reshape` et `sort`

Soit `T` un tableau de format (m, n). Si $m \times n = p \times q$, `reshape(T,p,q)` retourne un tableau de format (p, q) dont les éléments sont pris dans `T` en le parcourant colonne par colonne ; si $m \times n \neq p \times q$, `reshape(T,p,q)` retourne une erreur.

Exemple 2.6.8 :

```
>>  T =
    1    4    7    10
    2    5    8    11
    3    6    9    12

>> U = reshape(A,2,6)
  U =
    1    3    5    7     9   11
    2    4    6    8   10   12
```

Si `u` est un vecteur ou une ligne, la fonction `sort` ordonne les éléments de `u` par ordre croissant :

Exemple 2.6.9 :

```
>>  u = [4  2  1  7  3 ] ;

>> c = sort(u))
  c =
    1    2    3    4   7
```

Si `T` est un tableau, la fonction `sort(T, dim)` ordonne par ordre croissant :
- les colonnes de `T` si la variable `dim` a la valeur 1 ;
- les lignes de `T` si la variable `dim` a la valeur 1.

Exemple 2.6.10 :

```
>>  T = [3  1  7 ; 2   8   5]
 T =
    3    1    7
    2    8    5
>> U = sort(T, 1) % on ordonne le tableau T selon les colonnes
  U =
    2    1    5
    3    8    7
>> V= sort(T, 2)  % on ordonne le tableau T selon les lignes
  V =
    1    3    7
    2    5    8
```

La fonction `issorted` teste si une colonne, une ligne (plus généralement un vecteur ou une ligne) est ordonnée :

```
>> issorted(U(: , 1))
ans =
     1      % la première colonne de U est ordonnée

>> issorted(U(2 , :))
ans =
     0      % la deuxième ligne de U n'est pas ordonnée
```

3

Expressions, scripts et fonctions

3.1 Introduction

Un des avantages de MATLAB est de proposer une syntaxe très simple pour traduire les calculs matriciels. Les opérateurs sont représentés par les mêmes symboles (à une ou deux exceptions prés) que ceux utilisés communément en algèbre linéaire. Mieux, ils opèrent directement sur les tableaux (par exemple, il n'est pas nécessaire d'écrire des boucles pour effectuer la somme ou ou le produit de deux matrices).

3.2 Opérations de MATLAB

3.2.1 Opérateurs

Classés par ordre de priorité décroissante, les opérateurs utilisés par MATLAB sont les suivants :

- exponentiation et transposition
  - l'exponentiation `^` et `.^`
  - la conjugaison `'` et la transposition `.'`
- opérateurs multiplicatifs
  - les produits `*` et `.*`,
  - les divisions à droite `/`, `./` et à gauche `\`, `.\`
- opérateurs additifs et de négation
  - les opérateurs additifs unaire et binaires `+` et `-`
  - la négation `~`
- opérateurs booléens avec par ordre de priorité :
  - les opérateurs de comparaison
    - `<`, `>`, `<=` et `>=`
    - égalité `==`, non égalité `~=`
  - puis les opérateurs logiques
    - et logique `&`
    - ou logique `|`
    - et logique court-circuité `&&`
    - ou logique court-circuité `||`

Les opérateurs logiques **court-circuités** sont des opérateurs pour lesquels le résultat est acquis dès que l'évaluation du premier opérande permet d'affirmer le résultat global sans évaluer le second opérande. Si `a` et `b` sont des variables booléennes :

- `a && b` vaut `false` dès que `a` vaut `false`
- `a || b` vaut `true` dès que `a` vaut `true`.

D'autre part on rappelle qu'il n'existe pas dans MATLAB de "vrai" type booléen : `false` est représenté par la valeur 0 et que `true` est représentée par la valeur 1 et par extension par toute valeur non nulle, ce qui est confirmé par l'évaluation des expressions suivantes :

Exemple 3.2.1 :

```
>> 2&3
  ans =
     1      % true ET true vaut true
>> 2&0
  ans =
     0      % true ET false vaut false
>> 2|3
  ans =
     1      % true OU true vaut true
>> ~3
  ans =
     0      % le contraire de true (= 3) est false
>> 2==3
  ans =
     0      % l'égalité est bien celle des nombres et non celle des prédicats !
```

3.2.2 Opérateurs et opérations sur les tableaux

Lorsqu'ils sont appliqués à des nombres ou à des expressions booléennes, le résultat fourni par les opérateurs décrits ci-dessous, est le résultat usuel. Ils s'appliquent encore à des listes, des vecteurs ou plus généralement des tableaux ; on dit qu'ils sont *vectorisés*. Le résultat est bien sûr quelque peu différent.
Dans le tableau suivant, A et B sont des tableaux et c est un nombre :

Opérateur	Résultat	Conditions
$A+B$	tableau dont les éléments sont définis par $a_{ij}+b_{ij}$	A et B même format
$A+c=c+A$	tableau dont les éléments sont définis par $a_{ij}+c$	
$A-B$	tableau dont les éléments ont pour valeur $a_{ij}-b_{ij}$	A et B même format
$A-c$	tableau dont les éléments ont pour valeur $a_{ij}-c$	
$c-A$	tableau dont les éléments ont pour valeur $c-a_{ij}$	
$A*B$	tableau résultant du produit matriciel de A par B	nb col. A = nb lign. B
$A*c=c*A$	tableau dont les éléments ont pour valeur $c*a_{ij}$	
$A.*B$	tableau dont les éléments ont pour valeur $a_{ij}*b_{ij}$	A et B même format
$A\hat{}n$ $(n>0)$	$A*A*\cdots*A$ (n fois)	A carrée
$A\hat{}n$ $(n<0)$	$A^{-1}*A^{-1}*\cdots*A^{-1}$ ($\lvert n\rvert$ fois)	A inversible
$A.\hat{}B$	tableau dont les éléments ont pour valeur $(a_{ij})^{b_{ij}}$	A et B même format
A'	transposé-conjuguée du tableau A, $a'_{ij}=\overline{a_{ji}}$	
$A.'$	transposé du tableau A, $a'_{ij}=a_{ji}$ si tous les éléments de A sont réels, $A.'=A'$	
B/A	tableau X solution de l'équation matricielle $XA=B$ si A est inversible $X=BA^{-1}$	nb col. A = nb col. B
$A\backslash B$	tableau X solution de l'équation matricielle $AX=B$ si A est inversible $X=A^{-1}B$	nb lign. A = nb lign. B
$A./B$	tableau dont les éléments ont pour valeur a_{ij}/b_{ij}	A et B même format
$A.\backslash B$	tableau dont les éléments ont pour valeur b_{ij}/a_{ij} $A.\backslash B=B./A$	A et B même format
A/c	tableau dont les éléments ont pour valeur a_{ij}/c	

On notera qu'à certains opérateurs est **associé un opérateur pointé**, `*` et `.*` par exemple. De façon générale, l'opérateur pointé correspond à une opération semblable à celle représentée par l'opérateur non pointé, mais appliquée non pas "matriciellement" mais "terme à terme" : `*` est l'opérateur matriciel alors que `.*` est l'opérateur du produit "terme à terme".

Exemple 3.2.2 :

On crée deux matrices `A` et `B`. La fonction `ones(n)` crée une matrice carrée d'ordre 2 dont tous les termes sont égaux à 1 ; la fonction `eye(2)` crée la matrice identité d'ordre 2.

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

```
>> B = ones(2)+eye(2)
B =
  2    1
  1    2

>> C = A*B
C =
   4     5
   2     1

>> D = A.*B
D =
  2    2
  1    0
```

Notez bien la différence entre l'opération matricielle $*$ et l'opération "terme à terme" $.*$.

3.2.3 Opérations booléennes - Tableaux booléens

Dans la suite, **"tableau booléen"** ou *logical array* désignera un tableau dont les éléments ont pour valeur 0 ou 1, 0 représentant *false* et 1, représentant *true*. La fonction `islogical` teste le caractère booléen d'un tableau.

Opérateurs de comparaison

Les opérateurs booléens sont peut-être ceux dont le comportement peut apparaître le plus déroutant parce qu'il fonctionnent "terme à terme " et que le résultat est un tableau booléen :

Exemple 3.2.3 :

Avec les variables `A` et `B` définies dans l'exemple précédent on obtient :

```
 >> A , B
 A =
   1    2
   1    0
 B =
   2    1
   1    2

>> C =(A == B)
 C =              % C est le tableau resultant d'une opération booléenne
   0     0
   1     0

>> islogical(C)
 ans =
   1            % C est bien un tableau booléen
```

Dans l'expression `a == b` la comparaison porte sur les éléments homologues de `A` et `B` :
$1 == 3 \rightarrow false \rightarrow 0$, $2 == 1 \rightarrow false \rightarrow 0$
$1 == 1 \rightarrow true \rightarrow 1$, $0 == 2 \rightarrow false \rightarrow 0$
Il en est de même pour `A > B` :

```
>> A > B
 ans =
   0     1
   1     0
```

En résumé, si A et B sont des **tableaux de même format** et si `op` désigne un des opérateurs booléens `<`, `>`, `<=`, `>=`, `==`, `=` le résultat de A *op* B est défini par la règle suivante :

Opérateur	Résultat	Condition
A `op` B	“tableau booléen" dont les éléments sont définis par A_{ij} `op` B_{ij}	même format

Ainsi que le montre l'exemple ci-dessous, un tableau formé de 0 et de 1 n'est pas nécessairement un tableau booléen :

Exemple 3.2.4 :

```
>> A = eye(2)
 A =
   1     0
   0     1
>> islogical(A)
 ans =
   0           % A n'est pas un tableau booléen
>> class(A)
 ans =
   double
```

Seuls les tableaux
- résultant d'une opération booléenne ;
- formés de 0 et de 1 et transformés en tableaux booléens par la fonction `logical` ;

sont des tableaux booléens.

Exemple 3.2.5 :

Avec le même tableau `A` que précédemment :

```
>> B = logical(A)
 B =
   1     0
   0     1
>> test = islogical(B)
 test =
   1             % B est un tableau booléen
```

Opérateurs logiques

Soient `A` et `B` deux tableaux de même format, évaluons `A & B` :

Exemple 3.2.6 :

```
 >> A = [1 2 ; 1 0] , B = [2 1 ; 1 0]
 A =
   1    2
   1    0
```

```
 B =
   2    1
   1    2

>> A & B
ans =
  1     1
  1     0

>> islogical(ans)
 ans =
   1
```

MATLAB évalue sans problème `A & B` bien que les tableaux `A` et `B` ne soient pas des tableaux booléens. Le résultat de l'évaluation `ans`, est un tableau booléen : avant l'évaluation MATLAB a transformé les deux tableaux en tableaux booléens en suivant la règle usuelle, puis a effectué l'opération logique terme à terme.
Il en est de même pour les autres opérateurs logiques `|, || , &&`.

3.2.4 Évaluation des expressions - `ans`

Les expressions sont évaluées de la gauche vers la droite, suivant l'ordre de priorité indiqué plus haut. Pour des opérateurs de même ordre de priorité, la règle est comme en mathématiques, celle de l'associativité à gauche.
La frappe de <`entrer`> déclenche l'évaluation. La valeur de expression évaluée est affichée sous la ligne courante et en l'absence d'affectation explicite, elle est affectée à une variable-système générique désignée par `ans` pour *answer*.

Exemple 3.2.7 :

```
>> a = .5
a =
  0.5000

>> a*pi
ans =
    1.5708

>> b = 2*ans  % affectation explicite de l'evaluation à b
b =
    3.1416

>> ans
ans =
    1.5708
```

La dernière évaluation n'a pas modifié la valeur de `ans` puisqu'elle comportait une affectation explicite à la variable `b`.

3.3 Scripts et *m-files*

3.3.1 Scripts

Un script est une séquence d'expressions ou de commandes. Un script peut se développer sur une ou plusieurs lignes. Les différentes expressions ou commandes doivent être séparées par une virgule, un point-virgule ou par le symbole de saut de ligne qui en ligne de commande est constitué du symbole de continuation `...` suivis de <`entrer`> (la présence du symbole de continuation inhibe le mécanisme d'évaluation déclenché par la frappe de <`entrer`>).
Les expressions sont évaluées dans leur ordre d'écriture. Seule la valeur des expression suivie d'une virgule ou d'un saut de ligne est affichée, celle des expressions suivies d'un point-virgule, ne l'est pas.

Exemple 3.3.1 :

```
>> a = .5, 2*a, save a, b = pi; 2*b, c = a*b
a =
    0.5000
c =
    1.5708
ans =
    6.2832
c =1
    .5708

>> ans
ans =
    6.2832
```

Écrire un script est assez fastidieux, aussi MATLAB permet d'enregistrer le texte d'un script sous forme de fichier de texte appelés *m-files*, à ne pas confondre avec les emphmat-files que nous avons évoqués dans le chapitre précédant et qui sont des fichiers binaires permettant d'enregistrer des valeurs.

3.3.2 Création de *m-files*

Les *m-files* permettent d'enregistrer les scripts sous forme de fichiers-texte et servent en particulier à définir de nouvelles fonctions (une grande partie des fonctions prédéfinies de MATLAB sont stockées sous forme de *m-files* dans la `toolbox matlab`.

Les *m-files* peuvent être crées par n'importe quel éditeur. Dans les versions récentes de MATLAB il existe un petit éditeur intégré que l'on peut appeler à partir du menu `file` ou à partir de la barre de menu de la fenêtre de commande.

Exemple 3.3.2 :

Dans la fenêtre de l'éditeur tapez les lignes suivantes :

```
% script - essai . m
a = .5;
b = pi;
c = a * b
```

Sauvez le fichier dans le répertoire de travail sous le nom de `essai.m`.

!!! **Remarque :**

On peux utiliser les techniques du copier/coller pour transférer des parties de script de la fenêtre de commande de MATLAB vers l'éditeur et réciproquement. Il faut prendre garde au fait que dans la fenêtre de commande de MATLAB les sauts de ligne <`entrer`> lancent l'évaluation des expressions ; il faut alors faire précéder les sauts de ligne du symbole de continuation `...`

3.3.3 Exécution d'un *m-file*

Pour exécuter le script contenu dans un *m-file* et Il suffit de taper le nom de ce `m-file` dans la fenêtre de commande suivi de < *entrer* >

Exemple 3.3.3 :

Pour exécuter le script précédent, on tape `essai` et on obtient :

```
>> essai
c =
    1.5708
```

La présence d'un point-virgule ; à la fin des deux premières lignes du script a neutralisé l'affichage des valeurs de a et b.

3.3.4 Éléments d'écriture de *m-files*

Commentaires

Les lignes de commentaires sont précédées du caractère %.

Entrées - `input` et `menu`

La fonction `input` permet la saisie d'une valeur depuis le clavier. Plus précisément :
- Pour les valeurs numériques, *n* `= input('message')` affiche *message* et affecte à la variable *n* la valeur numérique entrée au clavier.
- Pour les chaînes de caractères, *str* `= input('message','s')` affiche *message* et affecte à la variable *str* la valeur entrée au clavier considérée alors comme une chaîne de caractères.

Exemple 3.3.4 :

```
>> n = input('Entrez la valeur de n ')
>> nom = input('Entrez votre nom ','s')
```

La fonction `menu` génère une fenêtre contenant un menu dans lequel l'utilisateur doit choisir une option :

result `= menu('titre', 'opt1', 'opt2', ..., 'optn')`

La valeur retournée dans la variable *result* est égale au numéro de l'option choisie ; `menu` est souvent utilisé en relation avec la structure algorithmique `switch-case` *c.f. 3.4.4.*

Exemple 3.3.5 :

```
result = menu('Traitement', 'Gauss', 'Gauss-Jordan', 'Quitter')
```

Sans la fenêtre ci-après, si l'utilisateur sélectionne *Gauss*, la variable `result` prend la valeur 1, la valeur 2 s'il sélectionne *Gauss-Jordan* et la valeur 3 s'il sélectionne *Quitter*.

Affichages - `disp` - `num2str` - `format`

La valeur d'une variable est très simplement affichée en faisant évaluer une expression réduite à la variable elle-même.

Exemple 3.3.6 :

```
>> a = [1 2] ;

>> a
a =
  1    2
```

La fonction `num2str`(x) où x est un nombre, retourne la valeur littérale de ce nombre.

Exemple 3.3.7 :

```
>> s = ['la valeur de pi est : ' num2str(pi)] ;

>> s
s =
   la valeur de pi est : 3.1416
```

La commande `disp`(t) où t est une chaîne de caractères ou un tableau, affiche la valeur de cette chaîne de caractère ou de ce tableau sans faire référence au nom de la variable. Cette commande sera souvent utilisée avec `num2str` pour afficher les valeurs des expressions numériques.

Exemple 3.3.8 :

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

>> disp(a)
  1    2
  3    4

>> disp([ 'ordre de la matrice a : '  num2str(size(a,1)) ] );
  ordre de la matrice a : 2
```

La commande `format` permet de choisir entre plusieurs modes d'affichage (sans interférer avec le `type` des valeurs numériques affichées qui est toujours le type `double`) :

Commande	Affichage	Exemple
`format short`	décimal à 5 chiffres	`31.416`
`format short e`	scientifique à 5 chiffres	`31.416e+01`
`format long`	décimal à 16 chiffres	`31.4159265358979`
`format long e`	scientifique à 16 chiffres	`314159265358979e+01`
`format hex`	hexadécimal	
`format bank`	virgule fixe à deux décimales	`31.41`
`format rat`	fractionnaire	`3550/113`
`format +`	utilise les symboles +, - et espace pour afficher les nombres positifs négatifs et nuls	+

Commande `pause`

La commande `pause` permet de ménager une pause dans l'exécution d'un *m file* :
- sans argument `pause` suspend l'exécution jusqu'à ce que l'utilisateur presse sur une touche.
- `pause(`*n*`)` suspend l'exécution pendant *n* secondes.

Interaction avec le système d'exploitation

MATLAB possède des fonctions et des commandes qui permettent d'obtenir la liste des répertoires accessibles ou `matlabpath`, la liste des fichiers d'un répertoire donné, les éditer et éventuellement les effacer :
- `addpath` *path* : ajoute *path* à la liste `matlabpath` des répertoires accessibles par MATLAB ;
- *p* `= pwd` : retourne dans *p* le chemin d'accès au répertoire de travail actuel ;
- `cd` *path* : change le répertoire de travail pour celui spécifié par *path* ;
- *d* `= dir` ou *d* `= ls` : retourne dans *d* la liste des fichiers du répertoire de travail ;
- `what` : retourne la liste des *m-files* et des `mat-files` du répertoire de travail ;
- `edit` *test* : édite le *m-file test.m*, identique à `Open` dans le menu `File` ;
- `delete` *test.m* : efface le *m-file test.m* ;
- `type` *test* : affiche le le *m-file test.m* dans la fenêtre de commande.

3.4 Structures algorithmiques

3.4.1 Sélection - `if...end` et `if...else...end`

Syntaxe

- `if` *(expression booléenne)* `/` *script* `/ end`
- `if` *(expression booléenne)* `/` *script si vrai* `/ else /` *script si faux* `/ end`

Le symbole `/` remplace l'un des symboles séparateur : virgule (`,`), point-virgule (`;`) ou saut de ligne. L'usage du point-virgule est vivement conseillé pour éviter les affichages souvent redondants. Dans d'anciennes versions de MATLAB et en mode commande, il est indispensable de faire précéder < *enter* > par le symbole de continuation constitué d'une séquence de trois points (`...`) pour passer à la ligne sans déclencher le processus d'évaluation.

Exemple 3.4.1 :

```
>> m = -1;
>> if (m<0) a =-m,  end
a =
     1
```

Cette autre présentation est plus lisible :

```
>> f (m<0)
      a = -m ;
   end
```

Utilisation de `elseif`

Lorsqu'il y a plus de deux alternatives, on peut utiliser la structure suivante :

`if` *(exp1)*
script1 (évalué si exp 1 est vraie)
`elseif` *(exp2)*
script2 (évalué si exp 2 est vraie)
`elseif` *(exp3)*
...
`else`
script (évalué si aucune des expressions exp1, exp2, ... n'est vraie)
`end`

3.4.2 Répétition - `for...end`

Syntaxe

`for` *(k = liste)* `/` *script* `/` `end`

Le symbole `/` représente comme dans le paragraphe précédent un symbole séparateur virgule (`,`), point-virgule (`;`) ou saut de ligne. D'autre part, il est préférable de ne pas utiliser i et j comme indices puisque dans MATLAB i et j sont des variables prédéfinies dont la valeur est $\sqrt{(}-1)$.

Exemple 3.4.2 :

```
>> x = [ ] ;
>> for (k = 1 : 5), x = [x, sqrt(k)],  end
x =
     1
x =
    1.0000    1.4142
x =
    1.0000    1.4142    1.7321
x =
    1.0000    1.4142    1.7321    2.0000
x =
    1.0000    1.4142    1.7321    2.0000    2.2361
```

cette autre présentation plus lisible, doit être privilégiée :

```
>> for (k = 1 : 5)
      x = [x, sqrt(k)] ;
   end
```

Boucles `for` emboîtés

Exemple 3.4.3 :

```
>> for (l = 1 : 3)
       for ( k = 1 : 3)
          A(l,k) = l^2 + k^2 ;
       end
    end

>> disp(A)
     2     5    10
     5     8    13
    10    13    18
```

Utilisation de `break` et de `continue`

Il est possible de sortir directement d'une boucle `for` ou `while` en utilisant la commande `break` :

Exemple 3.4.4 :

```
>> epsi = 1;

>> for (n = 1 : 100)
      epsi = epsi / 2;
      If ((epsi + 1) <= 1)
        epsi = epsi*2
        break
      end
    end

epsi =
   2.2204e-16

>> n
n =
    52
```

Le test `(epsi + 1) <= 1` provoque la sortie de la boucle `for` à la $52^{ième}$ itération.
Dans l'exemple suivant, le tableau `A` est celui de l'exemple précédent.

```
>> for (l = 1 : 3)
      for (k = 1 : 3)
        if (A(l,k) == 10)
          [l,k]
          break
          end
       end
    end
ans =
     1     3
```

```
ans =
     3     1
```

La double boucle se s'est pas arrêté après que le test `A(l,k) == 10` ait été validé lorsque `l=1` et `k=3`. En effet `break` **provoque la sortie de la boucle la plus proche**, ici la boucle `for` interne. Une version corrigée du test précédent pourrait être la suivante avec deux `break` pour pouvoir sortir des deux boucles `for` :

Exemple 3.4.5 :

```
>> sortie = 0;

>> for (l=1:3)
      if (sortie)
         break
      end
      for (k = 1:3)
        if (A(l,k) == 10)
          [l,k]
          sortie = 1 ;
          break
        end
      end
    end

ans =
     1     3
```

La commande `continue` interromps l'exécution du corps de la boucle et provoque le passage à l'itération suivante.

3.4.3 Itération conditionnelle - `while...end`

Syntaxe

`while` *(expression booléenne)* `/` *script* `/` `end`

Le symbole `/` représente comme dans les définitions précédents, un séparateur : virgule (`,`), point-virgule (`;`) ou saut de ligne. D'autre part, il faut éviter l'utilisation des variables i et j comme indices puisqu'elles sont des variables prédéfinies dont la valeur est $\sqrt{(}-1)$.

Exemple 3.4.6 :

```
>> n = 1 ;

>> while (2^n <= 100)
      n = n + 1;
    end

>> disp(n-1)
    6
```

3.4.4 Construction `switch...case`

Syntaxe

`switch` *(sélecteur)*
  `case` *valeur 1, ...* `/` *script 1* `/`
  `case` *Valeur 2, ...* `/` *script 2* `/`
  ...
  `otherwise` `/` *script*
`end`

Comme dans les définitions précédentes, le symbole `/` remplace un séparateur : virgule (`,`), point-virgule (`;`) ou saut de ligne.
sélecteur désigne une expression dont les valeurs peuvent correspondre aux valeurs associées aux différents `case`. Lorsque la valeur du sélecteur correspond à une valeur de `case`, une fois le script correspondant exécuté, l'exécution se continue immédiatement après le `end` contrairement à ce qui se passe pour certains langages. Ceci explique l'absence de `break` après chaque script.

Exemple 3.4.7 :

```
>> n =17

>> switch rem(n,3)  % reste de la division de n par 3
      case 0, disp('Multiple de 3')
      case 1, disp('1 modulo 3')
      case 2, disp('2 modulo 3')
      otherwise, disp('Nombre négatif')
    end

2 modulo 3
```

3.4.5 Traitement des erreurs - `try...catch...end`

La commande `try...catch` a pour but de permettre un traitement qui permette à l'utilisateur d'intervenir en présence d'erreurs ou de situations inhabituelles. Sa syntaxe est la suivante :

`try` *script1* `catch` *script2* `end`

Le fonctionnement en est assez simple pour les habitués des langages modernes, `java` par exemple :
- l'exécution de *script1* est lancée ;
- si une erreur survient, alors l'exécution de *script1* est arrêtée et *script2* est exécuté ;
- sinon, *script1* est exécuté jusqu'à son terme, *script2* n'est pas exécuté, les exécutions suivantes se poursuivent après le `end` final.

On peut utiliser `lasterr` pour accéder à l'erreur qui provoque l'arrêt de l'exécution de *script1.*

3.5 Fonctions

A coté des fonctions prédéfinies *c.f.4.1*, MATLAB offre à l'utilisateur la possibilité de définir ses propres fonctions. Trois méthodes sont possibles : les *m-fonctions* qui sont associées à des *m-files* auxiliaires, les fonctions anonymes et les fonction `Inline`. Ces deux derniers types de fonction peuvent être définies directement dans l'espace de travail où ces fonctions vont être utilisées.

3.5.1 *m-Fonctions*

Ces fonctions sont associées à un *m-file* auxiliaire, ce qui apporte deux avantages : la fonction possède son propre espace de travail (pas d'effet de bord) et il n'y a pas de limite à la complexité de la séquence de commandes qui définit la fonction.

Exemple 3.5.1 :

La *m-fonction* `moyenne` définie ci-dessous calcule la moyenne des éléments d'une liste ou d'un vecteur.
Saisissez le texte ci-après dans un éditeur (par exemple l'éditeur intégré à MATLAB mais tout éditeur fait l'affaire) et enregistrez le sous le nom `moyenne.m`.

```
1 function m = moyenne(x)
2 % MOYENNE(X) : moyenne des éléments d'une liste ou d'un vecteur
3 % un argument autre qu'une liste ou un vecteur conduit a une erreur
4 [k,l] = size(x) ;
5  if ( (k~=1) & (l~=1) )
6     error('l"argument doit être une liste ou un vecteur')
7  end
8 m = sum(x)/length(x) ;
9 return
```

On appelle la fonction `moyenne` depuis le fenêtre de commande par :

```
>> résultat =  moyenne(1 : 9)
resultat =
    5

>> A = [ 1 2 ; 3 4] ; moyenne(A)
??? Error using ==> moyenne
  l"argument doit être une liste ou un vecteur
```

Le traitement de l'erreur sur la nature de l'argument a été réalisé aux lignes 5 à 7 et par la commande `error`.
MATLAB utilise la section de commentaires des lignes 2 et 3 en conjonction avec la commande `help` pour fournir des informations sur la fonction :

Exemple 3.5.2 :

```
>> help moyenne
  MOYENNE(X) : moyenne des éléments d'une liste ou d'un vecteur
 un argument autre qu'une liste ou un vecteur conduit a une erreur
```

Syntaxe

Une fonction est constituée par :
- un en-tête qui si la fonction renvoie une seule valeur, a la forme suivante :
  function *résult = nom-de-fonction (liste de paramètres)*
  ou si la fonction renvoie plusieurs valeurs :
  function *[résult1, result2, ...] = nom-de-fonction (liste de paramètres)*
- une section de commentaires (9 lignes au plus, débutant par le symbole %) utilisée par les commande `help` ou `lookfor` ;
- le corps de la fonction défini par un script ;
- le `return` terminal est n'est pas obligatoire ; il est conseillé car il entraine un retour "propre" à la fonction ou au script appelant ou par défaut à la fenêtre de commande.

Règles et propriétés

- Le nom de la fonction *nom-de-fonction* est un identificateur construit conformément aux règles définies au paragraphe
- Le nom de la fonction et celui du fichier *m-file* qui en contient la définition **doivent être identiques**. Ce fichier est le **fichier *m-file* associé à la fonction**.
- Chaque fonction possède son **propre espace de travail** et toute variable apparaissant dans le corps d'une fonction est locale à celle-ci, à moins qu'elle ait été déclarée comme **globale** au moyen du qualificateur `global` précédant le nom de la variable **dans tous les espaces de travail où cette variable est utilisée**.
- Il est préférable que les fonctions soient *vectorisées* comme le sont les fonctions prédéfinies. Il faut alors n'utiliser que des opérateurs pointés dans la définition de la fonction.
- L'exécution d'une fonction s'achève :
  - lorsque la fin du script définissant la fonction a été atteint ;
  - lorsque une commande `return` ou un appel de la fonction `error` a été rencontré :
    - `return` termine immédiatement l'exécution de la fonction sans que la fin du script définissant celle-ci ait été atteint,
    - `error('message')` procède de même, mais en plus, affiche le contenu de *'message'*.

  Le contrôle est alors renvoyé au point d'appel de la fonction, fenêtre de commande ou autre fonction.
- Le fichier *m-file* associé à une fonction peut contenir d'autres définitions de fonctions. La fonction qui partage son nom avec le fichier ou **fonction principale** doit apparaître en premier. Les autres fonctions ou **fonctions internes** peuvent être appelées par la fonction principale, mais pas par d'autres fonctions ou depuis la fenêtre de commande.

Exemple 3.5.3 :

```
1  function [m, v] = myStat(x)
2  % MYSTAT(X) : moyenne et variance des elements d'une liste ou d'un vecteur
3  % un argument autre qu'une liste ou un vecteur conduit a une erreur
4  [k,l] = size(x) ;
5   if ( (k~=1) & (l~=1) )
6      error('l"argument doit être une liste ou un vecteur')
7   end
8  m = moyenne(x);
9  v = variance(x);
10 return

11 function a = moyenne(u)
12 % Calcul de la moyenne
13 a = sum(u)/length(u);s

14 function b = variance(u)
15 % Calcul de la variance
16 c = sum(u)/length(u);
17 u2 = (u - c).^2;
18 b = sum(u2)/length(u);
```

L'ensemble des trois fonctions est enregistré dans un seul fichier *m-file* portant le nom de la fonction principale `myStat.m`.

3.5.2 Fonctions `Inline`

Lorsque le corps de la fonction se résume à une expression relativement simple, on peut créer la fonction directement dans l'espace de travail courant, sans utiliser un *m-file* auxiliaire .

Syntaxe

La syntaxe des fonctions `Inline` est simple :

```
nom-de-fonction = inline ('expression', 'var1', 'var2', ...)
```

- L'expression mathématique qui constitue le corps de la fonction ainsi que les variables sont considérées par MATLAB comme des chaines de caractères et **doivent donc être tapées entre apostrophes**.
- Il est préférable que les fonctions soient *vectorisées* comme le sont les fonctions prédéfinies. Il faut alors n'utiliser que des opérateurs pointés dans l'expression définissant la fonction..
- La déclaration des variables peut être optionnelle dans la définition des fonctions `Inline`, MATLAB effectuant une déclaration implicite de celles-ci. Cette facilité, source d'ambiguïtés dans le cas de fonctions de plusieurs variables n'est pas à recommander dans ce cas.

Exemple 3.5.4 :

```
>> f = inline('x.^2 + x.*y', 'x', 'y')
f =
     Inline fonction :
     f(x, y) = x.^2 + x.*y

>> f(1, 2)
ans =
       3.0000

>> f([1 0], [2 1])
ans =
       3.0000    0.0000
```

L'exemple suivant met en évidence le mécanisme de déclaration implicite.

Exemple 3.5.5 :

La définition

```
>> f = inline('x.^2')
f =
     Inline fonction :
     f(x) = x.^2
```

est équivalente à :

```
>> f = inline('x.^2', 'x')
```

Ce mécanisme est ambigu dès qu'il y a plusieurs variables : la définition

```
>> f = inline('x.^2 + x.*y')
```

est équivalente à :

```
>> f = inline('x.^2 + x.*y', 'x', 'y')
```

alors que

```
>> f = inline('x.^2 + x.*t')
```

est équivalente à :

```
>> f = inline('x.^2 + x.*t', 't', 'x')
```

3.5.3 Fonctions anonymes

Ce troisième mode de définition de fonctions utilise comme pour les fonctions `Inline` l'espace de travail courant. La syntaxe minimale est peu explicite, mais les fonctions ainsi définies seraient plus efficaces que fonctions `Inline`.

Syntaxe

nom-de-fonction = `@(`*var1, var2, ...*`)` *expression*

- Contrairement aux fonctions `Inline` l'expression mathématique qui constitue le corps de la fonction ainsi que les variables **ne doivent pas** être tapées entre apostrophes.
- Il est préférable que les fonctions soient *vectorisées* comme le sont les fonctions prédéfinies. Il faut alors n'utiliser que des opérateurs pointés dans l'expression définissant la fonction.

Exemple 3.5.6 :

```
>> g = @(x, y) x.^2 + x.*y
g =
     @(x, y) x.^2 + x.*y

>> g(1, 2)
ans =
       3.0000

>> g([1 0], [2 1])
ans =
       3.0000    0.0000
```

3.5.4 Fonctions argument d'autres fonctions

Une façon simple d'utiliser une fonction comme argument d'une autre fonction est de transmettre à la fonction utilisatrice le *handle* de la fonction (le *handle* d'une fonction est la référence (l'adresse en mémoire) du code MATLAB qui définit le traitement effectué par la fonction).
Le *handle* d'une *m-fonction* ou d'une fonction prédéfinie de MATLAB est obtenu en faisant précéder le nom de la fonction par le symbole `@`. Par exemple, le *handle* de la fonction prédéfinie `exp` est `@exp`.
Le *handle* d'une fonction `Inline` ou d'une fonction anonyme est le nom avec lequel la fonction a été définie.

Dans ce premier exemple la fonction utilisatrice est définie comme une *m-fonction*.

Exemple 3.5.7 :

```
function y = trapeze(f, a, b)
% Input :    f : handle
%            a : borne inférieure
```

```
%            b : borne supérieure
% Output :   aire du trapezes
y = 0.5*(f(a) + f(b))*(b - a) ;
return

>> carre = inline('x.^2', 'x')
carre =
     Inline fonction :
     carre(x) = x.^2

>> trapeze(carre, 1, 2)
ans =
       2.5000
```

On peut également définir la fonction trapèze comme une fonction anonyme :

Exemple 3.5.8 :

```
>> trapeze2 = @(f, a, b)  0.5*(f(a) + f(b))*(b - a) ;

>> carre = @(t) t.^2 ;

>> trapeze2(carre2, 1, 2)
ans =
       2.5000
```

3.5.5 Commandes et fonctions `nargin` et `nargout`

La commande `nargin` qui s'utilise à l'intérieur du corps d'une fonction, donne le nombre de paramètres d'entrée effectivement passés lors l'appel de la fonction.

Exemple 3.5.9 :

```
fonction n = test(a, b)
if (nargin ==1)
    n = a ;
elseif (nargin == 2)
    n = a +b ;
end
```

La fonction `nargin`*`('nom-de-fonction')`* revoie le nombre prévu de paramètres d'entrée de la fonction dont le nom est passé en argument.

Exemple 3.5.10 :

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

La commande et la fonction `nargout` fonctionnent de même pour les paramètres de sortie.

3.6 Optimisation des calculs

Les calculs sont accélérés de façon spectaculaire en utilisant des opérations vectorielles en lieu et place de boucles. Comparons les deux fonctions suivantes (la commande `tic` déclenche un chronomètre ; `toc` arrête le chronomètre et retourne le temps écoulé depuis `tic`) :

Exemple 3.6.1 :

```
function [t,b] = test1(n)
% détermine le temps mis pour créer la liste
% des racines carrées des entiers compris entre 1 et n
  m = 0 ;
  tic ;
  for k = 1 : 1 : n
    b(k) = m+sqrt(k) ;
  end
  t = toc ;
  return

function [t,b] = test2(n)
% détermine le temps mis pour créer la liste
% des racines carrées des entiers compris entre 1 et n
  tic ;
  a = 1 : 1 : n ;
  b = sqrt(a) ;
  t = toc ;
  return
```

Les résultats obtenus montrent que `test2` est plus efficace que `test1`.

```
>>test1(1000)
  ans =
    0.1040

>>test2(1000)
  ans =
    0.0023
```

MATLAB dispose d'un utilitaire appelé **profiler** qui est précieux pour étudier les performances d'une ou plusieurs fonctions. Les modalités d'utilisation du profiler ont évolué en fonction des versions de MATLAB. On les trouvera dans l'aide en ligne `help` *`profile`*.

4

MATLAB et l'analyse numérique

Il est impossible dans un seul chapitre de faire le tour de toutes les fonctions de MATLAB liées à l'analyse numérique. Ce chapitre présente quatre familles de fonctions : les fonctions "numériques", les fonctions du calcul polynomial, les fonctions matricielles et les fonctions liées aux fonctions d'une variable.

4.1 Fonctions "numériques"

Les fonctions numériques de MATLAB généralisent les fonctions numériques usuelles, avec une différence cependant : comme les opérateurs définis en *3.2.2*, elles sont *vectorisées*. C'est à dire qu'elles s'appliquent aussi bien à des nombres qu'à des tableaux (ceci est assez normal puisqu'un nombre est un tableau particulier). Lorsqu'une de ces fonctions a pour argument un tableau, la fonction est appliquée à chacun des éléments du tableau. Le résultat obtenu est donc un tableau de même format que le tableau donné comme argument.

Exemple 4.1.1 :

```
>> t =[-3 2 0  ; -2 3 -1]
t =
    -3     2     0
    -2     3    -1
>> u = abs(t)
u =
     3     2     0
     2     3     1
>> v = exp(u)
v =
   20.0855    7.3891    1.0000
    7.3891   20.0855    2.7183
```

Les fonctions numériques sont nombreuses. L'aide en ligne en donne une liste exhaustive. Dans le tableau ci-dessous, on trouvera les fonctions les plus fréquemment utilisées.

Fonction	Résultat
`abs`	valeur absolue ou module (nb. complexes)
`angle`	argument (nb. complexes)
`real`	partie réelle (nb. complexes)
`imag`	partie imaginaire (nb. complexes)
`conj`	complexe conjugué
`sqrt`	racine carrée
`cos`	cosinus (angle en radian)
`sin`	sinus (angle en radian)
`tan`	tangente (angle en radian)
`acos`	arc cosinus (résultat en radian)
`asin`	arc sinus (résultat en radian)
`atan`	arc tangente (résultat en radian)
`exp`	exponentielle
`log`	logarithme népérien
`log10`	logarithme base 10
`round`	entier le plus proche
`fix`	partie entière mathématique
`floor`	troncature par défaut
`ceil`	troncature par excès

4.2 Polynômes

Pour MATLAB, un polynôme est une liste : la **liste des coefficients ordonnés par ordre décroissant** :

Exemple 4.2.1 :

Le polynôme $p(x) = 1 - 2x + 4x^3$ est représenté par le liste :

```
>> p = [4  0  -2  1]
 p =
   4     0     -2     1
```

Les fonctions usuelles du calcul polynomial sont les suivantes :

Fonction	Arguments	Résultat
`polyval`	un polynôme p et un nombre a	calcul de $p(a)$
`roots`	un polynôme p	la liste des racines de p
`conv`	deux polynômes p et q	le polynôme produit $p \times q$
`deconv`	deux polynômes p et q	le quotient et le reste de la division euclidienne de p par q
`polyder`	un polynôme p	le polynôme-dérivée de p

4.3 Calcul matriciel

Voici quelques-unes des fonctions usuelles du calcul matriciel :

Fonction	Résultat
`eig`	calcul des valeurs et des vecteurs propres
`poly`	polynôme caractéristique
`det`	calcul du déterminant
`trace`	calcul de la trace
`inv`	calcul de l'inverse

Exemple 4.3.1 :

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

>> vp = eig(a)
 vp =
   -0.3723
    5.3723

>> [p, d] = eig(a)
 p =                %  p est la matrice des vecteurs propres
   -0.8246   -0.4160
    0.5658   -0.9094

 d =                %  d est la matrice diagonalisée
   -0.3723         0
         0         5.37233

>> p = poly(a)
 p =
    1.0000   -5.0000   -2.0000
                        % le polynôme caractéristique est p(t) = t^2 - 5t - 2

>> roots(p)
 ans =
    5.3723
   -0.3723
                      % les racines de p sont bien les valeurs propres
```

4.4 Fonctions d'une variable

4.4.1 Recherche de minimum - `fmin`

La fonction `fmin` prend pour arguments le **nom de la fonction à étudier** écrite sous forme d'une chaîne de caractères, et les bornes inférieures et supérieures de l'intervalle d'étude. La fonction peut être une fonction prédéfinie de MATLAB ou une fonction définie par l'utilisateur, mais elle doit **impérativement être une fonction de la variable `x`** .

Exemple 4.4.1 :

```
>> xmin = fmin('cos', 3, 4) , ymin = cos(xmin)
 xmin =
    3.1416
 ymin =
   -0.7071
```

Lorsque la fonction est définie par une expression, on peut utiliser directement cette expression, écrite sous forme d'une chaîne de caractères, comme argument de `fmin` :

```
 >> fn = '2*exp(-x)*sin(x)' ; fmin(fn, 2, 5)
 xmin =
    3.9270
```

Pour calculer `ymin` on utilise la fonction `eval` qui prend comme argument une expression écrite sous forme de chaîne de caractère (comme cela est possible pour `fmin`) :

```
>> x = xmin ; ymin = eval(fn)
 ymin =
   -0.0279
```

Lorsque l'expression dont on veut calculer la valeur est définie par une fonction de MATLAB ou définie par l'utilisateur, on utilise `feval`

```
>> xmin = fmin('myFunct', 2, 4) , ymin = feval('myFunct', xmin)
 xmin =
    3.1416
 ymin =
   -0.7071
```

4.4.2 Recherche de racines - `fzero`

La syntaxe de la fonction `fzero` est voisine de celle de la fonction `fmin`. La fonction `fzero` prend pour arguments le nom de la fonction à étudier écrite sous forme d'une chaîne de caractères, et une valeur initiale voisine de celle d'une racine. La fonction peut être une fonction prédéfinie de MATLAB ou une fonction définie par l'utilisateur, mais elle doit impérativement être une fonction de la variable `x`.

Exemple 4.4.2 :

```
>> x0 = fzero('cos',1), y0 = cos(x0)
 x0 =
    1.5708
 y0 =
    0
```

Il n'est pas possible comme pour la fonction `fmin` de définir la fonction à étudier par une expression mathématique. On créera alors une fonction `m-file` pour définir cette fonction.

4.4.3 Intégration - `trapz`, `quad` et `quad8`

MATLAB propose plusieurs fonctions pour calculer numériquement la valeur de l'intégrale d'une fonction d'une variable, sur un intervalle fermé.

- `trapz` - La fonction `trapz` utilise la méthode des trapèzes. les arguments de `trapz` sont deux listes, dans l'ordre :
  - une liste `x` qui est une subdivision de l'intervalle d'intégration ;
  - une liste `y` dont les valeurs sont les images des valeurs de de la liste `x` par la fonction à intégrer (`y(k) = f(x(k))`).

  Exemple 4.4.3 :
  ```
  >> x = 0 : pi/100  : pi ; y = sin(x) ;

  >> z = trapz(x, y)
    z =
      1.9998
  ```

- `quad` et `quad8` - Ces deux fonctions sont fondées respectivement sur la méthode de *Simpson* et sur la méthode de *Newton-Cotes*. Leur syntaxe est celle de la fonction `fmin` *voir 4.4.1* :

  Exemple 4.4.4 :
  ```
  >> z = quad('sin', 0, pi)
    z =
      2.0000
  ```

5

Courbes et surfaces

5.1 Fenêtres graphiques

5.1.1 Création d'une fenêtre - fonctions `figure` et `gcf`

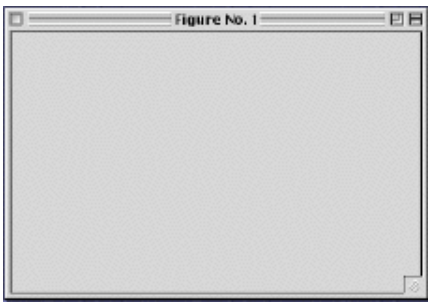
La fonction `figure` crée une fenêtre graphique vide

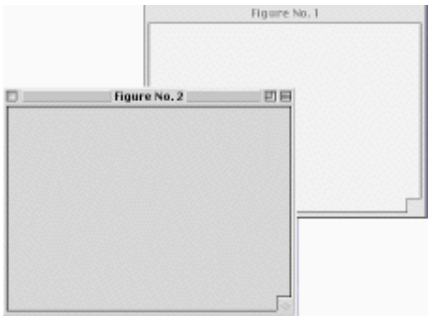
Exemple 5.1.1 :

```
>>  h = figure
 h =
    1
```

Une fenêtre appelée `Figure` $N°1$ apparaît. La valeur retournée par la fonction `figure` est le numéro de la fenêtre. Un second appel à la fonction `figure` crée une seconde fenêtre appelée, on s'en doute, `Figure` $N°2$.

```
>> h = figure
 h =
    2
```

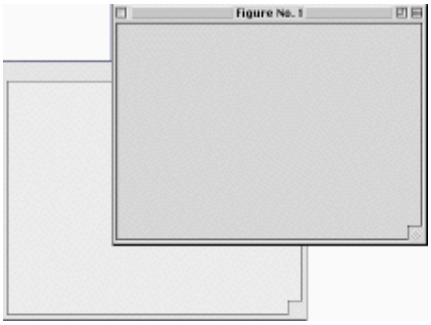




La dernière fenêtre crée est la fenêtre active (située au premier plan). Pour faire passer une fenêtre au premier plan, on utilise la fonction `figure` avec pour argument le numéro de la fenêtre que l'on souhaite activer. Si aucune fenêtre portant ce numéro n'existe elle sera créée.
Réciproquement, la fonction `gcf` *(**g**et **c**urrant **f**igure)* retourne le numéro (ou référence) de la fenêtre active.

Exemple 5.1.2 :

```
>> h = gcf
 h =
    2
>> figure(1)
>> h = gcf
 h =
    1
```



La fenêtre active est la `Figure` $N°2$, puis on rend active `Figure` $N°1$. Elle passe au premier plan (il est possible de faire la même chose en cliquant dans `Figure` $N°1$).

5.1.2 Attributs d'une fenêtre - `get`

(Ce paragraphe peut être laissé de côté en première lecture.)
Les fenêtres possèdent un grand nombre d'attributs, par exemple un nom (`Name`), une couleur de fond (`Color`), ... (consultez l'aide en ligne). On obtient la liste complète des **attributs de la fenêtre active** et de leur valeur, par `get(n)` où `n` est le numero de cette fenêtre.

Exemple 5.1.3 :

```
>> h = gcf ; get(h)
  BackingStore = on
  CloseRequestFcn = closereq
  Color = [0.8 0.8 0.8]
  Colormap = [ (64 by 3) double array]
  CurrentAxes = []
                . . .
```

La fonction `gcf` est utilisée pour obtenir le numéro de la fenêtre active, et `get` pour obtenir la liste des attributs de la fenêtre et de leur valeur sous la forme : *Nom de l'attribut = Valeur de l'attribut.*

On modifie la valeur des attributs d'une fenêtre avec la fonction `set` :

`set(`*'Nom de l'attribut', 'Valeur de l'attribut', . . . , . . .* `)`

On peut créer directement une fenêtre dont les attributs ont une valeur particulière :

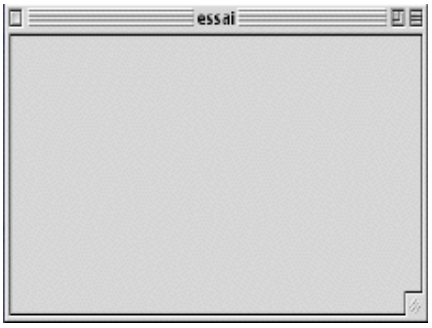
`figure(`*'Nom de l'attribut', 'Valeur de l'attribut', . . . , . . .* `)`

Exemple 5.1.4 :

La séquence :

```
>> figure('Name', 'essai', 'NumberTitle', 'off')
```

crée une fenêtre dont le nom est `essai`.



5.2 Courbes du plan

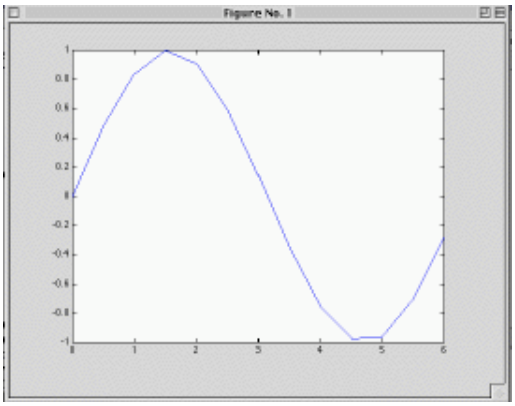
5.2.1 La fonction `plot`

Soient `x` et `y` deux listes (ou deux vecteurs) **de même longueur**. La fonction `plot(x, y)` trace dans la fenêtre active le graphe de `y` en fonction `x`. En fait le graphe est obtenu en joignant par de petits segments de droite les points de coordonnées `(x(k), y(k))` pour ($1 \leq k \leq$ `length(x)`). lorsqu'il n'y a pas de fenêtre active, MATLABcrée automatiquement une nouvelle fenêtre.

Exemple 5.2.1 :

Pour obtenir le graphe de la fonction $\sin(x)$ sur l'intervalle $[0, 2\pi]$:

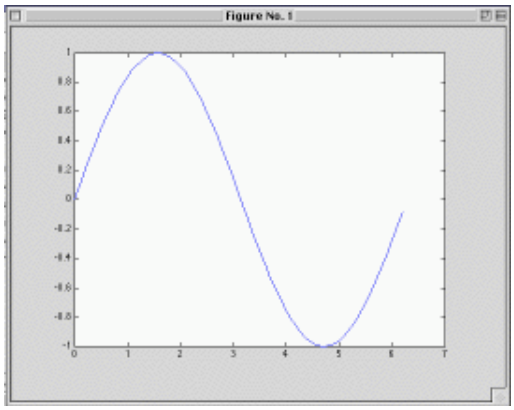
```
>> x=[0:.5:2*pi] ; y=sin(x) ; plot(x,y)
```



MATLAB définit automatiquement un système d'axes. La qualité du tracé dépend du nombre de points construits comme le montre l'exemple suivant dans lequel on a choisi un pas plus petit pour décrire l'intervalle $[0, 2\pi]$:

Exemple 5.2.2 :

```
>> x=[0:.1:2*pi] ; y=sin(x) ; plot(x,y)
```



On remarquera que premier tracé a été supprimé de la fenêtre et qu'il a été remplacé par le second.

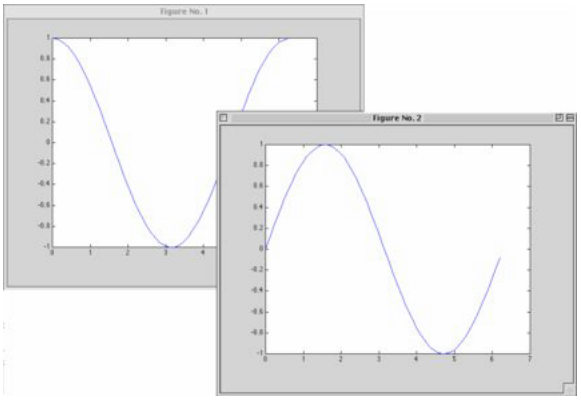
5.2.2 Tracer dans une ou plusieurs fenêtres

Suivant le contexte, on peut souhaiter que :

– **les tracés apparaissent dans des fenêtres différentes :** on crée autant de fenêtres que de tracés en utilisant la fonction `figure` :

Exemple 5.2.3 :

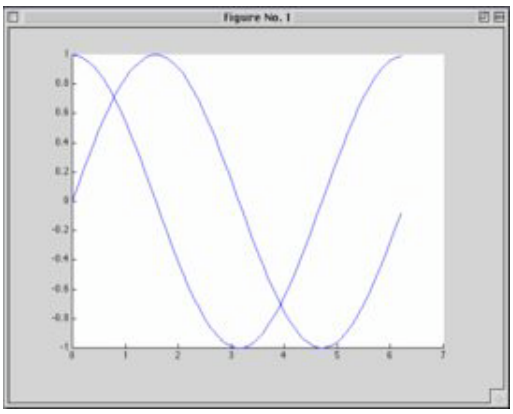
```
>> figure(1) ; x=[0:.1:2*pi] ; c=cos(x) ; plot(x,c)
>> figure(2) ; s=sin(x) ; plot(x,s)
```



Ces deux séquences construisent deux fenêtres, la première contenant le graphe de $\cos(x)$, la seconde le graphe de $\sin(x)$.

- **tous les tracés apparaissent simultanément dans la fenêtre active :** on peut procéder de deux façons :
  - soit en utilisant les **commandes `hold on` et `hold off`** : après `hold on` tous les tracés ont lieu dans la fenêtre active ; `hold off` fait revenir au mode de tracé normal.

    Exemple 5.2.4 :
    ```
    >> x=[0:.1:2*pi] ; c=cos(x) ; s=sin(x) ;
    >> hold on
    >> plot(x,c)
    >> plot(x,s)
    >> hold off
    ```
    Les deux graphes, celui de $\cos(x)$ et celui de $\sin(x)$, apparaissent dans la même fenêtre et dans le même système d'axes.



  - soit en donnant comme argument à `plot` la liste de tous les couples de listes (ou de vecteurs) correspondant aux courbes à tracer : `plot(`$x_1, y_1, x_2, y_2, \cdots$`)` ce qui est exactement équivalent à
    ```
    » hold on
    » plot(x1, y1)
    » plot(x2, y2)
    » . . . .
    » hold off
    ```

Lorsque plusieurs tracés ont lieu dans la même fenêtre, il peut être intéressant d'utiliser un style différent pour distinguer les différents tracés. Pour cela on ajoute un troisième argument à

la définition de chaque tracé : `plot(`$x_1, y_1,' st'_1, x_2, y_2,' st'_2, \cdots$`)` où $'st'_i$ est une chaîne de un à trois caractères pris dans le tableau ci-dessous :

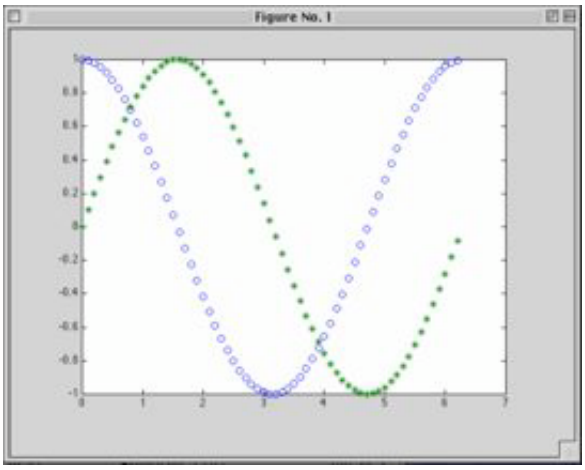
Couleur		**Marqueur**		**Style du tracé**	
y	jaune	.	point	- (∗)	ligne continue
m	magenta	o	cercle	:	ligne pointillée
c	cyan	x	x	- -	tirets
r	red	+	plus	-.	tiret point
g	vert	∗	étoiles		
b	bleu	s	carrés		
w	blanc	d	losanges		
k	noir (∗)	<, >	triangles		

Les valeurs notées (∗) *sont les valeurs par défaut.*

Exemple 5.2.5 :

```
>> x=[0:.1:2*pi] ; c=cos(x) ; s=sin(x) ;
>> plot(x,c,'o',x,s,'*')
```

Les deux graphes, celui de cos(x) et celui de sin(x), se distinguent par leur tracé.



5.2.3 La commande `print`

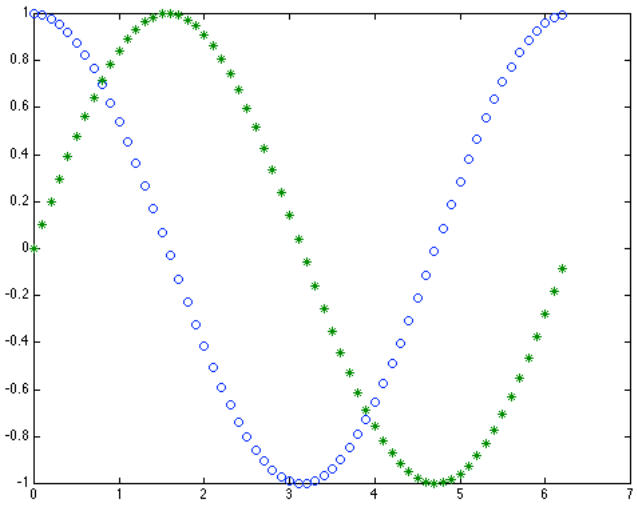
La commande `print` permet d'imprimer le contenu de la fenêtre active, plus précisément de la zone rectangulaire définie par les axes dans laquelle s'inscrit le tracé ou **plot-box**.
Avec pour argument un nom de fichier,

`print` *-option nomDeFichier*

cette commande permet de rediriger l'impression vers le fichier.
Les options dépendent de l'environnement (UNIX, Linux, Windows, MacOs) et de la version de MATLAB utilisée (*c.f* aide en ligne).

Exemple 5.2.6 :

La commande `print -deps fig-gr6.pdf` appliquée à la fenêtre ci-dessus donne la figure suivante :



5.2.4 Courbes paramétriques

La fonction `plot` permet aussi le tracé de courbes planes définies sous la forme :

$$\begin{cases} x = f(t) \\ \\ y = g(t) \end{cases} \quad t \in [t_0, t_1]$$

- on crée un vecteur `t` correspondant à une subdivision de l'intervalle $[t_0, t_1]$
- on crée deux vecteurs `x` et `y` en appliquent respectivement les fonctions f et g à `t`
- on trace le graphe avec `plot(x,y)` (*voir plus loin, exemple 5.2.8*).

5.2.5 Personnalisation des axes et de la *plotting-box*

(Ce paragraphe peut être laissé de côté en première lecture.)
La fonction `plot` ainsi que les autres fonctions de tracé, crée automatiquement deux axes gradués, l'*axe des x* et l'*axe des y* :

- l'*axe des x* est l'axe horizontal ; il est associé au vecteur qui est le premier argument de `plot` et couvre l'intervalle qui s'étend de la plus petite valeur *min* de ce vecteur à sa plus grande valeur *xmax* ;
- l'*axe des y* est l'axe vertical ; il est associé au vecteur qui est le second argument de `plot` et couvre l'intervalle qui s'étend de la plus petite valeur *ymin* de ce vecteur et sa plus grande valeur *ymax*.

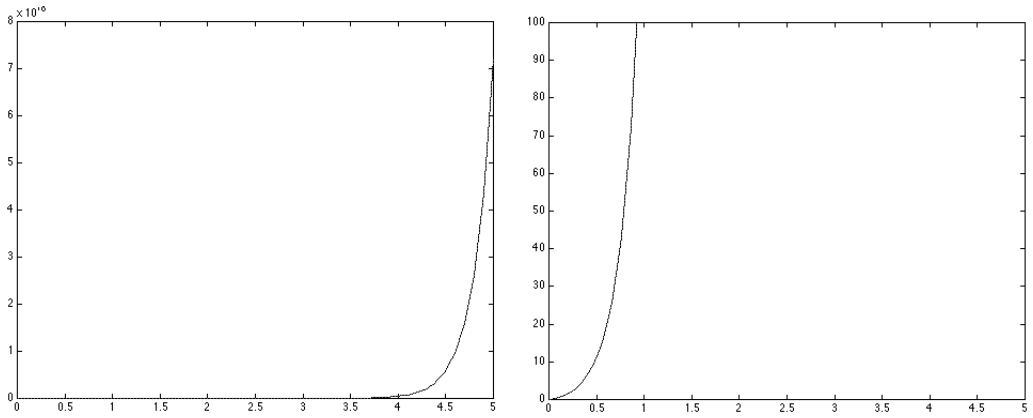
Ces deux axes définissent la zone rectangulaire ou *plotting-box* dans laquelle s'inscrivent les tracés.

la fonction `axis`

- `axis([`x_0, x_1, y_0, y_1`])` permet d'extraire de la *plotting-box* la région rectangulaire définie par les points (x_0, x_1) et (y_0, y_1), et de l'afficher dans la fenêtre active.

Exemple 5.2.7 :

```
>> x = linspace(0,5,100) ; y = exp(5*x)-1 ;
>> figure(1) ; plot(x,y)
>> figure(2) ; plot(x,y) ; axis([0,5,0,100])
```



L'exemple précédent montre que la modification de la zone affichée a une incidence non négligeable sur le **facteur d'échelle** utilisé par MATLAB. Le graphe de la fonction $\exp(5x)-1$ qui paraît très "plat" dans la figure de gauche ne l'est pas autant dans la figure de droite.

- `axis` *option* modifie l'apparence des axes et de la *plotting-box* :
  - `manual` fixe les bornes (et le facteur d'échelle) de chaque axe à leur valeur actuelle, de telle sorte que si `hold` a la valeur `on`, les tracés suivants ne pourront les modifier ;
  - `equal` fixe une échelle commune aux deux axes ;
  - `square` donne à la *plotting-box* une forme carrée ;
  - `normal` rend à la *plotting-box* sa forme rectangulaire usuelle et restaure les valeurs des bornes de chaque axe à leur valeur par défaut ;
  - `auto` retour au mode automatique de définition des axes et de la *plotting-box*.

  Consultez l'aide en ligne pour d'autres usages de `axis`.

Exemple 5.2.8 :

Dans cet exemple, on trace la courbe (un cercle) définie par l'équation paramétrique :

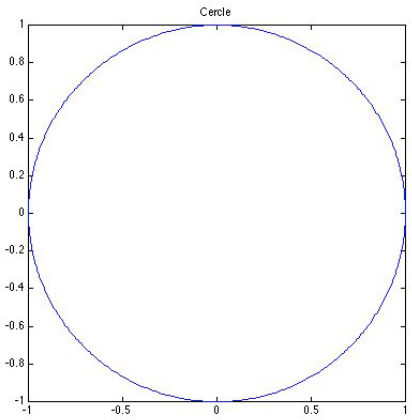
$$\begin{cases} x = \cos(t) \\ y = \sin(t) \end{cases} \quad \text{pour } t \in [0, 2\pi]$$

```
>> t = linspace(0,2*pi,500 ); x=cos(t) ; y=sin(t) ;
>> plot(x,y) ; axis equal square ; title('Cercle');
```

`axis equal` est ici nécessaire pour que le cercle n'ait pas la forme d'une ellipse :

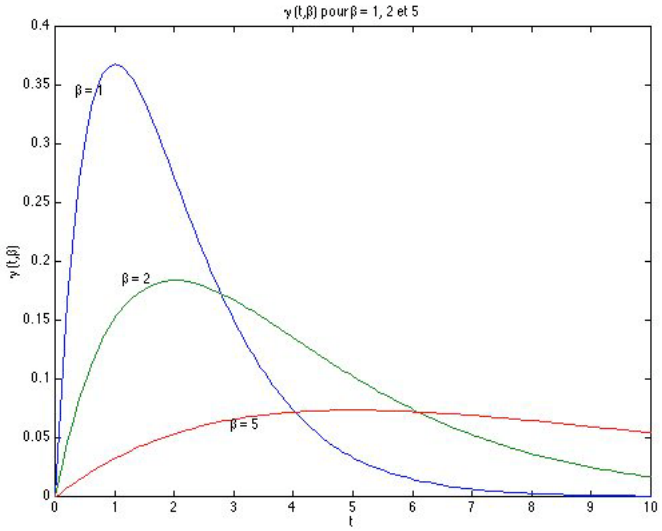
les fonctions `xlabel`, `ylabel`, `title` et `gtext`

Les fonctions `xlabel('x-legend')` et `ylabel('y-legend')` permettent d'associer une légende à l'axe de coordonnée correspondant. La fonction `title('titre')` ajoute un titre au tracé (pas à la fenêtre) et `gtext('texte')` permet de placer avec la souris le texte passé comme argument. Toutes ces chaînes de caractères peuvent contenir des commandes LaTeX.



Exemple 5.2.9 :

```
>> t = linspace(0,5,100) ;
>> x1 = t.*exp(t) ; x2 = t.*exp(t/2)/4 ; x3 = t.*exp(t/5)/25
>> plot(t,x1,t,x2,t,x3,) ; title('\beta = 1, 2 et 5')
>> xlabel('t') , ylabel('\gamma (t,\beta)')
>> gtext('\beta = 1')
>> gtext('\beta = 2')
>> gtext('\beta = 5')
```



les fonctions `box` et `grid`

La commande `box on` affiche un cadre qui délimite la *plotting-box* ; `box off` supprime ce cadre ; `grid on` superpose une grille au tracé ; `grid off` supprime cette grille.

5.2.6 Autres fonctions de tracé de courbes planes

Les fonctions dérivées de `plot` sont nombreuses. On peut se référer à l'aide pour en avoir une liste exhaustive.

fonctions `loglog`, `semilogx` et `semilogy`

Ces fonctions sont semblables à `plot` excepté le fait qu'une échelle logarithmique est utilisée respectivement pour les deux axes, l'*axe des x* et l'*axe des y*.

fonction `plotyy`

Avec une syntaxe proche de celle de `plot`, `plotyy`$(x_1, y_1, 'st'_1, x_2, y_2, 'st'_2)$ trace y_1 en fonction de x_1 et y_2 en fonction de x_2 avec deux *axes des y* l'un à gauche de la *plotting-box* adapté à y_1, l'autre à droite adapté à y_2.

fonction `fplot`

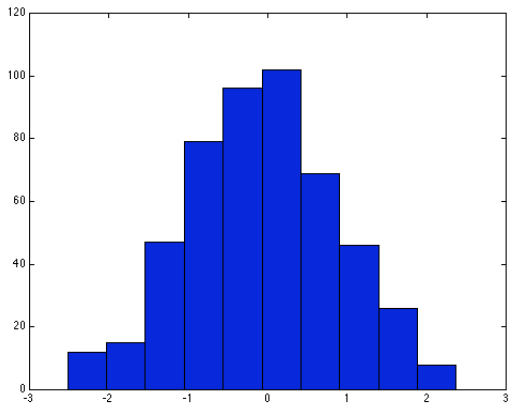
La syntaxe de la fonction `fplot` est voisine de celle de la fonction `fzero` *(c.f. 4.4.2)*. La fonction `fplot` prend pour arguments le nom (sous forme d'une chaîne de caractères) de la fonction dont on souhaite tracer le graphe et les valeurs des bornes de l'intervalle d'étude. L'intervalle d'étude est subdivisé par MATLAB de façon à donner le tracé le plus précis possible.

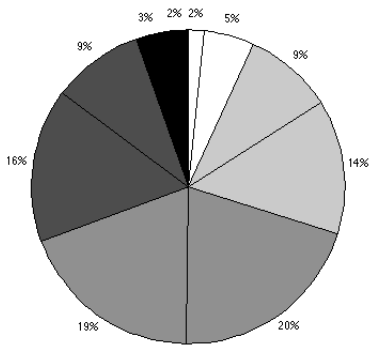
les fonctions `hist` et `pie`

la fonction `hist(x,n)` répartit les valeurs de la liste (ou du vecteur) `x` en `n` classes et trace l'histogramme correspondant (par défaut $n = 10$). `[N,X] = hist(x,n)` retourne dans `N` l'effectif de chacune des classes et dans `X` l'abscisse du centre de chaque classe.
La fonction `pie(x)` dessine un diagramme des valeurs de `x` normalisées par $s = \sum_{i=1}^{n} x_i$.

Exemple 5.2.10 :

```
>> a = randn(1,500) ; hist(a)
>> [N,X] = hist(a)
  N =
     12    15    47    79    96   102    69    46    26     8
  X =
    -2.275  -1.786  -1.297  -0.808  -0.318   0.171   0.660  1.150    1.639    2.128
>> pie(N)
```



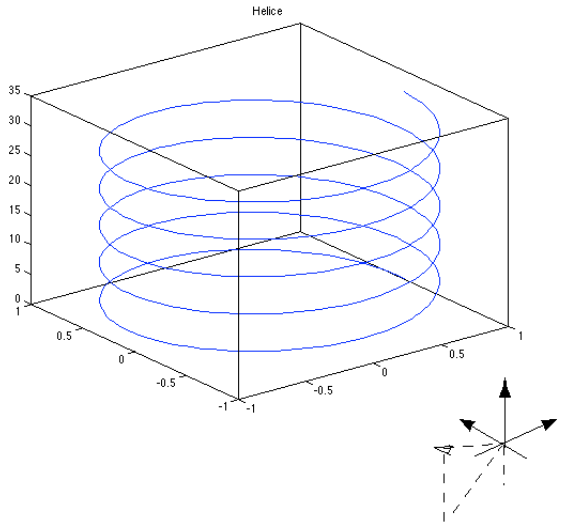


5.3 Courbes de l'espace - Fonction plot3

La fonction `plot3` étends les fonctionnalités de `plot` aux courbes de l'espace. Les possibilités de personnalisation des axes sont les mêmes :

Exemple 5.3.1 :

```
>> t = linspace(0,10*pi,500) ;
>> x = cos(t) ; y = sin(t) ;
>> plot3(x,y,t) ; title('Helice') ; box on ; rotate3d on
```



MATLAB donne une vue perspective du graphe de la fonction, inclus dans une *plotting-box* parallélépipédique. Dans les versions récentes de MATLAB, `rotate3d on` permet de déplacer la *plotting-box* avec la souris.

5.4 Surfaces de l'espace

Dans cette section on va voir comment MATLAB permet de représenter des surfaces définies par une relation $z = f(x, y)$ où f est une fonction continue, définie sur un domaine $[x_0, x_1] \times [y_0, y_1]$.

5.4.1 Modélisation du domaine $[x_0, x_1] \times [y_0, y_1]$ - fonction `meshgrid`

Cette modélisation se fait en deux étapes :

- définition de deux **subdivisions régulières** : `x` pour $[x_0, x_1]$ et `y` pour $[y_0, y_1]$;
- **construction d'une grille** modélisant le domaine $[x_0, x_1] \times [y_0, y_1]$: La grille est définie par deux tableaux `xx` et `yy` résultant de `[xx,yy] = meshgrid(x,y)`.
  Précisément, $xx(l, k) = x(k)$ et $yy(l, k) = y(l)$ pour tout k, $(1 \leq k \leq$ `length(x)`$)$ et pour tout l, $(1 \leq l \leq$ `length(y)`$)$ de telle sorte que $(xx(l, k), yy(l, k)) = (x(k), y(l))$.

Il est alors possible d'évaluer les valeurs de f suivant cette grille en appliquant f au couple de tableaux `xx` et `yy`.

5.4.2 Tracé de la surface - fonctions `mesh` et `surf`

Une fois le domaine d'étude modélisé par deux tableaux `xx` et `yy`, on évalue les valeurs de la fonction pour obtenir un tableau `z = f(xx,yy)`. On dessine la surface $z = f(x, y)$ (tracée en perspective dans une *plotting-box* comme pour `plot3`) avec l'une des fonctions suivantes :

fonction `mesh`

`mesh(xx,yy,z)` donne une représentation de la surface par un maillage “fil de fer".

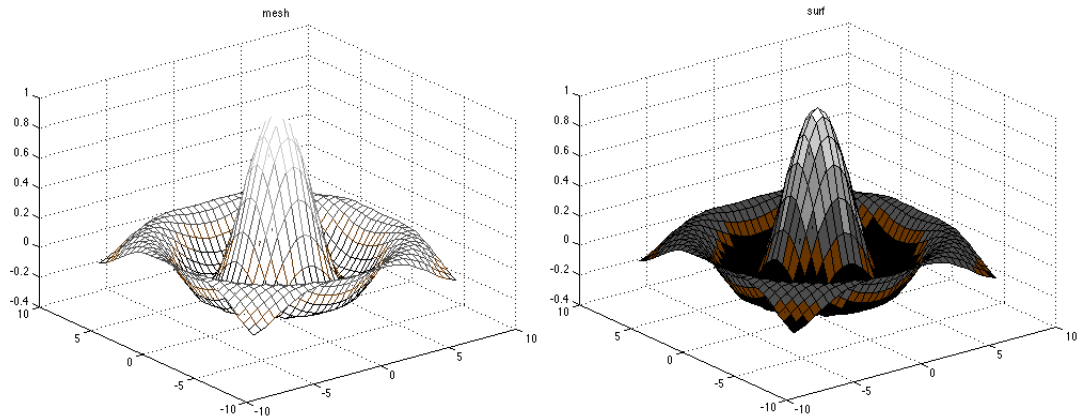
fonction `surf`

`surf(xx,yy,z)` donne une représentation où les mailles sont colorées.

Comme pour les courbes de l'espace, la commande `rotate3d on` permet de déplacer la *plotting-box* à l'aide de la souris.

Exemple 5.4.1 :

```
>> x = -7.5 : .5 : 7.5 ; y = x ; [xx,yy] = meshgrid(x,y) ;
>> r = sqrt(xx.^2+yy.^2)+eps ; z = sin(r)./r ;
>> figure(1) ; mesh(xx,yy,z) ; title('mesh')
>> figure(2) ; surf(xx,yy,z) ; title('surf')
```



5.4.3 Surfaces et courbes de niveau

Comme pour le tracé d'une surface, on commence par modéliser le domaine d'étude par deux tableaux `xx` et `yy`, puis on évalue les valeurs de la fonction et on obtient un tableau `z = f(xx,yy)`. Plusieurs fonctions permettent alors de dessiner les surfaces de niveau de f : `contour`, `contour3` et `pcolor`.

fonction `contour`

`contour(xx,yy,z,n)` détermine n surfaces de niveau(10 par défaut) et les projette sur le plan xoy. Au lieu de spécifier le nombre de niveaux, il est possible d'indiquer leur valeurs sous forme d'une liste $[z_0 : pas : z_1]$, en particulier pour obtenir la surface correspondant à un niveau donné z_0, on utilisera `contour(xx,yy,z,[z0])`. En niveaux de gris (*c.f.* `colormap`), la couleur des courbes de niveau est d'autant plus claire que la valeur du niveau l'est. Il est possible de fixer la couleur des courbes de niveau en utilisant un un caractère (*c.f. code des couleurs 5.2.2* comme dernier argument.

fonction `contour3`

Fonction semblable à `contours`, `contour3` détermine n surfaces de niveau et en donne une représentation en trois dimensions. Comme pour `contour`, la couleur des courbes de niveau est d'autant plus claire que la valeur du niveau l'est.

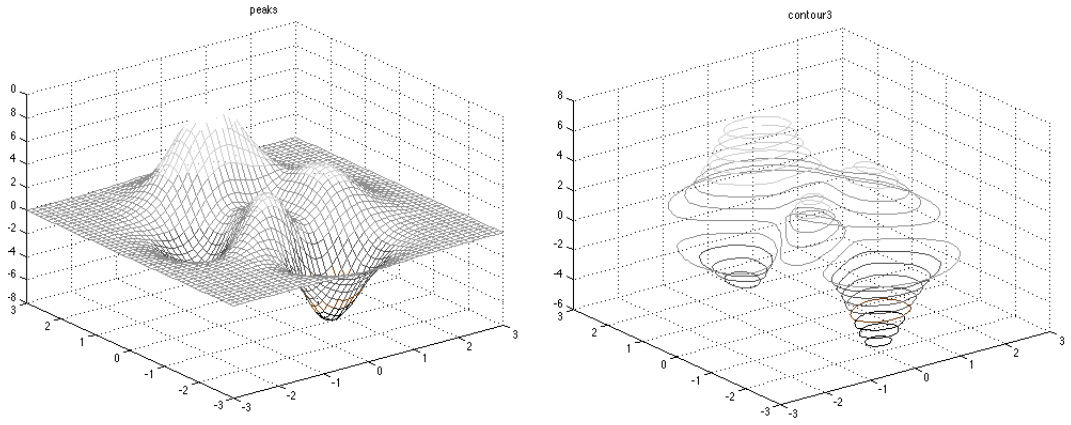
fonction `pcolor`

`pcolor(xx,yy,z)` génère une image plane à la même échelle que `contour` et dont les pixels ont une couleur qui si on utilise une échelle de gris (*c.f.* `colormap`), est d'autant plus claire que la valeur de $f(x, y)$ est grande. Cette fonction est utilisée en conjonction avec `contour`.

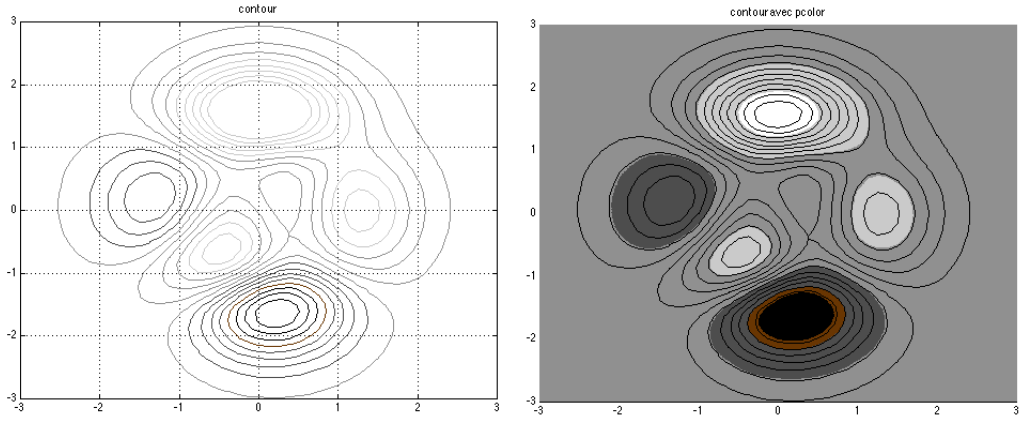
Exemple 5.4.2 :

Dans les exemples ci-dessous, on utilisera une fonction-test prédéfinie `peaks`.

```
>> [xx,yy,z] = peaks ;
>> figure(1) ; mesh(xx,yy,z) ; title('peaks')
>> figure(2) ; contour3(xx,yy,z) ; title('contour3')
```




```
>> figure(3) ; contour(xx,yy,z) ; title('contour')
>> figure(4) ; pcolor(xx,yy,z)
>> shading interp                    % supprime la grille
>> hold on
>> contour(xx,yy,z,'k')             % superpose les courbes de niveau en noir
>> title('contour avec pcolor')
>> hold off
```



6

Importation et exportation de données

Les échanges de données entre applications utilisent généralement des fichiers. MATLAB possède un grand nombre de fonctions pour les gérer *c.f.* `help iofun`. En particulier, les versions les plus récentes de MATLAB possèdent un assistant d'import de fichier qui permet d'accéder à de nombreux types de fichiers (consultez l'aide).
Cependant une méthode très simple consiste en l'utilisation de *fichiers-text*, l'utilisation de fichiers binaires se révélant parfois plus délicate (on peut en effet contrôler et modifier le contenu d'un fichier-text à l'aide d'un simple éditeur). C'est donc cette méthode que nous allons développer dans cette premiere section.

6.1 Retour sur les commandes `save` et `load`

La commande `save`, on l'a vu *c.f. 2.3.3*, permet d'enregistrer la valeur d'une ou plusieurs variables sous forme d'un fichier binaire appelé *fichier.mat*. Réciproquement, la commande `load` ajoute le contenu d'un tel fichier à l'environnement de travail de la session en cours.
Les commandes `save` et `load` peuvent aussi malgré quelques restrictions, être utilisées avec des fichiers-text.

6.1.1 Enregistrement de la valeur de tableaux dans un fichier-text - `save`

La commande

`save` *<nom de fichier>* *<liste de variables>* `-ascii`

enregistre les tableaux associés aux variables de *<liste de variables>* dans le fichier-text désigné par *<nom de fichier>*, en suivant les conventions suivantes :

- les valeurs des éléments du ou des tableaux sont enregistrées au format scientifique avec huit chiffres significatifs, **le séparateur décimal étant un point** ;

– les éléments d'une même ligne sont séparés par des espaces ;
– les lignes successives sont séparés par des sauts de ligne.

Exemple 6.1.1 :

```
>> a = 1 : 5 , h = hilb(3)
a =
    1 2 3 4 5
b =
   1.0000    0.5000    0.3333
   0.5000    0.3333    0.2500
   0.3333    0.2500    0.2000
>> save ab.txt a b -ascii
```

On peut éditer les fichier ainsi créés avec n'importe quel éditeur ou directement dans MATLAB en utilisant la commande `type` :

Exemple 6.1.2 :

```
>> type ab.txt
 1.0000000e+00   2.0000000e+00   3.0000000e+00   4.0000000e+00  5.0000000e+00
 1.0000000e+00   5.0000000e-01   3.3333333e-01
 5.0000000e-01   3.3333333e-01   2.5000000e-01
 3.3333333e-01   2.5000000e-01   2.0000000e-01
```

Les valeurs des éléments des deux tableaux ont été enregistrées en commençant par les valeurs de `a` suivies de celles de `b`. Cependant, on peut remarquer que le nombre d'éléments de chacune des lignes du fichier-text n'est pas constant : cinq pour la première ligne, trois ensuite.

Il est possible d'enregistrer les valeurs des éléments avec seize chiffres significatifs au lieu de huit, en faisant suivre `-ascii` de l'option `-double` :

Exemple 6.1.3 :

```
>> c = pi/6 : pi/6: pi/2
c =
    0.5236    1.0472    1.5708
>> save c.txt c -ascii -double
>> type c.txt
 5.2359877559829882e-01   1.0471975511965976e+00   1.5707963267948966e+00
```

6.1.2 Retrouver la valeur d'un tableau - `load`

On dira qu'un fichier-text a une **structure valide pour la commande `load`** lorsque qu'il a la forme d'un tableau de nombres :

– le fichier ne contient que des chaînes de caractères pouvant représenter des valeurs numériques (le séparateur décimal étant nécessairement un point), séparées par des espaces ;
– toutes les lignes du fichier comportent exactement le même nombre de telles chaînes de caractères.

Il n'est pas cependant nécessaire que le nombre d'espaces séparant les différentes chaîne numériques soit constant dans tout le fichier, ni que le format des chaînes numériques soit le même.

Exemple 6.1.4 :

Le fichier `d.txt` suivant

```
  1.00 2.0   3  4.0  5.0000000e+000  6.0000000e+000
   6.0e+000   5.00e+00  1.00 2.0  3 4.0
```

est valide. Il comporte six chaînes numériques par lignes. Par contre le fichier défini dans l'exemple *6.1.1* ci-dessus n'est pas valide puisque ses lignes ne comportent pas le même nombre de valeurs numériques.

Si le fichier <*fich*> est valide,

`load <`*fich*`> -ascii`

ajoute à l'environnement de travail une nouvelle variable dont le nom est celui du fichier privé de son extension et dont la valeur est le tableau défini par le fichier.

Exemple 6.1.5 :

```
>> load d.txt -ascii
d =
     1     2     3     4     5     6
     6     5     1     2     3     4
```

On notera que le comportement de `load` dans le cas où son argument est un fichier-text est très différent du cas où l'argument est un fichier *.mat* c.f. 2.3.3.

6.2 Lire et écrire dans un fichier Excel

On utilise les fonctions `xlsread` pour la lecture et `xlswrite` pour l'écriture.
La fonction `xlsfinfo` permet de savoir si le fichier dont le nom est passé en argument est un fichier excel qui peut être utilisé par les fonctions `xlsread` ou `xlswrite`.
La séquence

```
filename = 'exemple.xlsx';
statut = xlsfinf(filename)
```

retournera `Microsoft Excel Spreadsheet` si le fichier `exemple.xlsx` est bien un fichier Excel valide, la chaîne vide sinon.

6.2.1 Importer des valeurs d'un fichier Excel - `xlsread`

La séquence :

```
filename = 'exemple.xlsx';
sheet = 1;
xlRange = 'B2:C3';

A = xlsread(filename, sheet, xlRange)
```

affecte à la variable `A` le contenu numérique de la plage de cellules définies par la chaîne de caractères `xlRange` :

```
A =
     1     2
     3     4
```

Dans MATLAB et Excel, les dates peuvent être représentées par des entiers. Les fonctions `datestr` et `datenum` permettent facilement la conversion du format "string" au format numérique (attention la date origine de l'échelle des temps n'est pas la même dans MATLAB , dans Excel for Windows et dans Excel for Mac).
Supposons que le fichier Excel `exemple2.xlsx` comporte dans sa première feuille à partir de la cellule $A1$ les données suivantes :

cc	exam
1	2
4	5

alors, la séquence :

```
filename = 'exemple2.xlsx';
sheet = 1;
xlRange = 'A1:B3';

[A, texte, ' ']  = xlsread(filename, sheet, xlRange)
```

affecte à la variable `A` le contenu numérique de la plage de cellules définies par la chaîne de caractères `xlRange` et au tableau de chaînes de caractères `texte` le contenu non-numérique de la plage de cellules.

```
A =
     1     2
     3     4

texte =
     'cc'    'exam'
       ' '         ' '
       ' '         ' '
```

6.2.2 Exporter des valeurs vers une feuille Excel - `xlswrite`

`xlswrite` permet l'écriture de valeurs numériques ou non dans un fichier Excel.

Exemple 6.2.1 :

La séquence :

```
filename = 'exemple3.xls';
A = {'Heure','Temperature'; 10,98; 11,99; 12,97};
sheet = 1;
xlRange = 'E1';
status = xlswrite(filename,A,sheet,xlRange)
```

écrit dans la première feuille du fichier Excel `exemple3.xls`, à partir de la position $E1$, les valeurs données et affecte à la variable`status` la valeur 1 si l'opération s'est complètement effectuée, 0 sinon.

7

Matrices-test

On trouve un très grand nombre de matrices test aux adresses :
- *http ://www.netlib.org/toms/694*
- *http ://math.nist.gov/MatrixMarket/*

En voici quelques unes :
- **Matrices bordées** La matrice bordée d'ordre n est une matrice symétrique de terme général

$$\begin{cases} m_{ii} = 1 \\ m_{in} = m_{ni} = 2^{1-i} \\ 0 \quad \text{sinon} \end{cases}$$

  Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & 1/2 & 1/4 \\ 1/2 & 1 & 0 \\ 1/4 & 0 & 1 \end{bmatrix}$$

- **Matrices Ding-dong** La matrice ding-dong d'ordre n est une matrice symétrique de terme général

$$m_{ij} = \frac{1}{2(n-i-j+1.5}$$

  Exemple pour $n = 3$:

$$\begin{bmatrix} 1/5 & 1/3 & 1 \\ 1/3 & 1 & -1 \\ 1 & -1 & -1/3 \end{bmatrix}$$

- **Matrices de Franc** La matrice de Franc d'ordre n est la matrice de terme général

$$\begin{cases} m_{ij} = 0 \quad \text{si } i \geq j+2 \\ m_{ij} = \min i, j \quad \text{sinon} \end{cases}$$

  Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 0 & 2 & 3 \end{bmatrix}$$

- **Matrices de Hilbert** La matrice de Hilbert d'ordre n (générée par la fonction **matlab** hilb) est une matrice de terme général

$$m_{ij} = \frac{1}{i+j-1}$$

. Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}$$

– **Matrices kms** Les matrices kms d'ordre n dépendent d'un paramètre p compris strictement entre 0 et 1. Leur terme général est de la forme $m_{ij} = p^{|i-j|}$.
Exemple pour $n = 3$ et $p = 1/2$:

$$\begin{bmatrix} 1 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1 \end{bmatrix}$$

– **Matrices de Lehmer** La matrice de Lehmer d'ordre n est une matrice symétrique de terme général :

$$\begin{cases} m_{ij} = \frac{i}{j} & \text{pour } i \leq j \\ m_{ij} = \frac{j}{i} & \text{pour } j \leq i \end{cases}$$

Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1 & 2/3 \\ 1/3 & 2/3 & 1 \end{bmatrix}$$

– **Matrices de Lotkin** La matrice de Lotkin d'ordre n est obtenue à partir de la matrice de Hilbert d'ordre n en remplaçant les termes de la première ligne par des 1.

$$\begin{cases} m_{1j} = 1 & \text{pour } i = 1 \\ m_{ij} = \frac{1}{i+j-1} & \text{pour } i \neq 1 \end{cases}$$

Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}$$

– **Matrices de Moler** La matrices de Moler d'ordre n est une matrice symétrique définie positive de terme général :

$$\begin{cases} m_{ii} = i \\ m_{ij} = \min(i, j) - 2 & \text{pour } j \neq i \end{cases}$$

Exemple pour $n = 3$:

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 2 & 0 \\ -1 & 0 & 3 \end{bmatrix}$$

– **Matrices de Pascal** La matrices de Pascal d'ordre n est une matrice symétrique définie positive de terme général :

$$\begin{cases} m_{ij} = C_{i-1}^{j-1} & \text{pour } j \leq i \\ m_{ij} = C_{j-1}^{-1i} & \text{pour } i \leq j \end{cases}$$

Exemple pour $n = 4$:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 3 \\ 1 & 2 & 1 & 3 \\ 1 & 3 & 3 & 1 \end{bmatrix}$$

– **Matrices de Pei** Les matrices de Pei d'ordre n dépendent d'un paramètre a. Leur terme général est de la forme :

$$\begin{cases} m_{ii} = 1 + a \\ m_{ij} = 1 \quad \text{pour } i \neq j \end{cases}$$

Pour $n = 3$ et $a = 2$ on obtient :

$$\begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}$$

– **Matrice de Rosser** La matrice de Rosser est une matrice symétrique d'ordre 8 générée directement dans **matlab** par la fonction rosser.

– **Matrices de Vandermonde** Les matrices de Vandermonde sont des matrices $m \times n$ dont le terme général est de la forme $m_{ij} = p_j^{i-1}$ où les p_j sont des nombres réels donnés. Elles sont générées dans **matlab** par la fonction vander.
Pour m = 4 et n = 3, la matrice de Vandermonde générée par les nombres (1, 2, 5) est :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 5 \\ 1 & 4 & 25 \\ 1 & 8 & 125 \end{bmatrix}$$

– **Matrices de Wilkinson** Les matrices de Wilkinson sont des matrices tridiagonales générées dans MATLAB par la fonction `wilkinson`.

8

Exemples

Mise en œuvre

- Création d'un répertoire de travail
  - Créez dans votre *home directory* un répertoire *tpMatlab*.
- Une session MATLAB
  - Ouvrez une session sur votre poste de travail.
  - Lancer MATLAB en cliquant sur l'icône de MATLAB ou en tapant `matlab` depuis un terminal. MATLAB ouvre alors une fenêtre (prompt >>) appelée fenêtre de commande.
  - Choisir le repertoire *tpMatlab* comme répertoire de travail en tapant dans la fenêtre de commande :
    ```
    cd P:
    cd tpMatlab
    ```
  - En fin de session, vous quitterez MATLAB en tapant `quit` dans la fenêtre de commande.

Aide en ligne

MATLAB comporte un très grand nombre de fonctions. Heureusement, une aide en ligne efficace peut être utilisée :

- `help` : aide de l'aide et liste thématique ;
- `help` *`item`* : définition de *item* et exemples de mise en oeuvre ;
- `lookfor` *`sujet`* : liste des rubriques de l'aide en ligne en relation avec le sujet indiqué ;
- `helpdesk` : aide sous forme de fichiers hypertexte.

Exercice 1 - Opérations

1. Créez les tableaux suivants

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 3 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

2. Extraire la première ligne, la deuxième colonne de la matrice A, les termes diagonaux de B, la sous-matrice $A(1..2, 2..5)$ de A.
3. Testez les différentes opérations sur ces tableaux : $+, -, *, /, \backslash$. Que se passe-t-il lorsqu'on fait précéder ces opérateurs d'un point ?

Exercice 2

Avec une matrice A d'ordre élevé, la matrice de Hilbert par exemple (fonction *hilb*) on veut mettre en évidence les problèmes d'erreur de calcul :

1. Fixez un vecteur $x0$ et évaluez `b0 = A*x0`.
2. Résolvez à l'aide de l'opérateur \ le système $Ax = b0$. Si A, B, X sont des matrices :
   - $X = B/A$ est la solution de $XA = B$;
   - $X = A\backslash B$ est la solution de $AX = B$
3. Comparez x et $x0$ en évaluant l'erreur sur x, c'est à dire la norme $\|x - x0\|$ (`norm(x - x0)`) ou mieux la quantité $\epsilon = \frac{\|x-x0\|}{\|x0\|}$ qui est appelée erreur relative sur x.

Exercice 3

1. Créez à l'aide d'un éditeur un fichier que vous appellerez `sigma2a.m` qui contiendra le texte suivant :

```
function s=sigma2a(n)
% calcule la somme des carrés des n premiers entiers
  s=0; k=1;
  for k = 1:n
    s=s+(k*k);
  end % termine le for
```

2. Dans la fenêtre `matlab` taper `s=0 ; s = sigma2a(100)` puis `s=1000 ; s = sigma2a(100)`. Conclusion ?
3. Créez de même le fichier `sigma2b.m` contenant le texte :

```
function s=sigma2b(n)
% somme des carres a partir de la liste des n premiers entiers
  s=0; k=1;
  while (k<=n)
    s=s+(k*k);
    k=k+1;
  end
```

4. Enfin créez le fichier `sigma2c.m` contenant le texte :

```
function s=sigma2c(n)
% somme des carres a partir de la liste des n premiers entiers
  l=(1:n).^2;
  s=sum(l);
```

5. Testez les fonctions `sigma2a` et `sigma2b`. Pour comparer leurs performances utilisez la commande `profile` que vous chercherez dans l'aide.
6. Concluez ?

Exercice 4

1. Écrivez une fonction `normsup` qui détermine le plus grand élément en valeur absolue d'un vecteur ou d'une matrice. Pour cela vous pourrez utiliser la fonction `size` et la structure de sélection :

```
if (prédicat)
  instruction 1
else
  instruction 2
end
```

2. Complétez cette fonction pour qu'elle détermine également la position de cet élément dans le vecteur ou la matrice.
3. En vous aidant de l'aide comparez cette fonction à celle calculant la norme-sup d'un vecteur ou d'une matrice.

Exercice 5

1. Créez la fonction `puiss` suivante :

```
function [v,p]=puiss(Mat, u, n)
% calcule le n ième itéré du vecteur u
  v=u;
  for k=1:n
    z=Mat*v;
    p=v'*z;
    v=sign(z'*u)*z/normsup(z);
  end
```

2. Testez la fonction `puiss`. Elle sera appelée sous la forme

```
x,p =puiss(A,x,n)
```

où A, x et n ont des valeurs pertinentes.

3. Remplacez la boucle `for` par une boucle `while` pour que le calcul s'arrête lorsque

$$\|Ax - lx\| < \epsilon$$

.

Ex 6 - Algorithme de Gauss

1. Implantez l'algorithme de réduction d'un système linéaire $Ax = b$ à la forme triangulaire sous forme d'une fonction `Sg`. Testez-le avec des matrices test prédéfinies (utilisez l'aide en ligne).
2. Comparez le résultat avec celui obtenu par `b\A`.
3. Modifiez la fonction précédente pour qu'elle calcule l'inverse d'une matrice carrée d'ordre n (vous chercherez à résoudre le système matriciel $AB = I_n$).
4. Comparez le résultat avec celui obtenu par la fonction prédéfinie `inv`.

Ex 7 - Tracé de courbe On complète l'exercice 2 en faisant tracer la courbe de l'erreur relative commise sur le calcul de la solution de $Hx = b_0$ où $b_0 = Ax_0$, en fonction de l'ordre n de la matrice de Hilbert H_n. Vous prendrez pour x_0 le vecteur dont toutes les composantes sont égales à 1.

1. Écrivez une fonction `Erreur(n)` qui retourne en fonction de n, la valeur de

$$\frac{\|x - x_0\|}{\|x_0\|} \text{ où } x \text{ est la solution de } H_n x = b_0$$

   Enregistrez la fonction dans le fichier `Erreur.m`.

2. Appelez la fonction `Erreur(n)` depuis le script `Graph` suivant :

```
for n=1:20
  f(n) = Erreur(n) ;
end ;
figure(1)
plot(f)
```

Index