www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Introduction à Matlab

Ours Blanc des Carpathes ™

1. Bienvenue dans Matlab *2*
1.1 La fenêtre de commande Matlab 2
1.2 La gestion des commandes Matlab 3
1.3 La ligne de commande Matlab 5
1.4 Outils à usage général 6
2. Les opérations élémentaires dans Matlab *11*
2.1 Types de données 11
2.2 Opérations arithmétiques élémentaires 11
2.3 Opérations sur les matrices 11
3. Opérateurs généralistes de Matlab *18*
3.1 Opérateurs de comparaison 18
3.2 Opérateurs logiques 19
3.3 Autres fonctions de test 19
4. Quelques fonctions régulièrement utilisées *21*
4.1 Décompositions de matrices 21
4.2 Valeurs et vecteurs propres 22
4.3 Gestion des polynômes avec matlab 22
5. La programmation avec Matlab *27*
5.1 Ecriture de sous programmes 27
5.2 Les structures de contrôle 28
6. Manipulation des chaînes de caractères *29*
6.1 Représentation et fonctions élémentaires sur les chaînes 29
6.2 Utilisation « avancée » 30
7. Graphisme avec Matlab *32*
7.1 Généralités 32
7.2 Les tracés en 2 dimensions 35
7.3 Les tracés en 3 dimensions 40
8. Calcul de performance *43*
8.1 Syntaxe 43
8.2 Un exemple d'utilisation 43
9. La toolbox optimisation *44*
9.1 La programmation linéaire 44
9.2 La programmation quadratique 45
9.3 Le vecteur des options 45
9.4 L'optimisation sans contrainte 46
9.5 Optimisation avec contraintes 49
10. Conclusion *50*

1. Bienvenue dans Matlab

1.1 La fenêtre de commande Matlab

Lorsque vous ouvrez Matlab vous vous retrouvez face à un prompt (le plus souvent 2 guillemets) qui vous invite à entrer des commandes.

L'une des plus utiles est assurément **help** qui appelle pour vous la commande d'aide.

Utilisée seule et sans argument, la commande **help** affiche la liste des sujets pour lesquels l'aide est disponible, chaque sujet étant accompagné d'une brève description des fonctionnalités qu'il contient. Par exemple :

```
» help

HELP topics:

toolbox\local          -  Local function library.
toolbox\optim          -  Optimization Toolbox.
matlab\datafun         -  Data analysis and Fourier transform
functions.
matlab\elfun           -  Elementary math functions.
matlab\elmat           -  Elementary matrices and matrix manipulation.
matlab\funfun          -  Function functions - nonlinear numerical
methods.
matlab\general         -  General purpose commands.
matlab\color           -  Color control and lighting model functions.
matlab\graphics        -  General purpose graphics functions.
matlab\iofun           -  Low-level file I/O functions.
matlab\lang            -  Language constructs and debugging.
matlab\matfun          -  Matrix functions - numerical linear algebra.
matlab\ops             -  Operators and special characters.
matlab\plotxy          -  Two dimensional graphics.
matlab\plotxyz         -  Three dimensional graphics.
matlab\polyfun         -  Polynomial and interpolation functions.
matlab\sounds          -  Sound processing functions.
matlab\sparfun         -  Sparse matrix functions.
matlab\specfun         -  Specialized math functions.
matlab\specmat         -  Specialized matrices.
matlab\strfun          -  Character string functions.
matlab\dde             -  DDE Toolbox.
matlab\demos           -  The MATLAB Expo and other demonstrations.

For more help on directory/topic, type "help topic".
```

Comme vous l'apprendrait la commande `help help`, cette dernière est utilisable avec deux types de paramètres :

- **help** `sujet` renvoie l'aide sur un sujet complet. Par exemple, **help** `matfun` affiche l'aide relative aux fonctions agissant sur des matrices.
- **help** `commande` décrit une commande unique. Par exemple, **help** `schur` affiche l'aide relative à la commande `schur` qui permet de décomposer une matrice selon la forme de Schür.

Que ce soit relativement à un domaine ou a une commande unique, l'aide se décompose toujours en deux parties :

1) une courte description de quelques mots : **l'entête de l'aide**

2) quelques lignes décrivant plus en détails la méthode et que nous appelerons : **le corps de l'aide**

La commande **`lookfor`** `mot` permet de retrouver la liste de toutes les commandes dont l'entête d'aide contient `"mot"`. Ces dernières sont affichées accompagnées de leur entête d'aide. Par exemple :

```
» lookfor decomposition
QR       Orthogonal-triangular decomposition.
SCHUR    Schur decomposition.
SVD      Singular value decomposition.
DMPERM   Dulmage-Mendelsohn decomposition of matrix A.
```

L'exécution de cette commande peut prendre un certain temps, car Matlab est obligé de rechercher dans tous les fichiers d'aide à sa disposition.

Précision de la plus haute importance : Matlab est sensible à la casse des commandes. Et, quand bien même l'aide de Matlab indique le nom des commandes en majuscule, il est impératif de taper en *minuscules* le nom de toutes les commandes prédéfinies de Matlab, qu'elles soient internes ou externes.

1.2 La gestion des commandes Matlab

Matlab reconnaît deux types de commandes :

- **Les commandes internes** *(built-in commands)* sont compilées à l'intérieur du noyau Matlab. Elle ne nécessitent donc pas de chargement de fichier supplémentaire.

- **Les commandes externes** sont contenues dans des fichiers d'extension **`.m`** et de même nom que la commande. Par exemple, la commande **`spy`** est stockée dans le fichier `spy.m`.

De fait, lorsque vous tapez une commande au prompt de Matlab, 4 comportements différents sont possibles :

1) La commande correspond en fait à un nom de variable, auquel cas la valeur de celle-ci est immédiatement affichée.

2) La commande est interne ; elle est immédiatement reconnue par le noyau de Matlab et exécutée

3) La commande n'est pas reconnue comme étant interne ; Matlab va donc la considérer comme externe et rechercher dans les fichiers .m qui sont à sa disposition. Pour cela, il utilise une liste de répertoires que l'on peut obtenir en invoquant la commande path. Notons que si une commande externe a déjà été exécutée précédemment, il y a de grandes chances pour qu'elle soit encore chargée en mémoire. Ceci peut avoir une conséquence fâcheuse : dans certains cas, Matlab ne se rend pas compte que le source d'une fonction a changé et oublie de recharger le fichier source. Nous verrons qu'il est possible de palier à ce comportement à l'aide de la commande `delete`.

4) Matlab ne trouve aucun fichier correspondant au nom de la fonction invoquée et émet une erreur, par exemple :

```
» tklp???
Undefined function or variable tklp.
»
```

5) La commande **path** affiche le contenu de la variable d'environnement MATLABPATH, laquelle peut être définie, soit dans l'environnement du système d'exploitation sous-jacent, soit dans le fichier matlabrc.

Par exemple :

```
» path

                MATLABPATH

        h:\matlab\toolbox\local
        h:\matlab\toolbox\optim
        h:\matlab\toolbox\matlab\datafun
        h:\matlab\toolbox\matlab\elfun
        h:\matlab\toolbox\matlab\elmat
        h:\matlab\toolbox\matlab\funfun
        h:\matlab\toolbox\matlab\general
        h:\matlab\toolbox\matlab\color
        h:\matlab\toolbox\matlab\graphics
        h:\matlab\toolbox\matlab\iofun
        h:\matlab\toolbox\matlab\lang
        h:\matlab\toolbox\matlab\matfun
        h:\matlab\toolbox\matlab\ops
        h:\matlab\toolbox\matlab\plotxy
        h:\matlab\toolbox\matlab\plotxyz
        h:\matlab\toolbox\matlab\polyfun
        h:\matlab\toolbox\matlab\sounds
        h:\matlab\toolbox\matlab\sparfun
        h:\matlab\toolbox\matlab\specfun
        h:\matlab\toolbox\matlab\specmat
        h:\matlab\toolbox\matlab\strfun
        h:\matlab\toolbox\matlab\dde
        h:\matlab\toolbox\matlab\demos
        h:\matlab\obc

»
```

Dans le monde Matlab, on appelle `toolbox` un répertoire contenant des fichiers `.m`. Chaque toolbox vendue avec matlab correspond à un domaine d'aide particulier.

1.3 La ligne de commande Matlab

Celle-ci vous permet d'entrer vos commandes. Rappelons que Matlab est sensible à la casse. Les commandes d'édition sont simples :

- Flêche haut vous permet de récupérer les commandes précédentes
- Flêche bas vous permet de redescendre dans l'historique des commandes
- Flêche gauche, Flêche droite, backspace et delete vous permettent de modifier la ligne de texte courante

Toute commande entrée sur la ligne de commande donne lieu à l'affichage de son résultat *à moins qu'elle ne se termine par un point virgule*, auquel cas l'affichage est supprimé.

Il est possible de saisir une commande sur plusieurs lignes, à condition d'utiliser 3 points (…) comme caractères de suite. Par exemple :

```
»  1/2 + 1/3 + 1/4 ...
+ 1/5 + 1/6

ans =
    1.4500
»
```

Lorsque vous tapez une expression simple sur la ligne de commande, le résultat de celle-ci est automatiquement affecté à la variable prédéfinie **ans**.

De manière générale, une ligne de commande Matlab est de la forme :

```
variable(s) = expression
```

où `expression` est une commande Matlab quelconque.

Par exemple :

» x=0.5
x =
  0.5000
» y=**cos**(1-x^2)
y =
  0.7317
	» v=[x y]
	v =
	  0.5000 0.7317
	«

Si les deux premiers cas se passent de commentaire, le troisième montre comment générer un 2-vecteur en concaténant les variables `x` et `y` à l'aide de l'opérateur **[ ]**. Notez que le caractère de séparation des éléments à l'intérieur d'un vecteur est l'espace. Nous reviendrons en détail sur les tableaux car ils constituent l'élément fondamental de Matlab.

1.4 Outils à usage général

1.4.1 Variables et constantes prédéfinies

Matlab met à disposition de l'utilisateur quelques variables et constantes prédéfinies.

`Inf` dénote ∞. C'est le résultat produit par une opération du style 1 / 0. Par exemple :

```
« 1/0
Warning: Divide by zero
ans =
   Inf
«
```

`NaN (Not a Number)` est le résultat typiquement renvoyé par 0/0 ou ∞/∞. Par exemple :

```
« 0/0
Warning: Divide by zero
ans =
   NaN
«
```

`pi` quotient du périmètre d'un cercle par son diamètre (qui l'eut cru ?) et dont la valeur vaut approximativement π.

`i` et **`j`** bases de l'espace complexe. Au cas où, par inadvertance malheureuse, vous définiriez des variables de même nom, il est toujours possible d'accéder à celles ci en utilisant **`ii`** et **`jj`**.

1.4.2 Contrôle de la précision des calculs

La tolérance du « zéro machine », est fixée par la variable standard **`eps`**. C'est-à-dire que pour certaines opérations telles que la recherche des valeurs singulières d'une matrice, par exemple, les nombres `x` tels que : **`-eps`** `< x <` **`eps`**
seront considérés comme nuls.

En modifiant cette borne, vous pouvez renforcer ou abaisser la sévérité des tests de nullité. La valeur standard vaut :

```
« eps
eps =
  2.2204e-016
«
```

1.4.3 Contrôle du format d'affichage des nombres

La commande **`format`** permet d'influer sur le format d'affichage des nombres dans Matlab.

L'invocation de **`help format`** renvoie la liste des formats utilisables. Il est possible de spécifier l'utilisation de formats en virgule fixe, en notation scientifique, héxadécimale, ainsi qu'un format spécial destiné à représenter les matrices de façon très condensée une matrice. L'exemple suivant permettra de fixer les esprits :

<table>
<tr>
<td>

```
« a = [ -1 2 -1 ; 1 0 -1 ; …
      -1 -1 1]
a =
    -1     2    -1
     1     0    -1
    -1    -1     1
```

</td>
<td>

```
« format +
« a
a =
-+-
+ -
--+
«
```

</td>
</tr>
</table>

La matrice `a` est d'abord affichée en format normal avant que l'on utilise la commande **`format +`**. L'affichage utilise alors le formalisme suivant :

- Un espace code une valeur nulle
- Le caractère **+** code une valeur positive
- Le caractère **–** code une valeur négative

Ce formalisme (qui doit ensuite être désactivé avec une autre commande **`format`**) est particulièrement intéressant lorsque l'on souhaite visionner l'allure d'une grande matrice de valeurs centrées, c'est-à-dire, principalement, en analyse de données.

La figure suivante montre le résultat de l'invocation de **`help format`**.

```
« help format

 FORMAT Set output format.
        All computations in MATLAB are done in double precision.
        FORMAT may be used to switch between different output
        display formats as follows:
          FORMAT         Default. Same as SHORT.
          FORMAT SHORT   Scaled fixed point format with 5 digits.
          FORMAT LONG    Scaled fixed point format with 15 digits.
          FORMAT SHORT E Floating point format with 5 digits.
          FORMAT LONG E  Floating point format with 15 digits.
          FORMAT HEX     Hexadecimal format.
          FORMAT +       The symbols +, - and blank are printed
                         for positive, negative and zero elements.
                         Imaginary parts are ignored.
          FORMAT BANK    Fixed format for dollars and cents.
          FORMAT COMPACT Suppress extra line-feeds.
          FORMAT LOOSE   Puts the extra line-feeds back in.
          FORMAT RAT     Approximation by ratio of small integers.
«
```

1.4.4 La gestion de l'espace de travail

La notion d'espace de travail est fondamentale en Matlab. La fenêtre de commandes de Matlab est un exemple d'espace de travail. Celui-ci est déterminé par l'ensemble des variables et des fonctions présentes en mémoire. C’est une notion assimilable à celle d’espace de visibilité (scope) dans les langages de programmation modernes.

Lorsque vous demandez l'exécution d'un fichier **.m**, ce dernier invoque un nouvel espace de travail vierge (à l'exception des variables *globales* dont nous reparlerons plus tard) qu'il remplit avec ses variables propres. Lorsque son exécution se termine, son espace de travail meurt avec lui. Ainsi, chaque fichier .m contient son propre espace de nommage. Il n'y a donc aucun danger à utiliser des noms courants pour les variables locales.

1.4.5 Gestion de la mémoire

Matlab fournit des primitives permettant de gérer la mémoire occupée par un utilisateur :

- Affichage de la liste des variables en mémoire
- Suppression de variables, fonctions, etc ... présentes en mémoire

1.4.5.1 Affichage de l'espace mémoire occupé

La commande `who` permet de connaître à un instant donné la liste des variables utilisateur. Par exemple :

```
« who

Your variables are:

a          v          x          y

«
```

Cette liste est utile lorsque l'on souhaite connaître la liste des identificateurs utilisés (par exemple, au moment de créer une nouvelle variable). Toutefois, les informations proposées sont trop limitées pour permettre de gérer la mémoire du système. Dans ce cas, on préférera la commande `whos` dont nous donnons immédiatement un exemple :

```
« whos

Name        Size        Elements      Bytes    Density   Complex

   a       3 by 3              9         72       Full       No
   v       1 by 2              2         16       Full       No
   x       1 by 1              1          8       Full       No
   y       1 by 1              1          8       Full       No

Grand total is 13 elements using 104 bytes

«
```

Pour chaque variable on obtient les informations suivantes :

- L'identificateur
- La dimension sous la forme Lignes **by** Colonnes. Un scalaire est donc de dimension `1 by 1`.
- Le nombre d'éléments. Cette information peut paraître redondante par rapport à la précédente car l'on s'attend à ce que le nombre d'éléments soit égal au nombre de lignes multiplié par le nombre de colonnes. En fait, cette information est utile lors de la gestion des matrices creuses où elle fait référence au nombre de cases réellement pleines.
- Le nombre d'octets utilisés
- La densité (exprimée en pourcentages pour les matrices creuses ou par l'identificateur **Full**) donne le rapport entre le nombre d'éléments utilisés et le nombre d'éléments théorique d'un tableau.
- Finalement, la dernière colonne indique si l'identificateur fait référence à des données complexes ou réelles.

1.4.5.2 Suppression de variables

La commande **clear** permet de supprimer des objets de l'espace de travail. Utilisée seule, elle détruit *l'ensemble des variables* présentes en mémoire. Comme nous l'indique **help clear**, d'autres syntaxes sont disponibles :

```
CLEAR  Clear various quantities from the workspace.
        CLEAR removes all variables from the workspace.
        CLEAR VARIABLES does the same thing.
        CLEAR X removes variable or function X from the workspace.
        CLEAR FUNCTIONS removes all compiled M-functions.
        CLEAR MEX removes all links to MEX-files.
        CLEAR ALL removes all variables, functions and MEX links.

        If X is global, CLEAR X removes X from the current workspace,
        but leaves it accessible to any functions declaring it global.
        CLEAR GLOBAL X completely removes the global variable X.
        CLEAR GLOBAL removes all global variables.
```

Les liens *MEX* permettent d'écrire des fonctions utilisables depuis Matlab vers des modules écrits avec d'autres langages de programmation.

Il est parfois très intéressant de « décharger » une fonction utilisateur. En effet, lorsque l'on met au point une fonction, l'on effectue souvent de fréquents aller-retours entre l'éditeur et l'environnement Matlab afin de vérifier le fonctionnement de la fonction. Et, régulièrement, Matlab ne se rend pas compte que le code a changé et relance une version non mise à jour de la fonction à partir de son cache, plongeant habituellement l'utilisateur dans la plus profonde perplexité. Afin de palier à cet inconvénient majeur et, ainsi, éviter les maux de tête en résultant, il est possible de supprimer du cache la (mau)dite fonction en utilisant **clear**.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

1.4.5.3 Les variables globales

Par défaut, les variables sont locales à un espace de travail. La commande **global** permet toutefois de définir des variables globales, c'est à dire dont la portée n'est pas limitée à un espace de travail (par exemple, à un fichier .m) mais à toute une session de travail. Si ce mécanisme permet de partager facilement des données, il doit être utilisé avec parcimonie (et Abonessian) et avec des identificateurs judicieusement choisis afin d'éviter tout risque de collision sur le nommage des variables.

Les commandes suivantes permettent d'influer sur les variables globales :

global X Y Z définit les variables globales X, Y et Z

isglobal(X) renvoie TRUE si et seulement si la variable X est globale

who global, **whos global**, **clear global x**, **clear global**
agissent de la manière que **who**, **whos** et **clear** mais sur les variables globales.

Il est à noter que les variables globales apparaîssent sans distinction particulière lorsque l'on utilise **who** ou **whos** comme sur l'exemple suivant :

```
« a = [ 1 2 3 ]
a =
  1     2     3
« b = 3.4
b =
  3.4000
« global c
« c=1
c =
  1
« whos
       Name          Size         Elements      Bytes     Density   Complex

          a        1 by 3                3         24        Full        No
          b        1 by 1                1          8        Full        No
          c        1 by 1                1          8        Full        No

Grand total is 5 elements using 40 bytes

« whos global
       Name          Size         Elements      Bytes     Density   Complex

          c        1 by 1                1          8        Full        No

Grand total is 1 elements using 8 bytes

«
```

1.4.5.4 Sauvegarde et chargement de l'environnement de travail

La commande **save** permet de sauvegarder tout (par défaut) ou partie de l'espace de travail. Ces options permettent de spécifier :

- Le nom d'un fichier pour la sauvegarde des informations

- La liste des variables à sauvegarder
- Le format de sauvegarde. Par défaut Matlab sauvegarde l'état de l'espace de travail dans un fichier binaire. Toutefois, il est possible de lui spécifier de tout sauver sous format ASCII et même d'indiquer le nombre de chiffres significatifs à inclure.

Réciproquement la commande `load` permet de récupérer un espace de travail *à condition que l'extension du fichier spécifié en paramètre soit* `.mat`. Si l'extension est différente, par exemple un fichier abc.cde, alors Matlab considère que le fichier contient une matrice sous forme ASCII à réintégrer dans l'expasce de travail sous le nom de variable `abc`.

Finalement, la commande `diary` permet de sauvegarder la quasi-intégralité d'une session de travail sous Matlab, ce qui inclut la totalité des lignes de commande et la plus grande partie de leur résultat.

2. Les opérations élémentaires dans Matlab

2.1 Types de données

Matlab ne connaît qu'un seul type de données : les matrices de réels. Ainsi, un scalaire est une 1×1-matrice et un n-vecteur une $1 \times n$-matrice.

2.2 Opérations arithmétiques élémentaires

Matlab dispose du jeu complet d'opérations arithmétiques élémentaires sur les réels :

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
\	Division inverse (a \ b = b / a) inutile sur les réels seuls … mais diablement efficace avec des tableaux !
^	Mise à la puissance

… où les opérations sont données dans l'ordre croissant des priorités. Bien entendu, il est possible d'utiliser des expressions parenthésées pour modifier l'ordre d'évaluation.

2.3 Opérations sur les matrices

Dans cette partie, nous entendons le mot matrice au sens large, c'est à dire qu'il pourra signifier indifféremment matrice ou vecteur en l'absence d'indications plus précises.

Matlab étant orienté matrices, il était tout naturel qu'il propose un nombre impressionnant de fonctionnalités les concernant. Commençons tout d'abord par les opérations de saisie :

2.3.1 Saisie d'un vecteur

La première manière de saisir un vecteur consiste tout simplement à énumérer ses composantes, séparées par des espaces et entre des crochets. Par exemple :

```
« v=[1 2.0 4.3]

v =

    1.0000    2.0000    4.3000

«
```

Il est également possible d'utiliser des expressions là où l'on trouvait des constantes :

```
« v = [1.0 sqrt(2.0) cos(1.0 - 2.0 * sin (pi/4))]

v =

    1.0000    1.4142    0.9154

«
```

Une des particularités les plus intéressantes concerne toutefois l'utilisation de l'opérateur **:**. Celui-ci permet de créer des énumérations de valeurs sous la forme de boucles implicites. Par exemple :

```
« v=1:5
v =
     1     2     3     4     5
« v=1:0.2:2
v =
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
«
```

La syntaxe générale est la suivante :

debut [: pas] : fin

Bien entendu, il est possible de mixer différentes formes d'initialisation, comme le montre l'exemple suivant :

```
« v=[1 2 4:6 5:0.1:5.5]

v =

  Columns 1 through 7

    1.0000    2.0000    4.0000    5.0000    6.0000    5.0000
5.1000

  Columns 8 through 11

    5.2000    5.3000    5.4000    5.5000
```

Il est également possible de créer des vecteurs uniformément répartis avec la fonction **`linspace`** ou avec des intervalles logarithmiques en utilisant **`logspace`**. Ces deux fonctions prennent en argument, le premier et le dernier élément de la répartition ainsi que le nombre d'éléments à générer. Par exemple :

```
« logspace(1,4,5)
ans =
   1.0e+04 *
    0.0010    0.0056    0.0316    0.1778    1.0000

« linspace(1,2,5)
ans =
    1.0000    1.2500    1.5000    1.7500    2.0000
«
```

L'exemple précédent montre comment Matlab peut utiliser un facteur commun (en l'occurrence 10000) au moment d'afficher un vecteur.

2.3.2 Saisir des matrices

L'exemple (simpliste !) suivant montre comment saisir une matrice toute bête avec Matlab.

```
« A = [ 1 2 3 ; 4 5 6 ; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

«
```

Basiquement, il s'agit de spécifier chaque ligne comme un vecteur. L'opérateur **`;`** agissant alors comme un opérateur de concaténation ligne à ligne.

Bien entendu, on peut utiliser des boucles implicites :

```
« A = [1:3 ; 4:6 ; 7:9]

A =

     1     2     3
     4     5     6
     7     8     9

«
```

Il est également possible de construire une matrice à partir de vecteurs existants, que ce soit par ligne ou par colonne comme sur l'exemple suivant :

« v1=[1 2 3] v1 = 1 2 3 « v2=[4 5 6] v2 = 4 5 6 « v3=[7 8 9] v3 = 7 8 9	« a1=[v1 ; v2 ; v3] a1 = 1 2 3 4 5 6 7 8 9 « a2=[v1' v2' v3'] a2 = 1 4 7 2 5 8 3 6 9 «

On voit bien ici que ; est un opérateur de concaténation par lignes alors que l'espace est un opérateur de concaténation par colonnes. A titre d'exercice, pouvez vous deviner ce que donneraient les commandes suivantes :

2.3.3 Obtenir la taille d'une matrice

Deux commandes permettent de connaître la taille d'une matrice :

size renvoie un tableau dont le premier élément est le nombre de lignes et le second, le nombre de colonnes. Au cas où l'on applique **size** à un vecteur, cette commande renvoie un tableau dont l'unique élément indique la taille du vecteur.

length renvoie uniquement, sous forme scalaire, le nombre de colonnes d'une matrice ou le nombre d'éléments d'un vecteur.

Exemple d'utilisation de **size** et de **length** :

« **size**(a1) ans = 3 3»	« **length**(a1) ans = 3»

2.3.4 Construction de matrices particulières

eye(n)	Identité $n \times n$
zeros(n,m)	Matrice nulle $n \times m$
ones(n,m)	Matrice $n \times m$ dont tous les éléments valent 1
rand(n,m)	Matrice $n \times m$ créée aléatoirement selon une loi uniforme entre 0 et 1
randn(n,m)	Matrice $n \times m$ créée aléatoirement selon une loi gaussienne centrée et réduite

Exemple : construction d'une matrice identité de même taille qu'une matrice existante.

```
« ida=eye(size(a1))

ida =

     1     0     0
     0     1     0
     0     0     1

«
```

2.3.5 Opérations entre tableaux

Les opérations arithmétiques présentées précédemment sont utilisables sur les tableaux … à condition de respecter les dimensions.

+ et **–** les dimensions des tableaux doivent être identiques

***** les dimensions internes des tableaux doivent être identiques. En particulier, on verra que pour multiplier une matrice à droite par un vecteur, il faut transposer ce dernier pour qu'il devienne une colonne.

Les opérateurs / \ servent à résoudre les systèmes les systèmes linéaires. L'exemple suivant illustre l'utilisation de * et \ avec des matrices.

```
«  a = [ 3 0 1 ; 0 5 -2 ; 1 -2 7 ]
a =

     3     0     1
     0     5    -2
     1    -2     7
« x = [ 1 2 3 ]
x =

     1     2     3
« b = a * x
??? Error using ==> *
Inner matrix dimensions must agree.
« b = a * x'
b =

     6
     4
    18
« y = a \ b
y =

    1.0000
    2.0000
    3.0000

« y = b / a
??? Error using ==> /
Matrix dimensions must agree.
« y = b' / a

y =
    1.0000    2.0000    3.0000
```

Création d'une matrice et d'un vecteur reconnaissables

La multiplication d'une matrice par un vecteur exige que celui-ci soit sous forme d'une colonne ⇨ transposition obligatoire

La résolution du système linéaire passe par l'utilisation des opérateurs \ et / selon que l'on souhaite mettre la matrice à gauche ou à droite.

2.3.6 Opérations entre matrices et scalaires

Matlab gère convenablement les opérations mettant aux prises un tableau avec scalaire. Si + - et * ne posent pas de problème particulier, il est interdit d'utiliser / et \. Nous verrons qu'il faut pour cela utiliser les opérations élément par élément.

2.3.7 Les opérations élément par élément (ou, au sens de Hadamar)

Ce sont des opérateurs qui agissent élément par élément sur une matrice. Leur signe caractéristique est précédé d'un point

Si .+ et .- sont rigoureusement identiques à + et – (l'addition et la soustraction se font toujours membre à membre) il n'en est pas de même pour * et /. En effet la matrice c = a .* b a pour terme général $c_{ij}=a_{ij} \times b_{ij}$ et c = a./b a pour terme général $c_{ij}=a_{ij}/b_{ij}$.

Ces opérateurs sont particulièrement utiles pour calculer des valeurs tabulées de fonctions. Par exemple : $\frac{1}{1+x^2}$

```
« x = [0.5 : 0.1 : 1];
« y = 1.0 / (1+x.*x)
??? Error using ==> /
Matrix dimensions must agree.

« y = 1.0 ./ (1+x.*x)

y =

    0.8000    0.7353    0.6711    0.6098    0.5525    0.5000
```

L'utilisation directe de la division n'est pas possible entre un scalaire et un vecteur, il faut utiliser la division membre à membre

2.3.8 Transformation d'une matrice en vecteur

Parfois, il peut être agréable de transformer une matrice en vecteur (pour effectuer certains calculs plus simplement par exemple). Dans ce cas, on utilise une version spéciale de l'opérateur d'indexation qui range dans un vecteur colonne les éléments d'une matrice *colonne par colonne*. En fait, cet opérateur ne fait rien de spécial sinon refléter l'organisation mémoire de Matlab, laquelle est fortement inspirée du FORTRAN. Si vous désirez stocker une matrice dans un vecteur mais ligne par ligne par colonne, alors, il faut transposer au préalable.

Exemple :

```
« a = [ 1 2 ; 3 3 ]

a =

     1     2
     3     4

« b = a'
```

```
« a(:)

ans =

     1
     3
     2
     4

« b(:)
```

b = 1 3 2 4	ans = 1 2 3 4

2.3.9 Modification du rangement des éléments dans la matrice

Il existe deux autres opérations de rangement sur les matrices qui sont **fliplr** et **flipud** qui permettent respectivement d'inverser l'ordre et celui des colonnes de la matrice. Notez au passage que **fliplr** permettra de retourner un vecteur. Exemple :

« a a = 1 2 3 4 5 6 7 8 9	« **fliplr**(a) ans = 3 2 1 6 5 4 9 8 7	« **flipud**(a) ans = 7 8 9 4 5 6 1 2 3

2.3.10 Quelques opérations sur les matrices

Suit la liste de quelques unes des opérations les plus courantes réalisées sur des matrices.

inv	Inverse la matrice
rank	Rang de la matrice
cond	Nombre de conditionnement de la matrice
norm	Norme 2 de la matrice. Il est possible d'obtenir les autres normes en passant des paramètres supplémentaires.
mean	Moyenne de chaque colonne
std	Ecart type de chaque colonne
sum	Somme de chaque colonne
cumsum	Somme cumulée de chaque colonne
prod	Produit des éléments de chaque colonne
cumprod	Produit cumulé des éléments de chaque colonne
min, max, median	Minimum, maximum et médiane des éléments de chaque colonne

Toutes ces opérations s'appliquent également à un vecteur. Si vous désirez effectuer ces opérations non pas sur les colonnes mais sur les lignes, vous avez toujours la possibilité de transposer la matrice au préalable. De même il est possible d'appliquer l'opérateur à l'ensemble des éléments de la matrice en utilisant l'opérateur de transformation en vecteur.

Il faut prendre garde à l'opération de mise à l'exponentielle d'une matrice. En effet, comme l'on peut s'y attendre **`exp(a)`** est la matrice des exponentielles de chacune de ses composantes. Pour prendre l'exponentielle d'une matrice, il faut utiliser l'une des fonctions **`expm`**, **`expm1`**, **`expm2`** ou **`expm3`**. De même **`logm`** et **`sqrtm`** calculent respectivement le logarithme et la racine carrée matricielle.

3. Opérateurs généralistes de Matlab

3.1 Opérateurs de comparaison

Ils renvoient des valeurs booléennes où 0 signifie faux et 1 vrai.

Opérateur	Signification
`==`	Egalité
`<` **`>`** **`<=`** **`>=`**	Inégalités
`~=`	Différence

On pourra comparer :

- 2 scalaires
- 2 tableaux de même taille
- 1 tableau et un scalaire

Tous ces opérateurs renvoient un tableau de même taille que l'argument principal, par exemple :

```
v = [ 1 2 3 4 5 ]

v =

     1     2     3     4     5

» a = [1 2 3 ; 4 5 6 ; 7 8 9]

a =

     1     2     3
     4     5     6
     7     8     9

» v > 4

ans =

     0     0     0     0     1
```

```
» a == 2
ans =

     0     1     0
     0     0     0
     0     0     0

» a <= 3
ans =

     1     1     1
     0     0     0
     0     0     0

» a <= 5

ans =

     1     1     1
     1     1     0
     0     0     0
```

Donnons maintenant un exemple certes surprenant mais néanmoins classique dans l'environnement Matlab de l'utilisation des opérateurs de comparaison pour résoudre un problème d'indétermination au dénominateur. La fonction sinus cardinal a pour expression $\mathrm{sinc}(x) = \begin{cases} 1 & x = 0 \\ \dfrac{\sin x}{x} & x \neq 0 \end{cases}$

Il n'est pas possible de l'écrire `sin(x)./x` directement sur un intervalle contenant l'origine à cause de l'indétermination en 0, aussi il est possible de résoudre le problème en procédant en 2 étapes :

a) on élimine le 0 au dénominateur pour le remplacer par un 1, ce qui revient à remplacer x par x+(x==0). En effet, l'expression (x==0) vaudra 0 en tout autre point que 0 et 1 en 0. Ainsi, on ne modifie x qu'au point 0.

b) On effectue la division avec le dénominateur modifié, ainsi la fonction vaudra sin(0)/1 en 0, soit 0

c) on ajoute ensuite la valeur 1 au point 0, et pour ce faire, on réutilise une addition avec (x==0)

Finalement on obtient :

sin(x)./(x+(x==0))+(x==0)

Cette manière de procéder peut paraître lourde mais elle est nettement plus efficace que l'utilisation d'une boucle sur les valeurs de x.

3.2 Opérateurs logiques

Ce sont des opérateurs booléens qui agissent sur des tableaux de 0 et de 1. Sont disponibles :

`&`	Et logique
`|`	Ou logique
`xor`	Ou exclusif
`~`	Non logique

3.3 Autres fonctions de test

Les fonctions is… et apparentées renvoient un tableau booléen de même taille que leur argument en fonction des valeurs qu'il contient

`isinf`	1 si valeur = -inf ou +inf
`finite`	Vrai pour toute valeur numérique différente de l'infini ou de not a number
`isnan`	Vrai si la valeur vaut not a number

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

find	Vrai si la valeur est non nulle

Deux autres opérateurs sont un peu différents car ils renvoient une valeur scalaire unique

isempty	Vrai si le tableau est non vide
isreal	Vrai si le tableau contient des réels, c'est à dire qu'il ne contient pas de complexes

Find est un autre opérateur particulier qui renvoie les coordonnées des éléments non nuls. Il faut se méfier des valeurs renvoyées par **find** lorsqu'il travaille sur une matrice. En effet, **find** renvoie toujours un vecteur. Ainsi, pour une matrice 3x3, **find** renverra un vecteur à neuf éléments. Les indices renvoyés correspondent à un rangement par colonnes des valeurs. Il est possible d'obtenir un rangement par lignes en demandant explicitement deux arguments de retour. Par exemple, en réutilisant la matrice a de l'exemple précédent :

```
» a(:)

ans =

     1
     4
     7
     2
     5
     8
     3
     6
     9

» find (a <= 5)

ans =

     1
     2
     4
     5
     7
»
```

Les éléments inférieurs ou égaux à 5 ont pour rang 1, 2, 4, 5 et 7 dans le vecteur construit à partir de a

```
» [lig,col]=find(a<=5)

lig =

     1
     2
     1
     2
     1


col =

     1
     1
     2
     2
     3

»
```

Pour retrouver les éléments de a inférieurs ou égaux à 5, il faut considérer les coordonnées dans les deux tableaux lig et col.
Par exemple, si l'on considère les troisièmes éléments (en italique), on apprend que a(1,2) a une valeur inférieure à 5.

Maintenant que nous disposons de **find**, il est possible d'écrire la fonction sinus cardinal d'une nouvelle manière :

1) nous construisons un vecteur de 1 de même taille que x :y=**ones**(**size**(x))

2) nous recherchons les éléments non nuls de x indices=**find**(x)

3) puis, nous calculons **sin**(x)/x pour tous les éléments non nuls trouvés par **find**.

Nous obtenons finalement :

y(indices)=sin(x(indices))./(x(indices))

Deux autres opérateurs booléens très pratiques sont spécifiques à Matlab

`all(a)` renvoie vrai si tous les éléments du vecteur a sont non nuls. Appliqué à une matrice, il renvoie un vecteur effectuant la même opération pour chaque colonne de la matrice

`any(a)` renvoie vrai s'il existe un élément non nul dans le vecteur. Il fonctionne de la même manière que all sur une matrice.

Exemple : test d'égalité sur deux matrices : all(all(a==b)) doit être égal à 1

Exercice : test de symétrie d'une matrice

4. Quelques fonctions régulièrement utilisées

Loin d'être exhaustif, ce chapitre propose une revue de quelques fonctions de Matlab régulièrement utilisées et permettant de se familiariser

4.1 Décompositions de matrices

Rien ne saurait remplacer la lecture **`help`** `matfun` pour une documentation complète sur les opérations disponibles sur les matrices. Nous ne donnons ici, à titre d'exemple que quelques unes des opérations de décomposition.

4.1.1 Décomposition LU

`[L,U]=lu(A)` où U est triangulaire supérieure, L quasiment triangulaire inférieure à une permutation sur A près et A=L*U. Pour avoir réellement une matrice inférieure, il faut demander le troisième argument de retour de Matlab

`[L,U,P]=lu(A)` Ici U est triangulaire supérieure, L triangulaire inférieure avec des 1 sur la diagonale et l'on a : L*U=P*A

Exercice : inverser une matrice A en utilisant sa décomposition LU sous la deuxième forme

4.1.2 Décomposition QR

`[Q,R]=qr(A)` où Q est une matrice orthonormale et R une matrice triangulaire supérieure telles que Q*R = A.

Bien que très stable numériquement, il est possible d'améliorer encore les performances de cette méthode en demandant le 3ème paramètre de retour :

[Q,R,P]=qr(A) où Q est une matrice orthonormale, R une matrice triangulaire supérieure et P une matrice de permutation telles que Q*R=A*P. La principale amélioration tient au fait que les valeurs diagonales de R sont rangées par ordre décroissant en valeur absolue, phénomène connu pour améliorer la performance des opérations utilisant les matrices triangulaires supérieures, en particulier la résolution de systèmes linéaires (plus l'on divise par une valeur grande, moins l'on commet d'erreurs).

4.2 Valeurs et vecteurs propres

Matlab dispose d'une fonction particulièrement efficace permettant de trouver la liste des valeurs propres, éventuellement accompagnées de leurs vecteurs propres correspondants. L'extraction des valeurs propres se fait par la méthode QR accélérée. Une étude menée par l'auteur montre que l'algorithme employé est très efficace hormis pour les matrices tri diagonale où la méthode spécifique à ce type de matrice l'emporte. L'exemple suivant illustre cette fonction nommée **eig** d'après le mot eigenvalue, valeur propre en Anglais. La syntaxe de **eig** peut paraître spéciale.

eig(a) renvoie un vecteur colonne des valeurs propres de la matrice

[vecteurs, valeurs]=eig(a) renvoie une base de vecteurs propres et une matrice carrée dont les éléments diagonaux sont les valeurs propres correspondants.

```
« a = [1 2 -1 4 ; 2 3 1 0 ;…
      -1 1 -2 3 ; 4 0 3 -5]

a =

     1     2    -1     4
     2     3     1     0
    -1     1    -2     3
     4     0     3    -5
```

```
« [vecteurs,valeurs]=eig(a)
vecteurs =

    0.2508    0.6247    0.6226    0.3990
    0.1077   -0.6854    0.7123   -0.1061
   -0.8863   -0.0622    0.1393    0.4372
   -0.3740    0.3689    0.2925   -0.7990

valeurs =

   -0.5726         0         0         0
         0    1.2677         0         0
         0         0    4.9436         0
         0         0         0   -8.6386
```

4.3 Gestion des polynômes avec matlab

4.3.1 Représentation

Un polynôme est représenté sous la forme du vecteur de ses composantes par ordre *décroissant* de degré *i.e.* le membre de degré le plus haut est à gauche.

Par exemple $4x^3 - x + 5$ s'écrit [4 0 -1 5]

4.3.2 Evaluation

L'Evaluation des polynômes s'effectue bien évidemment par l'algorithme de Horner. La fonction à utiliser diffère selon que l'on souhaite évaluer le polynôme sur des valeurs scalaires ou sur une matrice.

polyval(p,x) évalue le polynôme p en chaque point du tableau x

polyvalm(p,a) ici l'on n'évalue pas sur chaque élément de la matrice a mais c'est la matrice qui devient la variable du polynôme.

```
» p = [3 0 -1 5]

p =

     3     0    -1     5

» x=[-1 0 1 ; 0 0 1 ; 2 1 -2]

x =

    -1     0     1
     0     0     1
     2     1    -2
```

```
» polyval(p,x)

ans =

     3     5     7
     5     5     7
    27     7   -17

» polyvalm(p,x)

ans =

   -21    -9    29
   -18    -1    20
    58    20   -59
»
```

4.3.3 Opérations élémentaires sur les polynômes

La multiplication classique de polynômes s'effectue par convolution des tableaux de coefficients, soit avec la fonction **conv**

R = p x q s'écrit donc r=**conv**(p,q)

La division euclidienne de polynômes (puissances décroissantes) s'effectue par déconvolution des tableaux de coefficients soit donc avec la fonction **deconv**.

Par exemple, examinons le cas des polynômes $x^2 + 3x - 1$ et $x - 4$ dont le produit vaut $x^3 - x^2 - 13x - 4$. La traduction en Matlab est la suivante :

```
« p1=[ 1 3 -1 ]

p1 =

     1     3    -1

« p2 = [ 1 -4]

p2 =

     1    -4

« p3 = conv(p1,p2)

p3 =
```

```
deconv(p3,p1)

ans =

     1    -4

« deconv(p3,p2)

ans =

     1     3    -1

«
```

1 -1 -13 4	

Après création des polynômes p1 et p2, on calcule le produit dans p3. Par chance, la division de p3 par p1 redonne p2 et réciproquement.

4.3.4 Réduction d'une fraction rationnelle en éléments simples du premier ordre

Matlab offre la fonction **residue** dont le but est de calculer la réduction en éléments simples de première espèce d'une fraction rationnelle. La syntaxe générale de ma fonction **residue** est la suivante :

[residues,poles,direct]=residue(numerateur,denominateur)

où :

direct est le polynôme de l'élément le plus simple

poles est la liste des pôles de degré 1 de la division

residus le vecteur des numérateurs associés aux pôles.

Il n'est pas possible d'obtenir la réduction d'une fraction rationnelle en éléments de deuxième espèce.

Considérons par exemple la fraction rationnelle : $\frac{x^3-6x^2+2x}{x^2+x-2}$ dont la décomposition s'écrit : $(x+5)+\frac{3}{x+2}-\frac{4}{x-1}$. Traduisons en Matlab cette opération :

```
> numerateur=[1 6 2 0]

numerateur =

     1     6     2     0

« denominateur=[1 1 -2]

denominateur =

     1     1    -2
```

```
« [residus,poles,direct]=
    residue(numerateur,denominateur)

residus =

    -4
     3


poles =

    -2
     1


direct =

     1     5
```

Le résultat contenu dans `direct` est donc le polynôme de degré 1 sans dénominateur, `poles` fournit la liste des *opposés* des monômes de degré 0 des dénominateur et `residus` les numérateurs associés.

4.3.5 Calcul des racines du polynôme

La fonction **roots** donne les racines réelles ou complexes du polynôme. Elle utilise la méthode de Müller puis affine les racines trouvées par la méthode de Bairstow.

4.3.6 Polynômes et matrices

La fonction **poly** permet d'extraire le polynôme caractéristique d'une matrice. L'exemple suivant illustre cette fonctionnalité, qui, couplée à **roots**, permet d'obtenir les valeurs propres d'une matrice. Toutefois, la qualité des solutions obtenue est très inférieure à celle obtenue par la méthode directe utilisant la fonction **eig**. Nous reprenons l'exemple précédent sur les valeurs propres.

```
» a = [1 2 -1 4 ; 2 3 1 0 ; -1 1 -2 3 ; 4 0 3 -5];
» p=poly(a)

p =

    1.0000    3.0000  -46.0000   27.0000   31.0000

» roots(p)

ans =

   -8.6386
    4.9436
    1.2677
   -0.5726
»
```

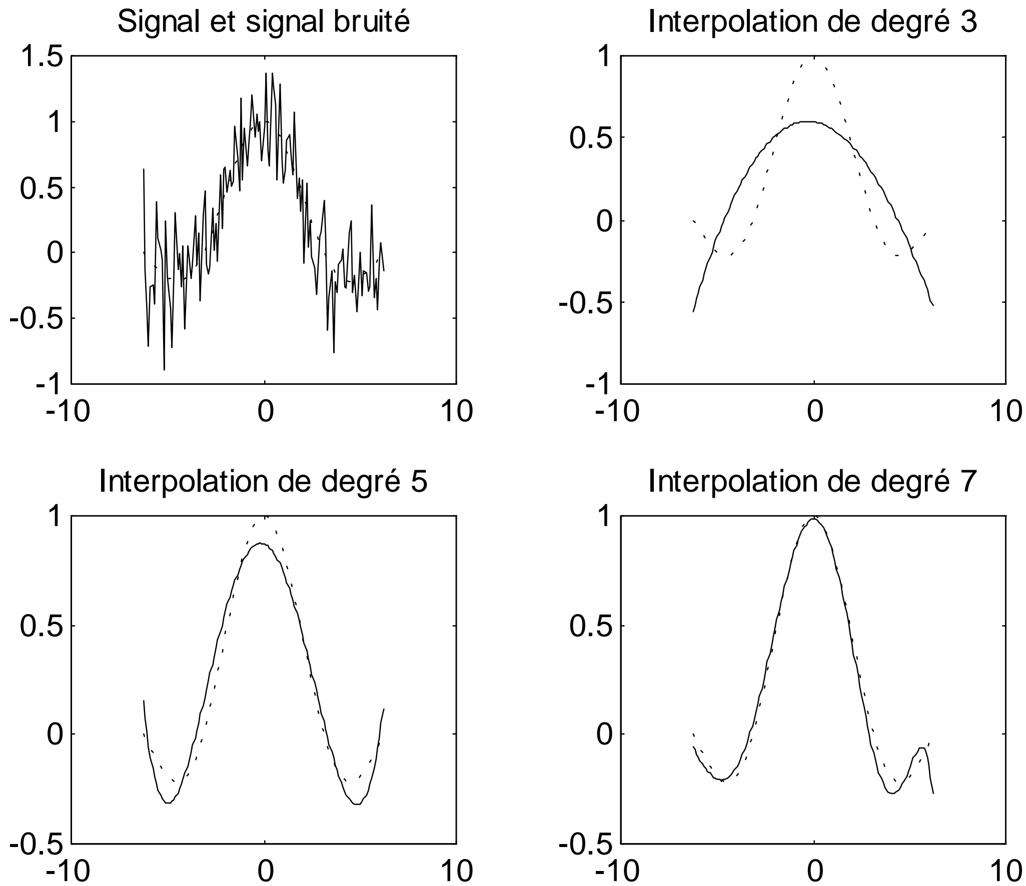
On retrouve bien les valeurs propres observés précédemment ! (Ouf ☺)

4.3.7 Interpolation polynomiale

Matlab propose des fonctions d'interpolation polynomiale au sens de Lagrange ou au sens des moindres carrés.

4.3.7.1 Interpolation polynomiale au sens des moindres carrés

Soient deux séries **x** et **y** telle que $y = f(x) + \delta$ où δ est un bruit de caractéristiques inconnues. Soit un entier **n**. La fonction **polyfit(x,y,n)** renvoie un polynôme de degré n approchant au mieux f au sens des moindres carrés. L'exemple graphique suivant expose cette fonctionnalité.



Code Matlab correspondant :

```
» x=[-2*pi:0.1:2*pi];
» y=sinc(x);
» bruite=y+randn(size(x))/4;
» p3=polyfit(x,bruite,3)

p3 =
   -0.0007   -0.0272    0.0137    0.6010

» p5=polyfit(x,bruite,5);

» p7=polyfit(x,bruite,7);

» subplot(2,2,1), plot(x,y,':',x,bruite,'-'), …
  title('Signal et signal bruité');
» subplot(2,2,2), plot(x,y,':',x,polyval(p3,x),'-'),…
  title('Interpolation de degré 3');
» subplot(2,2,3), plot(x,y,':',x,polyval(p5,x),'-'),…
  title('Interpolation de degré 5');
» subplot(2,2,4), plot(x,y,':',x,polyval(p7,x),'-'),…
  title('Interpolation de degré 7');
```

4.3.7.2 Interpolation polynomiale au sens de Lagrange

La fonction **`interp1(x,y,z)`** établit un polynôme d'interpolation des séries x et y et renvoie l'image de tous les points de z par ce polynôme. La fonction **`interp2(x,y,z,u,v)`** réalise le même travail mais en trois dimensions. Il est conseillé de consulter l'aide de Matlab pour la définition des options de ces fonctions.

4.3.8 Exercice sur les polynômes

Ecrire (en très peu de lignes) une fonction calculant la primitive qui s'annule en 0 d'un polynôme.

5. La programmation avec Matlab

Matlab propose en standard des fonctionnalités avancées de programmation.

5.1 Ecriture de sous programmes

Afin de pouvoir appeler un sous-programme de votre facture depuis la ligne de commande Matlab, il est important de le stocker dans un fichier portant le même nom. Attention donc aux environnements où la taille du nom d'un fichier est limitée.

La structure générale d'un sous programme Matlab est la suivante :

```
[liste des arguments de retour] = nomdelafonction (parametres)
```

L'une des particularités les plus étonnantes de Matlab réside dans le fait que les fonctions peuvent avoir plusieurs résultats. Si vous ne précisez pas de variable à laquelle affecter le résultat d'une fonction, seul le premier résultat est fourni. La fonction lu permettant de calculer la factorisation LU d'une matrice fournit un exemple typique, son utilisation est exposée en détails au chapitre ….

De même, les paramètres en entrée sont tous optionnels, il appartient donc au programmeur de vérifier qu'ils sont présents.

Les variables **`nargin`** et **`nargout`** contiennent respectivement le nombre d'arguments passés en entrée à l'appel de la fonction et le nombre de résultats demandés.

Il faut également se souvenir que toutes les variables que vous déclarez et utilisez à l'intérieur d'un sous-programme y sont locales. Si vous voulez renvoyer un résultat dans une variable il faut la rendre globale à l'aide de l'ordre **`global`** ; auquel cas, il faut faire attention aux éventuelles collisions de nommage.

Afin d'éviter que les résultats d'évaluation des différentes instructions d'un sous programme n'apparaissent dans la fenêtre Matlab, il est important de terminer chaque ligne par un point virgule. Toutefois, il est parfois utile, lorsque l'on débogue un programme de ne pas mettre ce dernier afin d'avoir une trace d'exécution sur les lignes les plus critiques.

5.2 Les structures de contrôle

Matlab propose les structures de contrôle les plus courantes : exécution conditionnelle et répétitive

5.2.1 Exécution conditionnelle

La structure générale d'une structure d'exécution conditionnelle (accompagnée d'un petit exemple) est la suivante :

if condition
  … instructions …
elseif condition
  … instructions …
else
  … instructions
end
	if (i<j)
	  a=a+b(i,j)
	else
	  a=a+b(j,i)
	end

Un point important mérite d'être signalé : l'expression `condition` se doit d'être *scalaire et booléenne*.

5.2.2 Exécution répétitive

Matlab dispose des deux structures de boucle les plus courantes : les boucles avec compteur (**`for`**) et les boucles conditionnelles (**`while`**).

Quel que soit le type de boucle choisi, il est possible d'utiliser l'instruction break pour sortir inconditionnellement de la boucle courante.

5.2.2.1 Structure générale de la boucle `while`

while condition
  … instructions …
end
	Exemple :
	while (e < eps)
	  e = … résultat d'un calcul
	end

La boucle **`while`** est très intéressante, par exemple, pour atteindre la convergence d'un processus.

5.2.2.2 Structure générale de la boucle `for`

```
for variable = debut [:pas]:fin
  … instructions …
end
```

Notons que le pas par défaut est de 1 et que la construction de la liste des valeurs possibles de la variable de contrôle se fait toujours de façon croissante. Pour parcourir des valeurs décroissants, il est nécessaire d'utiliser un pas négatif.

Cette boucle est très appréciée des débutants en Matlab car elle permet de transposer aisément un programme écrit dans un langage structuré classique. Toutefois, son emploi sur les tableaux doit être limité à son minimum avec Matlab. En effet, dans la plupart des cas, il est possible de remplacer une boucle for par des instructions directes sur les vecteurs ou sur les matrices.

Par exemple, considérons un sous programme qui calcule la somme des éléments sous diagonaux d'une matrice

La première méthode consiste à utiliser 2 boucles for imbriquées alors que la seconde utilise les fonctions étendues de Matlab sur les tableaux.

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

a =

     1     2     3
     4     5     6
     7     8     9
```

```
for i=1:length(a)
  for j=1:i-1
    s=s+a(i,j)
  end
end

s =
    19
```

```
for i=1:length(a)
  s=s+sum(a(i,1:i-1))
end

s =
    19
```

Sur une matrice de cette taille, la différence est minime. En revanche, sur des matrices plus conséquentes, la seconde méthode est bien plus rapide car elle tire partie des fonctionnalités du noyau de Matlab.

6. Manipulation des chaînes de caractères

L'aide concernant la manipulation des chaînes de caractères s'obtient par **help** strfun.

6.1 Représentation et fonctions élémentaires sur les chaînes

Dans Matlab, une chaîne de caractères est une séquence de caractères encadrée par des apostrophes, par exemple 'F4'. Pour insérer une apostrophe dans une chaîne, il suffit de la doubler.

En fait, elle est stockée sous la forme d'un tableau de caractères, aussi la plupart des opérations de base disponibles sur les vecteurs seront utilisables sur les chaînes. Par exemple, la longueur d'une chaîne s'obtient avec **length**. Toutefois, Matlab garde en mémoire un indicateur spécial indiquant si un tableau d'entier est en fait une chaîne de caractères. La fonction **isstr** renvoie 1 si son argument est une chaîne de caractères. Matlab propose de passer de la représentation chaîne de caractère à la représentation tableau de code ASCII et réciproquement à l'aide des fonctions **abs** et **setstr**.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

L'exemple de code suivant résume les commandes de base sur les chaînes de caractères.

```
» chaine='ISIMA F4.2'

chaine =

ISIMA F4.2

» isstr(chaine)

ans =

     1

» tableau=abs(chaine)

tableau =

  73   83   73   77   65   32   70   52   46   50
```

Transformation en tableau des codes ASCII
Notez que `isstr` du tableau renvoie faux

```
» isstr(tableau)

ans =

     0

» setstr([65 66 67])

ans =

ABC

»
```

Transformation d'un tableau de codes ASCII en chaîne

Exercices sur les chaînes de caractères : ressortir le partiel de l'an dernier.

6.2 Utilisation « avancée »

6.2.1 Conversions de chaînes vers des variables numériques

Les fonctions suivantes permettent de convertir une chaîne de caractères vers une valeur numérique.

num2str convertit une valeur numérique quelconque vers une chaîne. Typiquement prévue pour être utilisée sur un scalaire, le résultat obtenu à partir d'un vecteur mérite le détour !

int2str convertit un entier en chaîne ; si la chaîne représente en fait un nombre réel, **int2str** tronque sa partie décimale.

dec2hex transcrit un nombre entier vers une chaîne contenant sa représentation en base 16

mat2str renvoie une matrice sous la forme [valeurs ; valeurs ; …] c'est à dire la forme sous laquelle elle serait saisie sur la ligne de commande de Matlab. Cette fonction prend un second argument indiquant le nombre de décimales désirées.

str2num conversion d'une chaîne en nombre *scalaire*. Si la chaîne contient une opération, celle-ci est effectuée avant que son résultat ne soit renvoyé. Si la chaîne contient des crochets (et, éventuellement, des point virgules), le résultat sera sous forme d'un vecteur (matrice).

hex2num et **hex2dec** transforment respectivement une chaîne de caractères hexadécimaux vers un nombre flottant ou entier.

L'exemple suivant illustre ces fonctionnalités, souvent utiles lorsque l'on récupère des données en provenance d'un fichier texte.

6.2.2 La fonction `eval`

Elle permet d'évaluer une commande placée dans une chaîne de caractères. Contrairement à **`strnum`** qui ne pouvait réaliser que les opérations de base, **`eval`** peut appeler des sous programmes. Il est possible de passer à **`eval`** une seconde chaîne de caractères en second paramètre : elle sera évaluée en cas d'échec sur le premier paramètre.

6.2.3 Fonctions diverses

`isletter(s)` renvoie 1 pour chaque caractère alphabétique

`isspace(s)` renvoie 1 pour chaque caractère blanc

`upper(s)` transforme une chaîne en passant tous les caractères en majuscule

`lower(s)` transforme une chaîne en passant tous les caractères en minuscule

`blanks(n)` génération d'une chaîne de n espaces

`strcmp(s1,s2)` effectue une comparaison lexicographique des chaînes de caractères. Deux valeurs de retour sont possibles : 0 si les chaînes sont différentes et 1 si elle sont identiques. Si vous voulez effectuer une comparaison des valeurs des chaînes, utilisez préalablement **`eval`**

`strrep(cible, source, remplacement)` remplace dans `cible` toutes les occurrences de `source` par `remplacement`

`strtok(s)` renvoie le premier token de la chaîne. Si vous demandez un second résultat, celui-ci contiendra le reste de la chaîne ; ce qui s'avère très utile pour réaliser des appels successifs

`findstr(source, recherche)` recherche les positions de la chaîne `recherche` dans la chaîne `source` et renvoie le vecteur des positions trouvées.

L'exemple suivant illustre le fonctionnement de certaines de ces fonctionnalités.

<table>
<tr>
<td>

```
» chaine='Isima F4';
» isletter(chaine)

ans =

  1  1  1  1  1  0  1  0

» isspace(chaine)

ans =

  0  0  0  0  0  1  0  0

» upper(chaine)

ans =

ISIMA F4

» lower(chaine)

ans =

isima f4
```

</td>
<td>

```
» strcmp('ABCDE','ABEDC')

ans =

     0

» strcmp('ABCDE','ABCDE')

ans =

     1

» geographie='de l''est à l''ouest'

geographie =
de l'est à l'ouest

» strrep(geographie,'est','ours')
ans =

de l'ours à l'ouours

» findstr(geographie,'est')

ans =
     6    16
```

</td>
</tr>
</table>

Exemple d'utilisation de **strtok** sur une chaîne pour extraire tous les tokens.

<table>
<tr>
<td>

```
function tokenize(chaine)
  suite=chaine;
  z=[];
  while (~isempty(suite))
    [premier,suite]=strtok(suite);
    premier
  end
end
```

</td>
<td>

```
» tokenize('les trois éléments')
premier =
les

premier =
trois

premier =
éléments
»
```

</td>
</tr>
</table>

7. Graphisme avec Matlab

7.1 Généralités

Commençons par une précision importante. A l'heure où ce torchon prend vie, l'Ours Blanc des Carpathes ne dispose pas des dernières versions de Matlab. Il faut donc entendre que toutes les indications données ci-dessous s'entendent pour la version 4.2 de Matlab.

Matlab offre d'intéressantes possibilités de rendu graphique de données numériques, que cela soit en 2D ou en 3D. Notons immédiatement que les textes sont toujours affichés dans la même police et qu'il n'est pas possible d'en changer. En outre, nous ne donnerons ici qu'un bref aperçu des fonctions graphiques les plus fondamentales de Matlab.

7.1.1 Maintient des données affichées

Par défaut, tout ordre d'affichage de données efface la fenêtre d'affichage pour faire place nette à ces données. Il est possible de modifier ce comportement grâce à la commande `hold('on')` qui active le mode « rétention des données ». Une fois dans ce mode, tout ordre d'affichage superpose les nouvelles données à l'état actuel de la fenêtre. Les axes sont éventuellement mis à jour de manière à ce que toutes les données soient affichées. Ce fonctionnement persiste jusqu'à la prochaine commande `hold('off')`.

La commande `hold` utilisée sans paramètre agit comme bascule entre les deux modes.

7.1.2 Les chaînes de format

Il est possible de spécifier le format d'un tracé dans une chaîne de caractères. Celle-ci est composée de caractères spécifiant le style de tracé et la couleur. Voici la liste des codes correspondants :

Style de tracé		Couleur	
Code	**Signification**	**Code**	**Signification**
Dessin continu		`y`	Jaune
`-`	Trait plein	`g`	Vert
`:`	Pointillés	`w`	Blanc
`_ _`	Gros pointillés	`m`	Rose
`_ .`	Gros et petits pointillés en alternance	`b`	Bleu
Dessin discret		`k`	Noir
`o`	Petits ronds	`c`	Cyan
`*`	Etoiles	`r`	Rouge
`+`	Signes +		
`.`	Simples points		

Par exemple, la chaîne `'ow'` place le graphique en mode dessin discret et place des petits ronds blancs aux divers points de données.

7.1.3 Axes du graphique

Par défaut, Matlab affiche des axes permettant de se repérer facilement sur la figure. Ceux-ci sont gradués régulièrement.

Matlab offre la possibilité de modifier, dans une certaine mesure, les axes du graphique à l'aide de la commande `axis`. Celle-ci peut être utilisée sous un nombre impressionnant de formes différentes. Nous ne retiendrons ici que les principales.

`axis([xmin xmax ymin ymax])` spécifie l'étendue des axes des abscisses et des ordonnées.

axis([xmin xmax ymin ymax zmin zmax]) spécifie l'étendue des axes des abscisses, des ordonnées et des côtes pour un dessin en 3 dimensions.

axis('auto') permet de revenir aux étendues par défaut des axes, très pratique après des expérimentations peu concluantes ☺

axis('off') retire les graduations et les étiquettes des axes

axis('on') affiche les graduations et les étiquettes des axes

axis('AXIS') verrouille les axes dans leur position actuelle. S'emploie en mode `hold(on)`. On commence par afficher la série de valeurs de référence, puis on bloque les axes pour les prochaines. Cela permet de spécifier certains paramètres de centrage sur la série de référence.

Les commandes **xlabel** et **ylabel** et **zlabel** que nous présentons ci-dessous permettent d'associer des légendes aux axes. Il est possible d'afficher une grille avec la commande `grid`.

7.1.4 Titre, légendes et textes quelconques

title('chaine') permet de donner un titre à un graphique. Il est toujours placé au dessus du graphique et centré.

xlabel('chaine') place une légende sous l'axe horizontal

ylabel('chaine') place une légende à gauche de l'axe vertical. Le texte est écrit verticalement

zlabel('chaine') place une légende sur l'axe des côtes dans un graphique à trois dimensions.

Il n'existe pas de commande permettant de créer automatiquement une légende, comme, par exemple, avec Excel. Il est toutefois possible de placer un texte quelconque, soit directement sur la figure à l'aide de la souris par la commande `gtext('chaine')`, soit en utilisant ces coordonnées avec **text(x,y,'chaine')** où x et y sont les coordonnées du point le plus en bas à gauche du texte dans le système de coordonnées défini par les données du dessin.

7.1.5 Séparation en plusieurs sous graphiques

Il s'agit de la commodité que nous avons déjà utilisée lors de la présentation de l'interpolation polynomiale. Le but était de créer une mosaïque de graphiques. Ceci est réalisé de façon très simple grâce à la commande **subplot(n,m,position).** Celle-ci créée une $n \times m$ matrice dont les cellules sont numérotées de 1 à $n \times m$ depuis le coin supérieur gauche et dans le sens habituel de la lecture. Le paramètre nommé `position` sélectionne la cellule à laquelle vont s'appliquer tous les prochains ordres graphiques comme s'il s'agissait d'un graphique isolé.

La commande **subplot(1,1,1)** permet de revenir à une configuration à dessin unique.

7.2 Les tracés en 2 dimensions

7.2.1 La commande `plot`

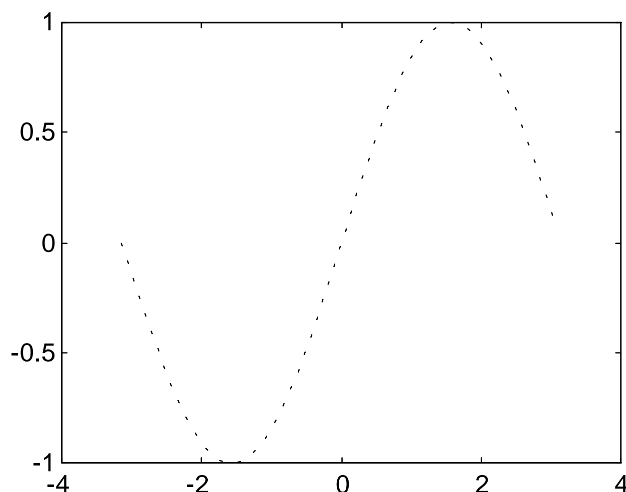
La commande plot est à la base de tout graphique en deux dimensions. La syntaxe est la suivante :

plot(abscisses, ordonnees [, format]{, abscisses, ordonnees [,format]}n)

Par exemple, la suite des commandes :

```
>> x=[-pi:0.1:pi];
>> y=sin(x);
>> plot(x,y,'w:');
>>
```

donne le résultat graphique suivant :



Il n'est pas possible d'afficher des données présentées sous forme symbolique : il faut toujours fournir des données tabulées sous la forme de deux séries : les abscisses et les ordonnées correspondantes.

7.2.2 Les commandes d'affichage 2D discrètes

En plus de la commande plot qui permet d'afficher un tracé continu ou discret de données présentées sous la forme d'un tableau, Matlab propose toute une panoplie de commandes destinées à présenter les données discrètes sous des formats spécialisés.

Les commandes **stem**, **bar** et **stairs** permettent respectivement de créer des diagrammes à *bâtons*, en *barres* (similaires à un histogramme) et en *escalier*.

Les paramètres sont similaires :

{stem | bar | stairs}(y) crée un diagramme régulièrement espacé des données de **y**.

{stem | bar | stairs}(x,y) crée un diagramme où les **x** déterminent la position en abscisse du motif et **y** son ordonnée.

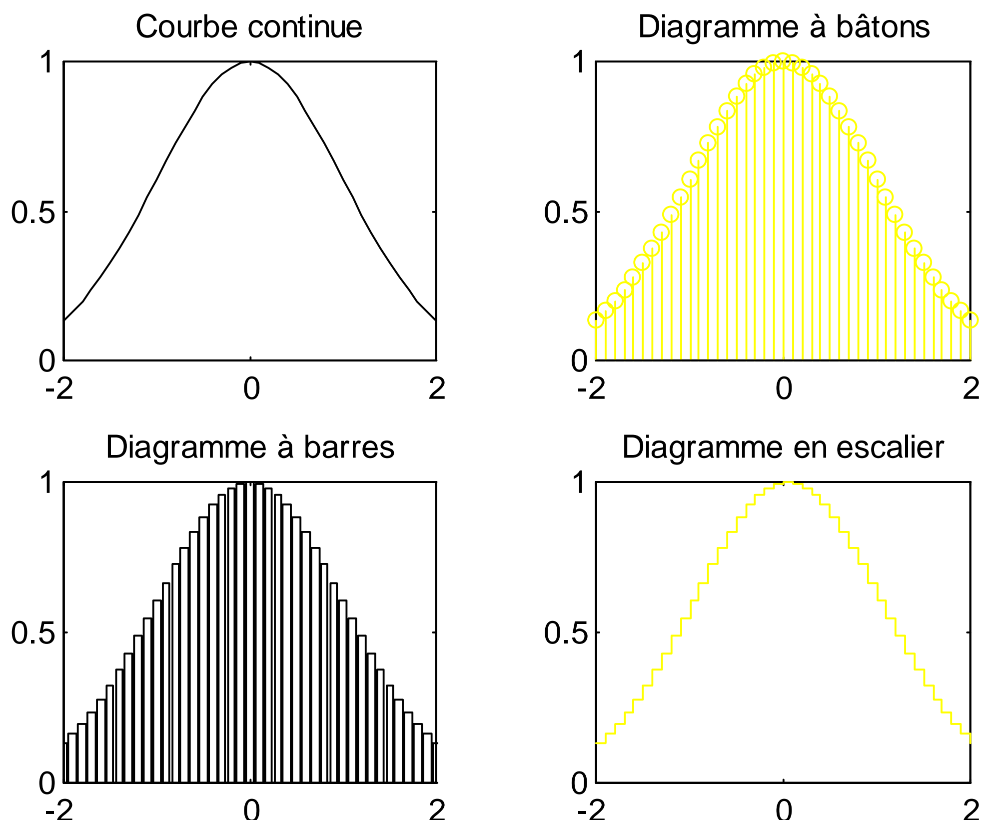
Il est possible de passer à ces commandes un argument de format sous la forme habituelle d'une chaîne de caractères.

En outre, pour les commandes **bar** et **stairs**, il est possible de demander des résultats de sortie sous la forme **[xx,yy]={bar, stairs}(x,y)**. Auquel cas, aucun graphe n'est tracé, mais ce dernier pourra être obtenu par la commande **plot(xx,yy).** Cette possibilité peut être utilisée pour afficher conjointement une courbe et un diagramme avec une mise en page spéciale.

Notez également que **bar** et **stairs** nécessitent que les abscisses x soient fournies en ordre croissant.

La figure suivante montre l'utilisation de ces différentes techniques sur un graphique contenant 4 sous graphiques, le code Matlab est le suivant :

```
x=[-2:0.1:2];
y=exp((-x.^2)/2);
subplot(2,2,1), plot(x,y), title('Courbe continue');
subplot(2,2,2), stem(x,y), title('Diagramme à bâtons');
subplot(2,2,3), bar(x,y), title('Diagramme à barres');
subplot(2,2,4), stairs(x,y), title('Diagramme en escalier');
subplot(1,1,1);
```



La commande **hist** travaille différemment car elle crée des classes dans les valeurs dans qui lui sont fournies puis renvoie un graphique montrant la cardinalité de chaque classe.

Sous la forme hist(y), la fonction crée exactement 10 classes de même taille. Il est possible de spécifier manuellement que l'on souhaite utiliser n classes par la commande hist(y,n). Finalement, l'utilisateur peut choisir lui même ses classes. Dans ce cas, elles sont passées en second argument : hist(y,x).

Le code suivant crée des notes réparties sur une gaussienne centrée en 14 et d'écart type 5. Les notes sont arrondies à l'entier immédiatement supérieur grâce à la fonction **ceil**. Dans un second temps, on seuille toutes les notes inférieures à 0 et celles supérieures à 20. On souhaite étudier graphiquement la répartition des notes, pour cela, on spécifie que chaque classe correspond à une note. Notez que la spécification manuelle des classes impose que celles ci correspondent exactement à un entier.

Génération aléatoire des notes

```
notes=ceil(randn(1,100)*5+12);
```

Seuillage des notes ≤ à 0

```
indices=find(notes < 0);
notes(indices)=zeros(length(indices));
```

Seuillage des notes > à 20

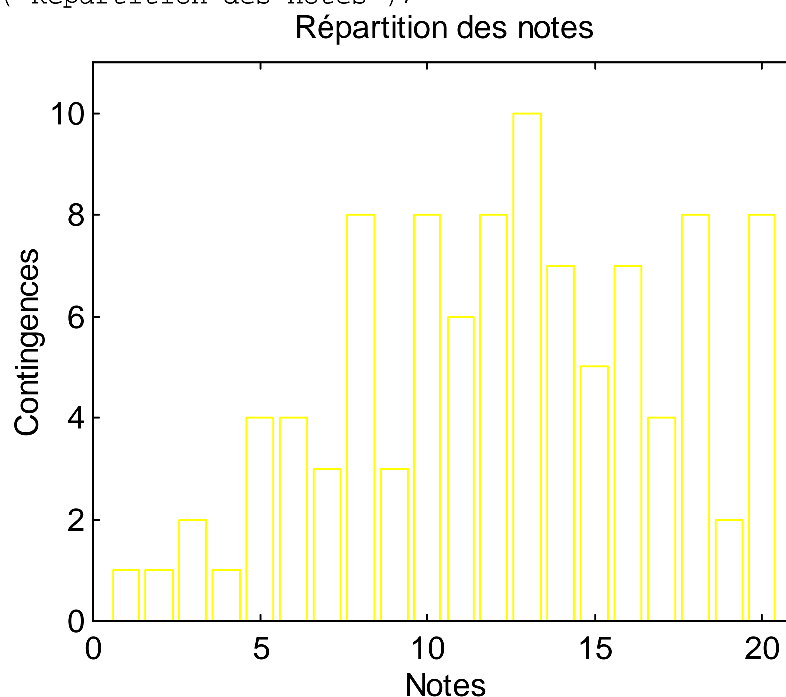
```
indices=find(notes > 20);
notes(indices)=ones(size(indices)).*20;
```

Génération des classes

```
classes=(0:1:20);
hist(notes,classes);
```

Elargissement des axes et addition d'enrichissements au graphique

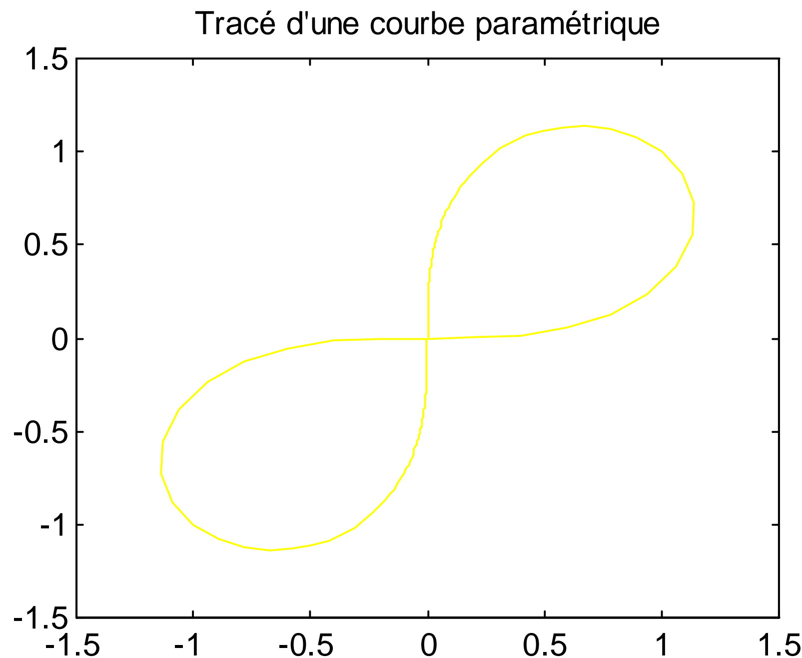
```
axis([min(notes)-1 max(notes)+1 0 length(find(notes==13))+1]);
xlabel('Notes');
ylabel('Contingences');
title('Répartition des notes');
```



7.2.3 Tracé de courbes paramétriques

Le principe revient à exprimer les abscisses et les ordonnées en fonction du paramètre puis à les afficher avec la commande plot. Par exemple, visualisons la courbe $\begin{cases} x = \dfrac{2t}{1+t^4} \\ y = \dfrac{2t^3}{1+t^4} \end{cases}$ Le code Matlab peut ressembler à :

```
» t=-1000:0.1:1000;
» x=(2.*t)./(1+t.^4);
» y=(2.*t.^3)./(1+t.^4);
» plot(x,y);
» title('Tracé d''une courbe paramétrique');
```



7.2.4 Courbes polaires

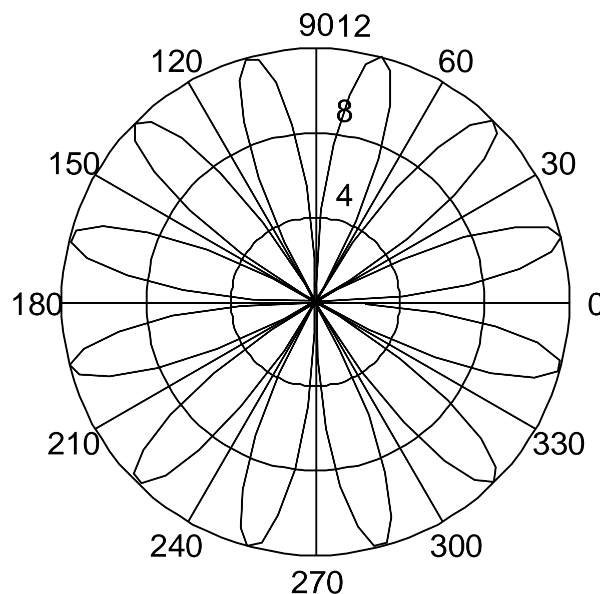
Il est possible d'afficher des courbes polaires à l'aide de la commande **`polar(theta, rho, format)`** où **`format`** fonctionne sur le même principe que celui de **`plot`**.

Par exemple, écrivons une fonction qui affiche une rosace à l'écran. Celle-ci est définie par deux paramètres, la longueur des limbes **`a`**, et le nombre de limbes **`n`**.

```
function rosace(a,n,style)
  if (nargin < 2)
    error('Pas assez de paramètres : rosace(longueur, nombre [,
format])');
  end;
  theta=-pi:0.05:pi;
  rho=a.*sin(n.*theta);
  if (nargin < 3)
    polar(theta,rho);
  else
    polar(theta,rho,style);
  end;
end;
```

Notez l'utilisation de **`nargin`** pour vérifier la présence des deux paramètres obligatoires et du troisième paramètre facultatif : le format d'affichage. Sur la figure résultant de **`rosace(12,6)`** notez les deux types de graduation :

- Les angles theta en degrés
- Les amplitudes rho en unités arbitraires



7.2.5 Echelles logarithmiques

Nous terminons cette brève introduction au graphisme en 2D de Matlab par l'étude des fonctions permettant de traiter des données avec des échelles logarithmiques. Ces fonctions sont similaires à **`plot`** : les arguments sont rigoureusement identiques.

`semilogx` trace le graphique en utilisant une échelle logarithmique sur l'axe des x et une échelle linéaire sur l'axe des y

`semilogy` trace le graphique en utilisant une échelle linéaire sur l'axe des x et une échelle logarithmique sur l'axe des y

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

`loglog` applique une transformation logarithmique sur les deux axes

7.3 Les tracés en 3 dimensions

Avouez que nous étiez nombreux à l'attendre cette section ! Et bien, vous serez mal récompensés de vos efforts car, comme je suis miro 15ème dan je suis bien mal placé pour vous placer du graphisme 3D … Si toutefois, vous êtes directement arrivés ici sans passer par la case 2D, je vous conseille d'y jeter un petit œil.

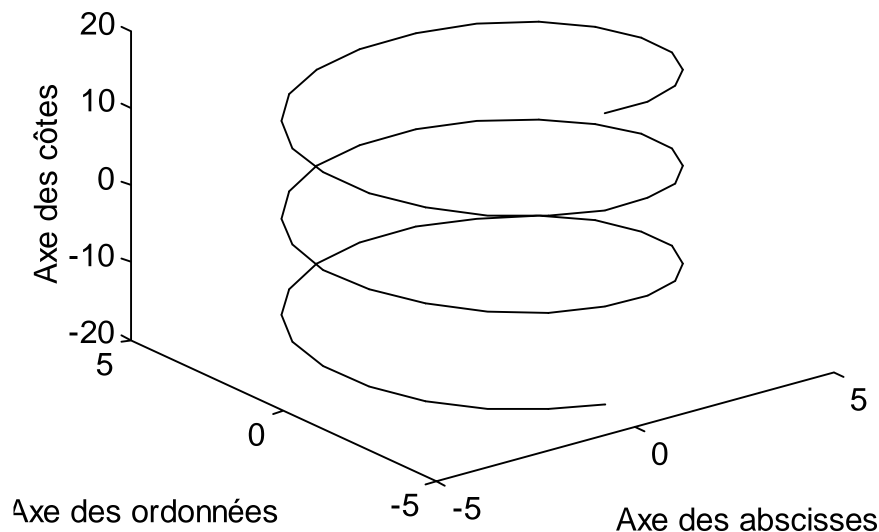
7.3.1 La commande plot3

Plot3 est au graphisme 3D ce que plot était au graphisme 2D : la fonction de base permettant d'afficher des courbes.

Etudions son comportement sur une fonction paramétrique :

```
» t=-3*pi:pi/10:3*pi;
» x=4*sin(t);
» y=4*cos(t);
» z=2*t;
» plot3(x,y,z), xlabel('Axe des abscisses'), …
  ylabel('Axe des ordonnées'), zlabel('Axe des côtes');
```

La distinction entre abscisses et ordonnées ne m'ayant jamais parue claire dans ce genre de graphiques, j'ai préféré les étiqueter



7.3.2 L'affichage de surfaces

L'affichage de surfaces passe par la réalisation d'un maillage du plan xy puis le calcul de la surface pour chacun des points le constituant.

La constitution du maillage passe par l'utilisation de la fonction **`meshgrid`**. Donnons d'abord un exemple de son utilisation :

```
» x=[-2:2];
» y=[3:4];
» [X,Y]=meshgrid(x,y)

X =

    -2    -1     0     1     2
    -2    -1     0     1     2


Y =

     3     3     3     3     3
     4     4     4     4     4
```

Voici quelques explications :

La fonction **`meshgrid`** a créé deux matrices **`X`** et **`Y`** de même taille et représentant le maillage. Chaque ligne de **`X`** correspond au vecteur **`x`** et **`X`** comporte **`length(y)`** lignes ; réciproquement, chaque colonne de **`Y`** correspond au vecteur **`y`** et **`Y`** comporte **`length(x)`** colonnes.

Une fois établies ces matrices de maillage, on peut calculer la valeur de la côte en chaque point. Par exemple, pour calculer une surface de parabole sur un carré de 4 unités de côté :

```
» x=[-2:0.05:2];
» y=[-2:0.05:2];
» [X,Y]=meshgrid(x,y);
» Z=X.^2 + Y.^2;
» mesh(X,Y,Z);
```

La famille des fonctions **`mesh`** trace un graphique en fils de fer. Elles prennent toutes 3 paramètres, dans l'ordre : les matrices de maillage **`X`** et **`Y`** ainsi que la matrice des côtes sur le maillage **`Z`**. Les autres possibilités de tracé sont les suivantes :

`meshz` ajoute des lignes de projection sur les côtes

`contour` trace les contours projetés de **`Z`** sur le plan xy

`meshc` combine **`mesh`** et **`contour`** sur le même graphique

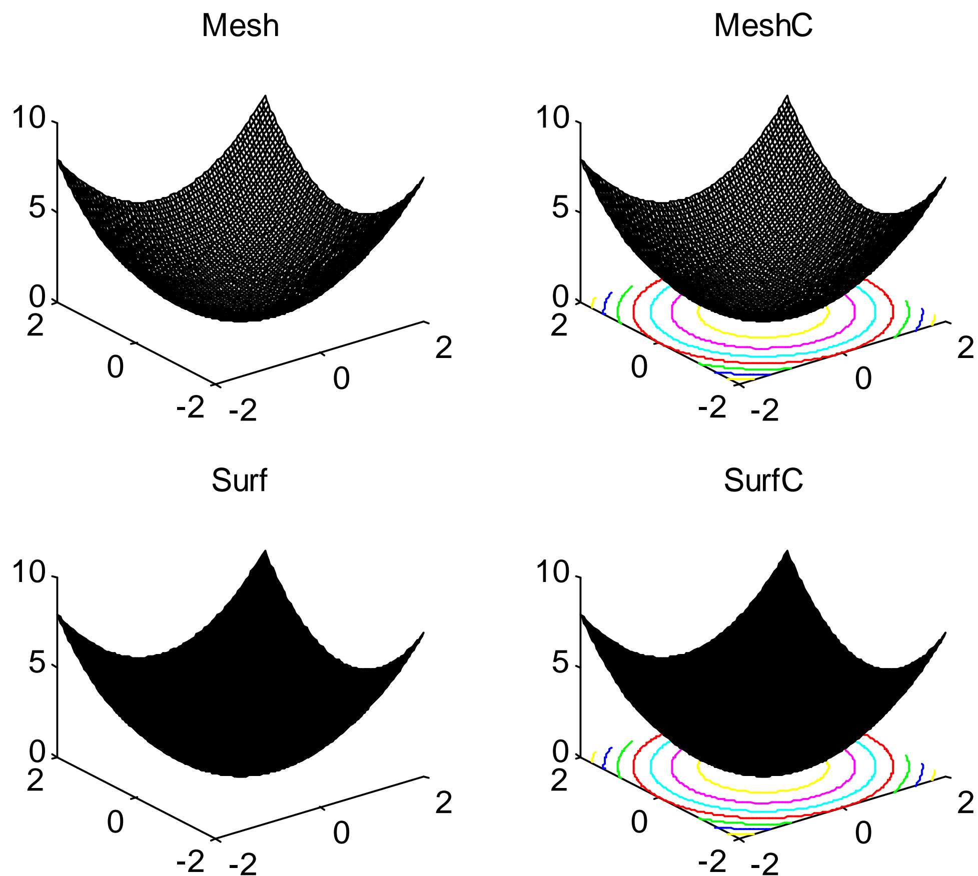
`surf` colore les espaces entre les fils

`surfc` combine **`surf`** et **`contour`**

`surfl` ajoute une source de lumière virtuelle.

Il existe bien d'autres options, permettant, par exemple, de régler la palette de couleurs ou bien le point de vue par rapport auquel on regarde la courbe. L'étude de toutes ses possibilités, bien au delà du contexte de cette introduction, est laissée au soin du lecteur.

La figure suivante illustre néanmoins certaines des fonctions nommées ci-dessus.



8. Calcul de performance

Il est possible de calculer les performances d'un calcul aussi bien en nombre d'opérations flottantes (les fameux flops) qu'en temps de calcul. Le premier élément est souvent plus intéressant car il sera le même quelle que soit la charge de la machine.

8.1 Syntaxe

La commande permettant d'évaluer le nombre d'opérations est intitulée **`flops`** et renvoie le nombre d'opérations en virgule flottante effectués depuis la dernière remise à zéro du compteur, liée à la commande **`flops(0).`**

Le temps de calcul se mesure grâce aux fonctions **`tic`** et **`toc.`** **`Tic`** lance un chronomètre, **`toc`** l'arrête et renvoie le temps écoulé dans une unité qui dépend de la plate forme matérielle.

8.2 Un exemple d'utilisation

Ce petit bout de code montre l'utilisation des commandes de mesure de performance et de **`nargout`**. Si un seul résultat est demandé, cette fonction renvoie la moyenne du tableau passé en paramètre. Les deuxième et troisième résultats correspondent respectivement au nombre d'opérations et au temps de calcul.

```
function [moy,ops,temps]=moyenne(v)

  if ((nargin == 0) | (nargin > 1) | (nargout > 3))
    error('Attention aux arguments : [moy,ops,temps]=moyenne(v)');
  end

  if (nargout > 1)
    flops(0);
    if (nargout > 2)
      tic;
    end;
  end;

  moy=mean(v);

  if (nargout > 1)
    ops=flops;
    if (nargout > 2)
      temps=toc;
    end;
  end;
end;
```

9. La toolbox optimisation

La librairie d'optimisation permet d'effectuer un certain nombre d'opérations d'optimisation. Nous n'étudierons pas ici l'intégralité de ces nombreuses fonctionnalités

9.1 La programmation linéaire

La fonction LP permet de calculer la solution d'un programme linéaire de taille relativement modeste.

Sa syntaxe la plus générale, est la suivante :

```
[x, lambda]=LP(c,A,b,inf,sup,init,N,-1)
```

Avec :

`x`	Vecteur solution
`lambda`	Vecteur des multiplicateurs de Lagrange à l'optimum. Permet de savoir quelles contraintes sont actives
`c`	Vecteur objectif, on cherche à minimiser la quantité `< x c>`
`A`	Matrice des contraintes
`b`	Second membre
`inf`	Vecteur des bornes inférieures sur les variables, peut être défini sur les |`inf`| premières variables seulement
`sup`	Vecteur des bornes supérieures sur les variables, peut être définir sur les |`sup`| premières variables seulement
`init`	Solution initiale à partir de laquelle on souhaite itérer
`N`	Permet de spécifier, si nécessaire, que les `n` premières contraintes sont de type égalité
`-1`	Si vous passez –1 en dernier paramètre, les messages d'avertissement de Matlab seront passés sous silence.

Tableau 9.1 Arguments de la fonction `lp`

Seuls les paramètres c, A et b sont réellement nécessaires.

Exemple :

Soit un problème à 4 variables et 6 contraintes

9.2 La programmation quadratique

Matlab permet de résoudre des problèmes quadratiques sous un formalisme très voisin de celui de la programmation linéaire. En effet, voici la syntaxe la plus générale :

```
[x, lambda]=LP(H,c,A,b,inf,sup,init,N,-1)
```

La seule différence concerne la fonction à minimiser qui est ici $\langle Hx, x \rangle + \langle cx \rangle$

9.3 Le vecteur des options

A partir de maintenant, toutes les procédures d'optimisation utilisent un vecteur d'options. La taille de ce vecteur est de 18 réels. La plupart des cases sont en entrée, mais certaines sont en sortie.

C'est un vecteur très général et seules certaines de ces cases sont utilisées par l'ensemble dans algorithmes. Le tableau suivant récapitule les informations rassemblées dans ce vecteur (la première colonne indique l'indice de l'options, la seconde s'il s'agit d'un paramètre en entrée ou en sortie, la troisième donne une brève description de l'option):

1	E	Détermine le niveau de renseignements fournis par l'algorithme : 0 muet 1 bavard
2, 3, 4	E	Tolérance sur le critère de terminaison. Avant de terminer un algorithme d'optimisation, trois critères sont vérifiés : (2) différences relatives sur les valeurs consécutives des variables (3) différence relative sur les valeurs consécutives de l'objectif (4) violation des contraintes Vous avez ici la possibilité de fixer ces tolérances. Les valeurs par défaut sont les suivantes : (2) 10^{-4} (3) 10^{-4} (4) 10^{-6}
5	E	Stratégie d'optimisation : utilisée pour l'optimisation multi critère
6	E	Méthode d'optimisation sans contrainte : 0 Quasi Newton type BFGS (par défaut) 1 Gradient conjugué type DFP 2 Méthode du gradient (à proscrire)
7	E	Stratégie de recherche linéaire

		0 Stratégie mixte cubic fit/quadratic fit (par défaut et à conserver !) 1 Cubic fit seul
8	S	Valeur de la fonction objectif à l'optimum
9	E	Doit être fixée à 1 si l'utilisateur fournit des fonctions pour calculer le gradient, et à 0 sinon (calcul du gradient par différences finies)
10	S	Nombre d'évaluations de la fonction objectif
11	S	Nombre d'évaluations du gradient
12	S	Nombre d'évaluations de contraintes
13	E	Nombre de contraintes de type égalité pour la procédure constr
14	E	Nombre maximal d'itérations, fixé par défaut à 100 fois le nombre de variables
15	E	Utilisé par l'optimisation multi critère
16, 17	E	Utilisés dans le cas du calcul du gradient par différences finies : respectivement égal aux changements minimal et maximal de valeur des variables lors du calcul du gradient en différences finies
18	E	Pas d'optimisation. Ne pas changer sous peine de catastrophe !

Tableau 9.2 Liste des options des fonctions d'optimisation

9.4 L'optimisation sans contrainte

9.4.1 Dimension 1

Le problème à résoudre est ici min f(x) sur l'intervalle [a, b] où f est une fonction numérique (le résultat est un scalaire) à une seule variable.

La fonction utilisée est **fmin**. Cette fonction n'appartient pas au paquetage optimisation mais au noyau central de Matlab. Sa syntaxe générale est très simple :

```
[Z,OPTIONS]=fmin('Fonction', a, b, OPTIONS, p1,p2,…,p10)
```

Avec :

'Fonction'	Chaîne de caractères indiquant l'expression de la fonction à minimiser. Utiliser une chaîne de caractères permet d'éviter la création d'un fichier .m. Il est toutefois possible de donner le nom d'un fichier .m

`a,b`	Bornes de l'intervalle de minimisation Ici se terminent les arguments obligatoires
`OPTIONS`	Vecteur d'options tel que présenté à la section précédente. Seules les options 1, 2 et 14 sont utilisées. Si Options n'est pas présent, les valeurs par défaut sont utilisées : Algorithme muet Tolérance sur les variables de 10^{-4} 500 itérations au maximum Il est possible de récupérer les OPTIONS en sortie pour connaître la valeur de l'objectif, le nombre d'appels réalisés, etc. (voir le tableau des options pour plus d'éclaircissements)
`p1 à p10`	Arguments passés à la fonction à minimiser. C'est utile si la fonction prend plusieurs arguments mais la minimisation ne porte que sur le premier. Par exemple si la fonction s'écrit y=f(x,z) alors, la minimisation portera sur x et vous pourrez passez une constante pour z dans p1

Tableau 9.3 liste des arguments de la fonction fmin

Seules les paramètres 'Fonction', a et b sont nécessaires.

9.4.2 Dimension n

Il s'agit ici d'appliquer des algorithmes classiques comme le gradient conjugué ou les méthodes de quasi newton.

Tout ceci est effectué via la fonction `fminu` dont la syntaxe est :

```
[x,options]=fminu('Fonction',initial,options,'gradient')
```

Avec :

`'Fonction'`	Chaîne donnant l'expression de la fonction à minimiser. Il est également possible de donner le nom d'un fichier .m
`initial`	Valeur de départ pour le vecteur solution
`options`	Vecteur des options, les options « utiles » sont les suivantes : 1,2,3,14 ainsi que 6 qui permet de choisir l'algorithme d'optimisation et 7 qui spécifie l'algorithme de recherche linéaire. Voir le tableau spécifique aux options pour connaître les valeurs par défaut.

gradient	Chaîne de vecteur indiquant la dérivée partielle de `Fonction` par rapport à chaque variable. Il est également possible de passer un fichier .m

Tableau 9.4 Liste des options de la fonction `fminu`

Seuls les arguments `'Fonction'` et `Initial` sont obligatoires. Si `Gradient` est omis, alors les dérivées sont calculées par différences finies.

9.4.3 Optimisation simplex

Dans certains cas, l'information sur les dérivées, même du premier ordre, n'est pas disponible. Il est alors possible de recourir à l'optimisation « simplex » de Brent (`fmins`) basée uniquement sur des considérations géométriques. Les arguments sont similaires à ceux de `fmin` :

```
fmins('f',x0,options,[],p1,p2,...)
```

… il est toutefois nécessaire de fournir une solution initiale (`x0`), et un vecteur vide juste après les options !

9.4.4 Exemple

Nous allons utiliser `fminu` pour calculer le minimum d'une fonction tristement célèbre : la banane de Rosenbrock dont la valeur est :

$$banane(x,y) = 100\,(y - x^2) + (1-x)^2$$

Le gradient est :

$$\begin{cases} \dfrac{\partial banane}{\partial x} = 400(x^3 - xy) + 2x - 2 \\ \dfrac{\partial banane}{\partial y} = 200(y - x^2) \end{cases}$$

Si nous décidons d'écrire ces fonctions dans des fichiers, nous obtenons :

```
%fichier banane.m : la fonction !
function z=banane(x)
  z = 100.0 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
end;
```

Le plus déroutant est ici de repasser du formalisme vecteur à celui des scalaires ! En effet, il faut se souvenir que x et y sont passés sous forme d'un vecteur à 2 éléments.

```
%fichier gradban.m : le gradient !
function z=gradban(x)
  z = [ 400*(x(1)^3-x(2)*x(1))+2*x(1)-2;…
        200*(x(2)-x(1)^2)];
end;
```

L'on ressort le gradient sous la forme d'un vecteur à deux composantes : les dérivées partielles respectivement à x et y.

Il est alors possible d'obtenir la minimisation par l'appel :

```
[x,options]=fminu('banane',[-3 2],[],'gradban')
```

Vous obtenez alors :

- Dans x le résultat de l'optimisation
- Dans options(8) la valeur de banane à l'optimum
- Dans options (10) le nombre d'évaluations de la fonction
- Dans options (11) le nombre d'évaluations du gradient

Eussiez vous voulu utiliser le formalisme par chaînes de caractères plutôt que des fichiers .m que vous auriez pu taper :

```
Banane=' 100.0 * (x(2) - x(1)^2)^2 + (1 - x(1))^2'

GradBanane='[400*(x(1)^3-x(2)*x(1))+2*x(1)-2 ;200*(x(2)-x(1)^2)]'

[x,options]=fminu(Banane,[-3 2],[],GradBanane)
```

9.5 Optimisation avec contraintes

Il faut désormais non seulement passer la fonction à minimiser mais également les contraintes !

La syntaxe générale est la suivante :

```
constr('Fonction + contraintes', initial, options, inf, sup, 'gradient').
```

Avec :

`'F+C'`	Chaîne indiquant l'expression de la fonction et des contraintes ou bien un fichier les calculant
`initial`	Valeur initiale de x permettant de démarrer l'algorithme
`options`	Options d'optimisation. Si 'F+C' contient des contraintes égalités, ce sont les options(13) premières contraintes.
`inf et sup`	Vecteurs indiquant respectivement les bornes inférieures et supérieures des variables. Ces vecteurs peuvent avoir des dimensions inférieures à celles de x et même être vides
`gradient`	Chaîne indiquant l'expression du gradient de la fonction et le jacobien des contraintes ou bien un fichier permettant de les calculer

Tableau 9.5 Liste des arguments de la fonction `constr`

A mon avis, à partir du moment où l'on travaille avec des contraintes, utiliser des chaînes de caractères directement devient trop pénible et il faut s'en remettre aux fichiers. En outre, cela permet de présenter de bien meilleur façon les problèmes.

Supposons que l'on veuille minimiser la fonction :

$$e^x(4x^2+2y^2+4xy+2y+1)$$

sous les contraintes :

$$\begin{cases} c_1 \ : \ xy-x-y \leq -\dfrac{3}{2} \\ c_2 \ : \ y-x \leq 10 \end{cases}$$

Il est possible de traiter ce problème grâce aux fichiers consf.m et consg.m fournissant respectivement :

- La fonction objectif et les contraintes sous la forme d'une fonction qui renvoie 2 arguments :

```
% fichier consf.m
function [f,g]=consf(x)
  f=exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
  g=[ 1.5+x(1)*x(2)-x(1)-x(2), -x(1)*x(2)-10];
end
```

  La valeur de la fonction est retournée sous forme d'un scalaire dans le premier argument de retour

  Les contraintes sont positionnées sur la même ligne d'une matrice

- Le gradient de la fonction objectif et la matrice jacobienne des contraintes sous la forme suivante :

```
% fichier consg.m
function [gf,gg]=consg(x)
t= exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
gf=[t+4*exp(x(1))*(2*x(1)+x(2)) ; 4*exp(x(1))*(x(1)+x(2)+0.5)];
gg = [ x(2)-1,-x(2);x(1)-1,-x(1)];
```

  Le premier terme de retour correspond au gradient de la fonction objectif exprimé sous forme d'un vecteur colonne (les point virgules séparant les différentes lignes sont clairement visibles dans l'exemple ci-dessus). En règle générale, et si la fonction f est à n variables, ce vecteur est sous la forme :

$$\left[\frac{\partial f}{\partial x_1};\frac{\partial f}{\partial x_2};\cdots;\frac{\partial f}{\partial x_n}\right]$$

  Le second argument est associé à la matrice jacobienne des contraintes.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Une ligne est associée à une variable, une colonne à une contrainte !, en règle générale, si nous avons m contraintes pour n variables, la matrice est sous la forme :

$$\begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_2}{\partial x_1} & \cdots & \frac{\partial g_m}{\partial x_1}; \\ \frac{\partial g_1}{\partial x_2} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_m}{\partial x_2}; \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial x_n} & \frac{\partial g_2}{\partial x_n} & \cdots & \frac{\partial g_m}{\partial x_n} \end{bmatrix}$$

Vous pouvez voir ici un avantage de l'utilisation des fichiers : il est possible d'effectuer des calculs intermédiaires !

10. Conclusion

Au travers de cette (courte) introduction, j'espère vous avoir montré l'intérêt de Matlab, outil qui m'a considérablement aidé au cours de mon travail de recherche. Les aspects que j'ai présentés ne sont toutefois que les plus basiques, la richesse de Matlab rendant toute synthèse de moins de 300 pages impossible. Les personnes intéressées sont donc invitées à poursuivre leur étude par la lecture d'ouvrages plus complets, et surtout l'expérimentation personnelle.

Ce poly est dédié aux étudiants de la première promotion de F4 qui ont eu à subir ce cours et m'ont considérablement aidé à l'améliorer. J'ai nommé (sans aucune intention de classement) : Caroline, Hanane, Julie, Julien A., Julien P., Rocco, Omer, Junior, Tchoum, Sony, Christian, Stéphane, Nitro, Loïc, Yannick, Marco, Pookie, Guillaume, Buz, Antoine, Dany, FX et Philippe avec une mention toute particulière pour le ranger numénoréen qui se reconnaîtra.

F4 For Ever