TABLE DES MATIÈRES

A l'origine MATLAB[1] a été conçu pour le calcul matriciel, l'algèbre linéaire et l'analyse numérique. Il s'agissait de mettre à la disposition des utilisateurs un logiciel simple mais avec des possibilités suffisamment variées afin de ne pas devoir écrire des programmes en Fortran. Dans sa version actuelle[2], MATLAB comporte des fonctionnalités qui vont bien au-delà de ce qu'a été le Matrix Laboratory. Il s'agit aujourd'hui d'un logiciel de haute performance pour le calcul numérique et la visualisation. On trouve MATLAB pratiquement sur toutes les plateformes de calcul.

[1] MATrix LABoratory était originalement écrit en Fortran par Cleve Moler au début des années 1980. En 1984, il fonda avec John Little l'entreprise The MathWorks Inc. qui, aujourd'hui, commercialise MATLAB. Pour plus d'informations on peut consulter `http://www.mathworks.com` ou écrire à l'adresse électronique `info@mathworks.com`.

[2] Cette introduction porte sur la version 4.2 de MATLAB.

1

PREMIÈRE APPROCHE

MATLAB (version 4.2) travaille essentiellement avec un seul type d'objet qui est constitué par une matrice numérique. Le cas échéant la matrice peut contenir des éléments complexes. Les scalaires et les vecteurs constituent évidemment des cas particuliers.

La syntaxe des opérations et commandes de MATLAB est naturelle, dans le sens où elle correspond, la plupart du temps, à ce que l'on écrirait sur un papier.

Dans un environnement DOS, MATLAB s'active avec la commande `matlab`, dans un environnement Windows, en cliquant sur l'icône corrrespondante. On peut terminer une séance de travail avec la commande:

```
>> quit
```

1.1 INTRODUIRE DES MATRICES

Pour introduire une matrice il n'est nécessaire de déclarer ni la dimension ni le type de variable. La mémoire est allouée dynamiquement dans les limites de la machine.

La façon la plus simple d'entrer des matrices de taille modeste consiste à donner la liste des éléments qui constituent les lignes. Les lignes sont séparées, soit par des points-virgules, soit par Return. Par exemple, en tapant les commandes:

```
A = [1 2 3; 4 5 6; 7 8 0]
```

on obtient la réponse

```
A =
     1     2     3
     4     5     6
     7     8     0
```

et la matrice A reste définie dans la mémoire du programme.

Il est également possible de générer des matrices particulières à l'aide de fonctions (`rand`, `zeros`, `ones`), de lire des matrices contenues dans des fichiers ASCII ayant l'extension `".m"` ou de les charger à partir de fichiers de données externes (`load`).

1.2 SYMBOLES ; , ...

Les éléments d'une liste sont séparés, soit par un ou plusieurs blancs, soit par une virgule. Par exemple:

```
b = [1,2     3 4,5]
```

Une virgule sert également à séparer plusieurs commandes sur une même ligne. Par exemple:

```
a = [1 2 3], b = [4 5 6]
```

Le symbole ; agit également comme séparateur de commandes, mais il supprime aussi l'affichage du résultat.

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

Pour pouvoir étendre une même commande sur plusieurs lignes on utilise le symbole ... comme suit:

```
s = 1 - 1/2 + 1/3 -1/4 ...
+ 1/5 - 1/6;
```

1.3 ELÉMENTS D'UNE MATRICE

Un élément d'une matrice peut être constitué de n'importe quelle expression MATLAB. Si l'on écrit par exemple

```
x = [1 2+5 sqrt(7)]
```

on obtient

```
x =
    1.0000    7.0000    2.6458
```

Des éléments particuliers d'une matrice sont adressés à l'aide d'un indice entre parenthèses. En utilisant le vecteur `x` défini ci-dessus on peut écrire

```
x(5) = log(x(2))
```

ce qui produit

```
x =
    1.0000    7.0000    2.6458         0    1.9459
```

On remarque que la dimension du vecteur a été automatiquement ajustée et que les éléments non définis sont initialisés à zéro.

Un élément de matrice peut aussi être constitué d'une matrice, comme c'est le cas dans l'exemple suivant:

```
p = [10 11 12];
B = [A; p]
```

qui produit le résultat

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

1.4 INSTRUCTIONS ET VARIABLES

MATLAB est un langage du type "*expression language*". Une expression tapée par l'utilisateur est d'abord interprétée puis évaluée par MATLAB. En général, une commande MATLAB est de la forme

variable = *expression*

ou encore

expression

Une expression est composée d'opérateurs, de caractères spéciaux, de fonctions et de noms de variables. L'évaluation de l'expression produit une matrice qui est assignée à la variable pour qu'elle puisse être utilisée ulterieurement. Si le nom de variable et le signe `=` sont omis, MATLAB crée automatiquement une variable de nom `ans`. Par exemple

```
1/4
```

produit

```
ans =
      0.2500
```

Les noms des variables et fonctions sont constitués d'une lettre qui peut être suivie d'un nombre quelconque de lettres, nombres ou caractères _. Seuls les 19 premiers caractères d'un nom sont reconnus. Exemple de nom:

```
v12AB_cde_fG123_6789bcd = exp(pi)
```

MATLAB distingue entre caractères minuscules et majuscules (case sensitive) `a` $\neq$ `A`. Les noms des fonctions sont toujours écrits en minuscules.

```
inv(A)
INV(A)    ( pas reconu)
```

1.5 INFORMATIONS SUR L'ESPACE TRAVAIL

Toutes les variables créées dans les exemples précédents ont été conservées dans la mémoire de MATLAB. La commande

```
who
```

produit la liste des variables définies dans la mémoire:

```
Your variables are:
A                          p
a                          s
ans                        v12AB_cde_fG123_678
b                          x
```

On reconnaît les variables créées auparavant ainsi que la variable `ans`. Une information plus détaillée, qui renseigne aussi sur la taille des variables, est obtenue avec `whos`. Dans notre cas, on aura:

```
                  Name   Size  Elements  Bytes  Density  Complex
                     A  3 by 3         9     72     Full     No
                     a  1 by 3         3     24     Full     No
                   ans  3 by 3         9     72     Full     No
                     b  1 by 3         3     24     Full     No
                     p  1 by 3         3     24     Full     No
                     s  1 by 1         1      8     Full     No
v12AB_cde_fG123_678  1 by 1         1      8     Full     No
```

```
                x  1 by 5          5      40      Full      No
Grand total is 34 elements using 272 bytes
```

Chaque élément nécessite 8 octets de mémoire. La taille des variables que l'on peut traiter avec MATLAB dépend des caractéristiques physiques de l'ordinateur.

Il existe plusieurs variables définies par MATLAB, mais non affichées avec la commande `whos`. Ces variables dites *permanentes* sont:

`eps` précision machine ($2^{-52} \approx 2.2 \times 10^{-16}$) tolérance utilisée pour certains calculs

`inf` résultat de l'opération 1/0

`NaN` (Not a Number) résultat de inf/inf où 0/0

1.6 NOMBRES ET EXPRESSIONS ARITHMÉTIQUES

On construit des expressions en utilisant les opérateurs arithmétiques et les règles de précédence usuelles.

$+$	addition
$-$	soustraction
$*$	multiplication
$/$	division par la droite
$\backslash$	division par la gauche
$\hat{}$	puissance

MATLAB distingue deux opérateurs pour la division qui seront utiles lors des opérations sur les matrices. Les expressions scalaires 1/4 et 4\1 ont la même valeur numérique.

Sont aussi disponibles un grand nombre de fonctions mathématiques élémentaires comme par exemple `log`, `sqrt`, `sin`, `abs`, `exp` etc. Soit l'exemple d'une instruction suivante:

```
y = 1/sqrt(2*pi)*exp(-x(2)^2/2)
```

Certaines de ces fonctions retournent simplement des constantes comme `pi`, ou des valeurs particulières comme `Inf` ou `NaN` qui correspondent à des representations particulières dans le processeur (standard IEEE[1]). L'opération

```
s = 1/0
```

produit le résultat accompagné du message

```
s =
     Inf
Warning:  Divide by zero.
```

La particularité de l'arithmétique IEEE consiste dans le fait que cette opération ne se termine pas en erreur avec abandon de l'exécution. Au contraire le résultat peut être utilisé pour des calculs subséquents comme montré ci-après:

```
x = 1/0; y = x * 0
y =
     NaN
```

1.7 FORMAT DE SORTIE

Tout résultat d'une expression MATLAB est affiché à l'écran (sauf s'il est suivi du caractère ";". La commande `format` permet le contrôle de l'affichage sans pour autant affecter la précision des calculs. (MATLAB effectue tous les calculs en double précision.)

Si tous les éléments d'une matrice sont des entiers le résultat est affiché sans décimales. Par exemple en tapant

```
x = [ 1 2 3 0 ]
```

on obtient

```
x =
     1     2     3     0
```

Si au moins un des éléments est réel

```
x = [ -5/3  6.5432e-6 ]
```

l'affichage se presente suivant le format choisi comme suit:

```
format compact
format short
```

[1] Institute of Electrical and Electronics Engineers. Organisation professionelle avec plus de 100'000 membres qui chapeaute 35 sociétés techniques dont la Computer Society.

```
     -1.6667    0.0000
format short e
     -1.6667e+00   6.5432e-06
format long
     -1.66666666666667   0.00000654320000
format long e
     -1.666666666666667e+00   6.543199999999999e-06
format hex
     bffaaaaaaaaaaaab   3edb71b51ee73e14
format +
     -+
```

Lorsque le plus grand élément d'une matrice est supérieur à 1000 ou le plus petit élément inférieur à .0001 les formats "short" et "long" mettent en évidence un facteur constant. Par exemple

```
x = 1234 * x
x =
    1.0e+006 *
   -2.5379    0.0000
```

1.8 FONCTION HELP

Cette fonction affiche la description complète d'une fonction donnée. Des exemples d'utilisation sont:

`help` *nom_fonction*
`help eig`

`help` *sujèt*
`help elfun`

`lookfor` *mot-clef*
`lookfor eigenvalue`

2

OPÉRATIONS MATRICIELLES

Les opérations matricielles jouent un rôle central dans MATLAB et une caractéristique saillante de MATLAB est la syntaxe très simple et naturelle pour les opérations matricielles. Dans la mesure des contraintes imposées par le clavier cette syntaxe est naturelle, c'est-à-dire qu'elle correspond à l'écriture usuelle.

2.1 TRANSPOSITION

Le caractère ' désigne la transposition d'une matrice. Avec les commandes

```
A = [1 2 3; 4 5 6; 7 8 10]
B = A'
```

l'on obtient

```
A =
     1     2     3
     4     5     6
     7     8    10
B =
     1     4     7
     2     5     8
     3     6    10
```

et

```
x = [2 -1 0]'
```

produit

```
x =
     2
```

```
-1
 0
```

2.2 ADDITION ET SOUSTRACTION

L'addition et la soustraction de matrices sont definis pour des matrices de même dimension ou si un des opérans est un scalaire. Par exemple

```
C = A + B
```

donne

```
C =
      2     6    10
      6    10    14
     10    14    20
```

et

```
y = x - 1
```

donne

```
y =
     1
    -2
    -1
```

2.3 MULTIPLICATION DE MATRICES

Un produit de matrices est noté * et il est défini si le nombre de colonnes de la première matrice est égal au nombre de lignes de la deuxième matrice. On a par exemple le produit scalaire

```
x'*y
ans =
      4
```

ou les produits suivants:

```
x*y'
ans =
      2    -4    -2
     -1     2     1
      0     0     0
y*x'
ans =
```

```
 2    -1     0
-4     2     0
-2     1     0
```

Soit encore le produit d'une matrice par un vecteur

```
b = A*x
b =
     0
     3
     6
```

ou le produit d'un scalaire par un vecteur

```
pi*x
ans =
      6.2832
     -3.1416
           0
```

2.4 "DIVISION" DE MATRICE

Il existe deux symboles, `/` et \ pour la "division de matrice". Si A est une matrice carrée non singulière alors `A\B` $\equiv A^{-1}B$ et `B/A` $\equiv BA^{-1}$ mais le resultat est calculé sans passer par le calcul de l'inverse. En général

`X = A\B`	est la solution de	`A*X = B`
`X = B/A`	est la solution de	`X*A = B`

Comme dans l'exemple plus haut on a défini `b` comme étant `A*x` l'expression

```
x = A\b
```

donne

```
x =
      2.0000
     -1.0000
     -0.0000
```

2.5 PUISSANCE DE MATRICE

Si A est une matrice carrée et p un entier alors

`A^p` $\equiv \underbrace{A * A * \cdots * A}_{p}$

et si $p \in \mathbb{R}$ alors MATLAB utilise la décomposition $A = VDV^{-1}$ et calcule A^p comme $A^p = VD^pV^{-1}$ avec V et D la matrice des vecteurs propres respectivement valeurs propres de A.

```
A^p ≡ V*D^p/V
```

Les matrices V et D peuvent être calculées avec la commande `[V,D] = eig(A)`.

2.6 FONCTIONS MATRICIELLES ÉLÉMENTAIRES

Lorsque l'argument d'une fonction élémentaire est une matrice la fonction s'applique aux éléments de la matrice. Par exemple `exp(A)` ou `sqrt(A)` calcule l'exponentielle respectivement la racine de chaque élément de la matrice A.

D'autres fonctions matricielles élémentaires consistent en:

`poly`	Polynôme caractéristique
`det`	Déterminant
`trace`	Trace
`kron`	Produit de Kronecker

Exemple 2.1 Soit une matrice A et ses valeurs propres λ qui satisfont $Ax = \lambda x$, $x \neq 0$, avec x les vecteurs propres. On rappelle que λ est une valeur propre, ssi la matrice $\lambda I - A$ est singulière, c'est-à-dire si l'equation caractéristique de A verifie $|\lambda I - A| = 0$.

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \quad |\lambda I - A| = \begin{vmatrix} \lambda - 1 & -2 \\ -2 & \lambda - 1 \end{vmatrix} = (\lambda + 1)(\lambda - 3) = \lambda^2 - 2\lambda - 3.$$

Les coefficients de ce polynôme peuvent être obtenus avec la fonction `poly`.

```
p = poly(A)
p =
     1 -2 -3
```

3

OPÉRATIONS ÉLÉMENT PAR ÉLÉMENT

Pour indiquer des opérations élément par élément les symboles usuels `* / \ ^` sont précédés par un point (`.* ./ .\ .^`).

3.1 ADDITION ET SOUSTRACTION

Pour l'addition et la soustraction, les opérations matricielles et les opérations élément par élément sont les mêmes. Ainsi `+` et `-` peuvent être considérés soit comme une opération matricielle, soit comme une opération élément par élément.

3.2 MULTIPLICATION ET DIVISION

Si `A` et `B` ont la même dimension alors `A .* B` produit une matrice dont les éléments correspondent au produit des éléments de `A` et `B`. Par exemple:

```
a = [1 2 3]; b = [4 5 6];
```

alors

```
z = a .* b
z =
     4    10    18
```

Les expressions `A ./ B` et `A .\ B` calculent le quotient des éléments individuels.

```
z = a .\b
```

```
z =
    4.0000    2.5000    2.0000
```

3.3 PUISSANCE

Ci-après, des opérations de puissance élément par élément avec les vecteurs `a` et `b` définis avant.

La base est un vecteur:

```
c = a.^2
c =
     1     4     9
```

La base et l'exposant sont des vecteurs:

```
c = a.^b
c =
     1    32   729
```

La base est un scalaire et l'exposant est un vecteur:

```
c =  2.^a
c =
     2     4     8
```

3.4 OPÉRATEURS RELATIONNELS

Il existe six opérateurs relationnels pour la comparaison de matrices de même dimension:

OPÉRATEURS RELATIONNELS	
$<$	plus petit que
$>$	plus grand que
$<=$	plus petit ou égal
$>=$	plus grand ou égal
$==$	égal
$\sim =$	différent

La comparaison se fait élément par élément et le résultat est une matrice dont les éléments sont soit `1`, soit `0` (matrice logique).

Par exemple, une condition et le résultat correspondant

```
r = (2+2~=4)
r =
     0
A = [1 2; 3 4]; B = [4 2; 3 1]; C = A==B
C =
     0     1
     1     0
```

Les fonctions `find`, `isnan`, `finite` et `isempty` présentées ci-après sont très utiles lorsqu'elles sont combinées avec des opérateurs relationnels.

La fonction `find` produit un vecteur d'indices

```
x = [1 4 3 -2]; i = find(x<2)
i =
     1     4
```

Le vecteur `i` contient les indices des éléments du vecteur `x` pour lesquels la condition est vraie. (Les éléments 1 et 4 du vecteur `x` sont inférieurs à 2.)

La fonction `find` s'applique également aux matrices.

```
A = [1 3 5; 2 4 6]
A =
     1     3     5
     2     4     6
[i,j] = find(A<4); disp([i j])
     1     1
     2     1
     1     2
```

Les fonctions `isnan` et `finite` associent à un vecteur donné un vecteur logique.

```
x = [1 2 NaN 4 NaN];  i = isnan(x)
i =
     0     0     1     0     1
i = finite(x)
i =
     1     1     0     1     0
```

L'exemple suivant montre que `finite` n'est pas simplement la négation de `isnan`.

```
x = [1 2 Inf 3 NaN]; i = isnan(x)
i =
     0     0     0     0     1
i = finite(x)
```

```
i =
     1     1     0     1     0
i = ~isnan(x)
i =
     1     1     1     1     0
```

La fonction `isempty` permet de vérifier si un vecteur (matrice) est vide.

```
x = [5 7 9 12];  isempty(find(x<5))
ans =
     1
```

3.5 OPÉRATEURS LOGIQUES

Il y a trois opérateurs logiques. Ils s'appliquent en général à des matrices `0-1` et agissent élément par élément. Le résultat est une matrice `0-1`.

OPÉRATEURS LOGIQUES	
&	et
|	ou
~	négation

Par opposition aux opérateurs & et | l'opérateur ~, est unitaire.

```
A = [1 0 1; 0 0 1], B = [0 1 1; 1 0 0]
A =
     1     0     1
     0     0     1
B =
     0     1     1
     1     0     0
A & B
ans =
     0     0     1
     0     0     0
~A
ans =
     0     1     0
     1     1     0
~(~A);
```

Les fonctions `any` et `all` sont utiles lorsqu'elles sont utilisées en liaison avec les opérateurs logiques. Pour un vecteur `a`, la fonction `any` renvoie `1` si un élément

au moins du vecteur est différent de zéro (`0` sinon). La fonction `all` renvoie `1` si tous les éléments du vecteur sont différents de zéro (`0` sinon).

```
a = [1 2 0 4]; any(a)
ans =
      1
all(a)
ans =
      0
```

Lorsque l'argument de ces deux fonctions est une matrice, elles agissent sur les vecteurs colonnes de la matrice.

```
A = [1 0 4; 2 0 5; 3 0 0]
A =
     1     0     4
     2     0     5
     3     0     0
any(A)
ans =
       1     0     1      (1 si la colonne contient au moins 1 élément ≠ 0)
all(A)
ans =
       1     0     0      (1 si tous les éléments de la colonne sont ≠ 0)
any(any(A))
ans =
       1      (1 si au moins 1 élément de la matrice ≠ 0)
all(all(A))
ans =
       0      (1 si tous les éléments de la matrice ≠ 0)
```

Ces fonctions sont utiles, par exemple, pour brancher des instructions. Ici, on teste une condition scalaire:

```
if all(all(A>0))
    instructions
end
```

Si tous les éléments de la matrice `A` sont positifs on exécute un jeu d'instructions (e.g. logarithme des éléments).

Ci-après, la liste des fonctions relationnelles et logiques que l'on trouve dans MATLAB:

FONCTIONS RELATIONNELLES ET LOGIQUES	
`any`	condition logique
`all`	condition logique
`find`	recherche d'indices d'éléments d'un vecteur
`exist`	verifie l'existence d'une variable
`isnan`	détection de `NaN`
`finite`	détection de `Inf`
`isempty`	détection de matrices vides
`isstr`	détection de chaînes de caractères
`strcmp`	comparaison de chaînes de caractères

3.6 FONCTIONS MATHÉMATIQUES ÉLÉMENTAIRES

Les fonctions mathématiques élémentaires agissent élément par élément lorsqu'elles sont appliquées à des vecteurs et matrices.

FONCTIONS TRIGONOMÉTRIQUES	
`sin`	sine
`cos`	cosine
`tan`	tangent
`asin`	arcsine
`acos`	arccosine
`atan`	arctangent
`atan2`	four quadrant arctangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`asinh`	hyperbolic arcsine
`acosh`	hyperbolic arccosine
`atanh`	hyperbolic arctangent

Fonctions mathématiques élémentaires	
`abs`	absolute value or complex magnitude
`angle`	phase angle
`sqrt`	square root
`real`	real part
`imag`	imaginary part
`conj`	complex conjugate
`round`	round to nearest integer
`fix`	round toward zero
`floor`	round toward $-\infty$
`ceil`	round toward ∞
`sign`	signum function
`rem`	remainder
`exp`	exponential base e
`log`	natural logarithm
`log10`	log base 10

Fonctions mathématiques particulières	
`bessel`	bessel function
`gamma`	gamma function
`rat`	rational approximation
`erf`	error function
`inverf`	inverse error function

4

VECTEURS ET MATRICES

MATLAB permet d'adresser des éléments ou sous-ensembles d'éléments d'un vecteur ou d'une matrice d'une façon simple et efficace. Ceci constitue une des caractéristiques importantes de MATLAB. L'opérateur : joue un rôle central dans toutes ces manipulations.

4.1 GÉNÉRATION DE VECTEURS

On peut générer un vecteur avec des éléments allant de 1 à 5 avec un incrément de un en utilisant la commande

```
x = 1:5
x =
     1     2     3     4     5
```

Il est aussi possible de spécifier des incréments qui ne sont pas unitaires:

```
y = 0:pi/4:pi
y =
     0    0.7854    1.5708    2.3562    3.1416
n = 9:-1:4
n =
     9     8     7     6     5     4
```

Exemple 4.1 Exemple de construction d'un tableau donnant l'intégrale de 0 à x de la fonction $\frac{2}{\sqrt{\pi}}e^{-t^2}$ pour x allant de 1 à 1.4 avec un incrément de .1:

```
x = (1.0:0.1:1.4)';
F = erf(x);
```

$$F = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t$$

```
disp([x F])
    1.0000    0.8427
    1.1000    0.8802
    1.2000    0.9103
    1.3000    0.9340
    1.4000    0.9523
```

La fonction `erf` est utile pour évaluer des intégrales du type $F(y) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^y e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} dx$. On effectue le changement de variable $x = \sigma t\sqrt{2} + \mu$ avec $dx = \sigma\sqrt{2}\, dt$ et en remplaçant on obtient $\frac{1}{\sqrt{2\pi}} \int_0^z e^{-t^2} dt$, ce qui correspond à $\frac{1}{2}$`erf(z)` avec $z = \frac{y-\mu}{\sigma\sqrt{2}}$.

Les fonctions `linspace` et `logspace` permettent également la construction de vecteurs à éléments équidistants. Dans ce cas, on spécifie le nombre de points et non l'intervalle entre les points.

```
x = linspace(-3,3)       (crée 100 points équidistants par defaut)
x = logspace(1,2,5)
x =
    10.0000   17.7828   31.6228   56.2341  100.0000
log10(x)
ans =
     1.0000    1.2500    1.5000    1.7500    2.0000
```

4.2 INDEXATION DES ÉLÉMENTS D'UNE MATRICE

Un élément particulier d'une matrice est référencé par ses indices entourés d'une parenthèse. Si des expressions sont utilisées à la place des indices, elles sont arrondies à l'entier le plus proche. Soit une matrice `A`:

```
A =
     1     2     3
     4     5     6
     7     8     0
```

alors on peut modifier un élément moyennant la commande

```
A(2,3) = A(1,3) + A(2,1)
A =
     1     2     3
     4     5     7
     7     8     0
```

Un indice peut aussi être constitué par un vecteur. Soit deux vecteurs `y` et `v`; `y(v)` est equivalent à `[y(v(1)), ..., y(v(n))]`. De cette façon, il est

possible d'adresser des ensembles d'éléments non contigus d'une matrice ou d'un vecteur.

```
y=1:10:80
y =
     1    11    21    31    41    51    61    71
x=8:-2:1
x =
     8     6     4     2
y(x)
ans =
     71    51    31    11
```

Voici des exemples d'extraction de sous-matrices de la matrice `A`:

```
B = A(1:2,2)
B =
     2
     5
B = A(1:2,2:3)
B =
     2     3
     5     7
A(:,1:2)
B =
     1     2
     4     5
     7     8
```

Exemple 4.2 Résolution de la variable x_3 du système linéaire $Ax = b$ avec la règle de Cramer.

```
A = [1 9 3; 4 7 7; 7 2 0]; b = [47 57 22]';
det( [A(:,1:2) b] ) / det(A)
ans =
    3
x = A\b;      (pour vérification)
```

Utilisation de vecteurs pour indicer une matrice:

```
B = A([1 3],[1 3]);
B =
     1     3
     7     0
l = [1 3]; c = [1 3]; B = A(l,c);      (même effet)
A(:,2:-1:1)            (renverse l'ordre des colonnes)
```

Les fonctions `fliplr` et `flipud` renversent l'ordre des colonnes, respectivement lignes.

Il est possible de transformer une matrice en un vecteur qui correspond aux colonnes concaténées de la matrice (opérateur *vec*). Soit la matrice `A`

```
A = [1 2 3 4;5 6 7 8;9 10 11 12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

alors

```
b = A(:)'
b =
    1   5   9   2   6  10   3   7  11   4   8  12
```

On peut également reformater (reshape) une matrice:

```
B = zeros(2,6); B(:)  = A
B =
     1     9     6     3    11     8
     5     2    10     7     4    12
```

Ce même résultat peut être obtenu directement avec la fonction `reshape`.

```
B = reshape(A,2,6);
```

4.3 VECTEURS LOGIQUES COMME INDICES

Des vecteurs `0-1` généralement obtenus à partir d'opérations relationnelles peuvent être utilisés pour définir des sous-matrices. Soit une matrice `A` et un vecteur logique `k` (avec éléments soit `1`, soit `0`), il est alors possible d'extraire une sous-matrice de la façon suivante:

```
A = [1 2; 3 4; 5 6];  k = [1 0 1];
A(k,:)
     1     2
     5     6
```

Attention: Dans les versions 5.x de MATLAB il n'est plus possible d'utiliser des vecteurs logiques de cette façon. Ci-après un autre exemple d'utilisation d'un vecteur logique pour extraire un sous-ensemble d'éléments d'une matrice.

La commande suivante élimine tous les éléments plus grands ou égaux à 4 du vecteur `x`:

```
x = fix(10*rand(1,8))
x =
    7     2     0     7     3     6     7     9
x = x(x>=4)
x =
     7     7     6     7     9
```

Cette syntaxe est acceptée par MATLAB 5.x.

4.4 MATRICES VIDES

La commande suivante crée une matrice vide

```
A = []
```

il devient alors possible d'utiliser cette matrice ultérieurement. La commande

```
clear A
```

a un effet différent, en ce sens qu'elle efface la variable `A` de l'espace travail. La fonction `exist` permet de vérifier l'existence d'une matrice et la fonction `isempty` indique si la matrice est vide ou non.

Une matrice vide peut être utilisée pour détruire des lignes ou des colonnes d'une matrice:

```
A(2,:)  = [];
```
(efface la deuxième ligne de A)

4.5 MATRICES PARTICULIÈRES

Il existe un ensemble de fonctions pour créer des matrices particulières.

Matrices particulières	
`zeros`	Matrice nulle
`ones`	Matrice constante
`rand`	Matrice aléatoire (loi uniforme)
`randn`	Matrice aléatoire (loi normale)
`eye`	Matrice identité
`linspace`	Vecteur à éléments équidistants
`logspace`	Eléments équidistants (échelle logarithmique)
`meshdom`	Domaine pour graphiques 3D

4.6 MANIPULATION DE MATRICES

MANIPULATION DE MATRICES	
`diag`	Extraction ou création de la diagonale
`tril`	Extraction triangle inférieur
`triu`	Extraction triangle supérieur
`reshape`	Reformatage
`:`	Reformatage
`'`	Transposition
`fliplr`	Inversion de l'ordre des colonnes
`flipud`	Inversion de l'ordre des lignes
`rot90`	Rotation

Pour la manipulation de matrices il faut encore mentionner les fonctions `size` et `length` qui renvoient la dimension d'une matrice, respectivement la longueur d'un vecteur.

4.7 MATRICES COMME ÉLÉMENTS D'UNE MATRICE

Il est possible de former des matrices dans lesquelles les éléments sont constitués par des matrices plus petites.

La matrice $\begin{bmatrix} n & \iota' Z \\ Z'\iota & Z'Z \end{bmatrix}$ peut être construite comme suit:

```
M = [n ones(1,n)*Z; Z'*ones(n,1)  Z'*Z]
```

Exemple 4.3 Calcul de la matrice des covariances MC correspondant à une matrice d'observations X.

```
k=3; X = randn(n,k);        (générer une matrice aléatoire)
XC = X - ones(n,1) * sum(X)/n;        (centrage des observations)
MC = XC' * XC /(n-1);
```

Observer la matrice MC pour un petit, puis un grand nombre d'observations.

5

CONTRÔLE DE L'EXÉCUTION

MATLAB possède un jeu d'instructions permettant le contrôle de l'exécution qui est semblable à ce que l'on trouve dans la plupart des langages de programmation. C'est grâce à ces instructions que MATLAB peut être utilisé comme un langage de programmation très puissant.

5.1 BOUCLE "FOR"

A l'aide de l'instruction `for`, il est possible de répéter l'exécution d'un jeu d'instructions un nombre de fois donné. La syntaxe d'une boucle `for` est la suivante:

```
for v =  expression
     instructions
end
```

L'*expression* est constituée par une matrice. Les colonnes sont assignées l'une après l'autre à la variable `v` et les instructions exécutées. Ci-après des exemples:

```
for i = 1:n
    x(i) = 0
end
```

Assigne la valeur zéro aux `n` premiers éléments de `x`. L'instruction reste correcte si `n` est inférieur à 1 mais l'instruction n'est pas exécutée. En effet, ceci résulte en une *expression* qui est vide. Si le vecteur `x` contient moins de `n` éléments, ou bien s'il n'existe pas, il sera alloué dynamiquement.

En utilisant la virgule comme séparateur de commandes, les instructions ci-dessus peuvent également s'écrire sur une seule ligne

```
for i = 1:n, x(i) = 0, end
```

Des boucles peuvent être imbriquées

```
for i = 1:n
    for j = 1:n
        A(i,j) = abs(i-j) + 1;
    end
end
A
```

On remarque que, à chaque `for`, doit correspondre une instruction `end`.

MATLAB n'est pas sensible aux indentations dans l'écriture du code. Le recours aux indentations contribue cependant à une meilleure lisibilité du programme du fait que les structures apparaissent plus clairement. Cette pratique est vivement recommandée.

5.2 BOUCLE "WHILE"

Avec l'instruction `while`, il est possible de répéter l'exécution d'un jeu d'instructions un nombre indéfini de fois sous le contrôle d'une condition logique. Les variables définissant la condition logique sont évidemment modifiées par les instructions exécutées. La syntaxe d'une boucle `while` est la suivante:

```
while  expression
    instructions
end
```

Les instructions sont exécutées aussi longtemps qu'il y a des éléments non nuls dans la matrice *expression*. En général l'*expression* est constituée par un scalaire et ainsi non-nul correspond à vrai. On rappelle qu'une *expression* matrice peut être réduite à un scalaire à l'aide des fontions `any` ou `all`.

Exemple 5.1 Rechercher le premier entier `n` pour lequel `n!` est un nombre à 100 digits:

```
n = 1;
while prod(1:n) < 1.e100
    n = n + 1;
end
n
```

Exemple 5.2 Déterminer la précision de la machine:

```
e = 1;
while 1+e > 1
    e = e/2;
end
prec_mach = 2*e
```

La boucle `while` est particulièrement utile pour programmer des procédés itératifs. La convergence d'un tel procédé n'est cependant pas toujours garantie. Afin d'éviter que, dans de tels cas, le nombre d'itérations ne devienne excessif, voir infini, il est très conseillé de prévoir un mécanisme d'arrêt en fonction du nombre d'itérations.

```
it = 0; itmax = 50;
while  expression
    instructions
    it = it + 1; if it > itmax, disp('Message'), break, end
end
```

La condition d'arrêt sur le nombre d'itérations pourrait directement intervenir dans l'*expression* (`( ... )` | `(it < itmax)`). En procédant de cette façon, l'utilisateur ne sait cependant pas quelle condition a provoquée l'arrêt de l'exécution.

5.3 INSTRUCTIONS "IF" ET "BREAK"

L'instruction `if` permet l'exécution d'un jeu d'instructions sous certaines conditions. La syntaxe de l'instruction `if` est la suivante:

```
if  expression1
    instructions1
elseif  expression2
    instructions2
elseif  expression3
    instructions3
    ...
else
    instructions
end
```

Si l'*expression1* après le `if` est non nulle (matrice ou scalaire), les *instructions1* sont exécutées et le programme se branche à l'instruction `end`. Dans le cas

contraire, matrice nulle ou scalaire nul, le programme se branche au prochain `elseif` et vérifie si l'*expression2* est "vraie" (matrice non nulle), etc.

Les instructions suivant le `else` sont exécutées si aucune des expressions précédentes n'est "vraie". Ce dernier ensemble d'instructions peut également être vide.

La forme sans `elseif` et la forme sans `elseif` ni `else` sont également possibles.

```
if  expression1
    instructions1
else
    instructions2
end
if  expression1
    instructions
end
```

Exemple 5.3 Ce problème de la théorie des nombres fait intervenir l'utilisation du `while` et du `if`. La fonction `input` permet de lire des données à partir du clavier et `break` interrompt l'exécution d'une boucle `for` et `while`. On choisit un nombre entier positif arbitraire n, s'il est pair on le divise par deux, sinon on calcule $n = 3n + 1$. On applique au nouveau n les mêmes règles jusqu'à ce que l'on obtienne $n = 1$. La question (encore ouverte) est s'il existe un nombre n pour lequel le procédé ne converge pas vers 1.

```
% Probleme "3n+1" de la theorie des nombres
while 1
    n = input('Donner n, (n<0 => arret) --> ');
    if n<=0, break, end
    while n>1
        if rem(n,2) == 0
            n = n/2
        else
            n = 3*n+1
        end
    end
end
```

6

FICHIERS `M`

MATLAB exécute les commandes au fur et à mesure qu'elles sont tapées en affichant, le cas échéant, le résultat. Il est également possible d'exécuter des séquences de commandes contenues dans un fichier.

Les fichiers contenant des commandes MATLAB sont appelés *fichiers m* étant donné que l'extension de tels fichiers est obligatoirement "`.m`". Ainsi, un fichier `mco.m` pourraît contenir les commandes pour calculer un estimateur des moindres carrés ordinaires.

Ainsi, un *fichier m* consiste simplement en une collection d'instructions MATLAB. Il peut faire appel à d'autres *fichiers m* et peut même s'appeler soit-même, de manière récursive.

Une raison qui motive l'utilisation des *fichiers m* consiste à pouvoir exécuter une procédure avec une seule commande (le nom du fichier contenant la procédure). De tels fichiers sont appelés fichiers *script*.

Un autre type de fichiers est constitué par les *fichiers function* qui permettent à l'utilisateur d'introduire dans MATLAB des fonctions spécifiques à ses problèmes particuliers.

Les deux types de fichiers *script* et *function* sont de simples fichiers ASCII que l'on manipulera avec l'éditeur de texte de son choix.

6.1 FICHIERS "SCRIPT"

Lorsque l'on tape le nom d'un fichier *script* MATLAB exécute simplement les commandes qu'il trouve dans le fichier. Les variables définies dans un fichier *script* sont définies globalement, c'est-à-dire qu'elles sont conservées dans la mémoire vive de MATLAB.

Dans l'utilisation courante de MATLAB, on aura sans cesse recours à de tels fichiers. Pour le problème de la théorie des nombres de la section précédente par exemple, on créera un fichier `thnombre.m` qui contiendra les instructions. En tapant simplement `thnombre` (sans l'extension) on exécute les instructions.

Exemple 6.1 Fichier *script* qui affiche le jour, le mois et l'année. Les commandes suivantes se trouvent dans le fichier `chrono.m`

```
% Fichier M qui affiche jour, mois, annee
t = clock;
disp([ t([3 2 1 4 5]) fix(t(6)) ])
```

Les commandes s'exécutent en tapant le nom du fichier:

```
chrono
    14          11        1996          20          13        45
```

6.2 FICHIERS "FUNCTION"

Si le premier mot de la première ligne d'un fichier contient le mot `function`, il s'agit d'un fichier *function*. Une *function* est l'équivalent d'un sous-programme FORTRAN, dans le sens qu'il est possible de passer des arguments et que les variables à l'intérieur de la fonction sont locales, ce qui n'est le cas des fichiers *script*.

Il est possible de créer des fonctions qui font à leur tour appel à des fonctions.

Exemple 6.2 Exemple d'une fonction qui calcule la moyenne des colonnes d'une matrice. Le fichier `moyenne.m` contient les instructions suivantes:

```
function xbar = moyenne(X)
% MOYENNE Calcul de la moyenne
%    Si X est une matrice xbar est un vecteur avec les
%    moyennes de chaque colonne de X.
[nobs,nc] = size(X);
```

```
if nobs == 1,
    nobs = nc;  % Traiter le cas d'un vecteur ligne
end
xbar = sum(X) / nobs;
```

On pourra alors exécuter cette fonction de la manière suivante:

```
y = 1:99;
ybar = moyenne(y)
ybar =
      50
```

Une fonction est notamment caractérisée par ce qui suit:

- Dans la première ligne, on déclare le nom de la fonction, les arguments d'entrée et les arguments de sortie. Sans cette ligne, le fichier serait un fichier *script*.
- Les lignes suivantes, qui sont précédées par le symbole `%`, documentent la fonction. Cette documentation s'affiche lorsqu'on donne l'instruction `help moyenne`. La commande `lookfor` explore la première ligne du commentaire.
- Les variables `X`, `nobs`, `nc` et `xbar` sont locales; elles n'existent plus une fois l'exécution de la fonction `moyenne` terminée.
- Le nom de l'argument de la fonction, `y` dans notre cas, peut être quelconque. C'est la fonction qui établit la correspondance avec la variable `X`.

Les fonctions permettent la définition de plusieurs arguments à l'entrée et à la sortie.

Exemple 6.3 Fonction `stats` avec deux arguments de sortie: la moyenne et l'écart-type des colonnes d'une matrice:

```
function [mu,sigma] = stats(X)
% STATS Calcule la moyenne et l'ecart-type
%     [mu,sigma] = stats(X) moyenne du vecteur ou des colonnes
%     de la matrice X.
mu = moyenne(X);
if nargout == 2
    [nobs,nc] = size(X);
    if nobs == 1, nobs = nc; end
    sigma = sqrt(sum(X.^2) / nobs - mu.^2);
end
```

Il est possible d'appeler une fonction avec un nombre d'arguments inférieur à celui prévu dans la fonction. Les fonctions `nargin` et `nargout` permettent de déterminer le nombre d'arguments effectivement passés à l'appel de la fonction. On ne pourra évidemment pas utiliser plus d'arguments que prévus par une fonction particulière.

Encore quelques remarques importantes concernant les fonctions:

- Lorsqu'une fonction est appelée pour la première fois durant une séance MATLAB, elle est compilée et copiée en mémoire vive. Des appels successifs se font alors sans que la fonction soit recompilée.
- La commande `what` liste tous les fichiers `.m` définis dans le répertoire courant. La commande `which` *nom_fonction* donne le nom du repertoire dans lequel se trouve la fonction. La commande `type` affiche un fichier `.m` à l'écran.
- Afin d'éviter des confusions entre noms de variable et noms de fichiers, il est important de savoir dans quel ordre l'interpréteur de MATLAB procède. Si l'on tape `essai` par exemple, MATLAB procède comme suit:
  - vérifie si `essai` est une variable;
  - vérifie si `essai` est une fontion codée de MATLAB;
  - vérifie si, dans le répertoire courant, existe un fichier `essai.m` (script ou function);
  - explore tous les répertoires du `MATLABPATH` pour vérifier l'existence d'un fichier `essai.m`.

  Ainsi, MATLAB essaie d'abord d'utiliser `essai` comme variable, avant d'essayer de l'utiliser comme fonction.

6.3 UTILITAIRES LIÉS AUX FICHIERS M

En tapant des commandes l'une après l'autre l'interactivité entre l'utilisateur et MATLAB est totale. En exécutant un fichier `.m` cette interactivité est perdue. Les fonctions `echo`, `input`, `keyboard` et `pause` permettent d'avoir une interaction plus grande lors de l'exécution de fichiers `.m`.

Fonction "echo"

Les commandes d'un fichier `.m` ne défilent pas à l'écran lorsqu'on l'exécute. Lors de la mise au point d'un programme ou pour les besoins d'une démonstration il peut être utile de voir la séquence de commandes qui s'exécutent. La fonction `echo` permet de faire défiler les commandes d'un fichier `.m`.

Pour les fichiers *script* on spécifie `echo on` pour démarrer le défilement et `echo off` pour arrêter le défilement.

Pour des fichiers *function* il faut donner le nom du fichier pour lequel on désire faire défiler les commandes. La syntaxe est `echo` *nom_fichier* `on`. On arrête le défilement avec `echo` *nom_fichier* `off`.

Fonction "input"

La fonction `input` affiche un prompt défini par l'utilisateur et attend la saisie d'une donnée.

```
k = input(' Donner n --> ')
Donner n --> 5
k =
     5
```

Dans cet exemple la variable `k` prendra la valeur de la réponse saisie après le prompt.

Losqu'on saisit un vecteur ou une matrice il faut veiller à respecter la syntaxe MATLAB pour la saisie:

```
v = input(' Donner le vecteur --> ')
Donner le vecteur --> [5 7 2]'
v =
     5     7     2
```

Pour la saisie d'une variable qui n'est pas numérique la syntaxe est la suivante:

```
reponse = input(' Repondre oui/non --> [oui]','s');
Repondre oui/non --> [oui]
if isempty(reponse), reponse = 'oui'; end
reponse =
oui
```

Avec la fonction `input` il est entre autres possible de piloter le déroulement d'un programme à l'aide de menus (voir également la fonction `menu`).

Fonction "keyboard"

La fonction `keyboard` est similaire à la fonction `input`. Elle est cependant plus puissante étant donné qu'elle confère le contrôle complet au clavier en l'assimilant à un fichier *script*. Cet état est caractérisé par le prompt particulier suivant:

```
K>>
```

Pour retourner le contrôle au fichier `.m` qui contient la commande `keyboard` il suffit de taper `return` en toutes lettres (il ne s'agit pas d'appuyer sur la touche Return).

La fonction `keyboard` est particulièrement utile lors de la mise au point d'un programme car elle permet d'examiner les variables et le cas echéant de continuer l'exécution après avoir modifié leur valeur.

Fonction "pause"

La commande `pause` interrompt l'exécution d'un fichier .m à l'endroit où la commande est donnée. L'exécution redémarre en tapant sur n'importe quelle touche.

L'exécution peut aussi être interompue pendant un temps limité avec la commande `pause(n)` ou `n` spécifie la durée en secondes.

6.4 DÉBOGAGE DES FONCTIONS

MATLAB identifie les erreurs de syntaxe lors de la compilation. Leur correction ne pose en général pas de problèmes. L'identification d'erreurs lors de l'exécution est plus délicate étant donné que les variables locales aux fonctions ne sont plus connues après l'arrêt forcé de l'exécution. La suppression des ; et les commandes `echo` et `keyboard` peuvent être utilisés pour afficher des résultats intermédiaires. (Le cas échéant on peut aussi transformer temporairement une `function` en fichier `script`).

La façon la plus efficace pour déboguer une `function` est d'utiliser les commandes suivantes disponibles dans MATLAB:

Commandes de débogage	
`dbtype`	Affichage d'un fichier `.m` avec numérotation des lignes
`dbstop`	Placer une interruption du programme
`dbclear`	Enlever les interruptions du programme
`dbstatus`	Liste des interruptions de programme
`dbcont`	Reprendre l'exécution du programme
`dbstep`	Exécuter une ou plusieurs lignes
`dbdown`	Changer espace de travail
`dbup`	Changer espace de travail
`dbstack`	Liste de l'enchaînement des appels de fonctions
`dbquit`	Arrêt du mode de débogage

Exemple 6.4 Illustration d'une séance de débogage:

```
dbtype moyenne
dbstop in moyenne
dbtype stats
dbstop at 13 in stats
[m,s]=stats(randn(15,3))
7   [nobs, ncols] = size(X);
dbstack
  In c:\ chemin \moyenne.m at line 7
  In c:\ chemin \stats.m at line 9
whos
dbup
In workspace belonging to  c:\ chemin \stats.m
whos
dbdown
In workspace belonging to  c:\ chemin \moyenne.m
dbstep
8   if nobs == 1
```

```
dbcont
  13        sigma = sqrt(sum(X.^2) / nobs - mu.^2);
whos
dbquit
```

6.5 CHAîNES DE CARACTÈRES

Toute donnée, qu'elle soit numérique ou constituée par un caractère, est toujours representée dans l'ordinateur par un mot de 32 bits (la séquence de bits dans le mot permet de representer des entiers).

Pour coder des caractères on se sert d'une table qui fait correspondre à un caractère donné un entier. Une table (standard) largement utilisée est l'ASCII (American Standard Code for Information Interchange). A titre d'exemple quelques caractères et le code ASCII correspondant:

Mot binaire	Valeur decimale	Caractère
00000000000000000000000001000001	65	A
00000000000000000000000001100001	97	a
00000000000000000000000000110000	48	0

Au moment ou l'on demande l'affichage du contenu d'une variable (mot) MATLAB doit savoir si la variable doit être oui ou non convertie en un caractère lors de l'affichage. Cette information fait partie du codage sous la forme d'un *flag* (bit dans une position particulière du mot).

Pour instruire MATLAB que la variable est à considérer comme une chaîne de caractères (initialiser le *flag*) on utilise la syntaxe suivante:

```
t = 'Resultat'
t =
Resultat
```

Comme déjà mentionné, MATLAB utilise un mot pour représenter chaque caractère. Pour la variable `t` ceci correspond donc à un vecteur de 8 éléments. En effet on a

```
size(t)
ans =
     1     8
```

La fonction `abs`, appliquée à une variable caractère, affiche son contenu numérique, ce qui permet de découvrir le code utilisé. Pour notre variable `t` on aura

```
abs(t)
ans =
    82   101   115   117   108   116    97   116
   ( R     e     s     u     l     t     a     t )
```

et on reconnait notamment le code 97 pour la lettre `a`. La fonction `setstr` performe l'opération contraire

```
setstr([108 117 115 116])
ans =
lust
```

Des chaînes de caractères peuvent être concaténées de la même façon que des vecteurs.

Exemple 6.5 La fonction `isstr` renseigne s'il s'agit d'une chaîne de caractères et `str2mat` construit une matrice en complétant les caractères manquants par des blancs.

```
if isstr(t)
   s = [t,' final'];
end
s =
Resultat final
s = str2mat(t,'final')
s =
Resultat
final
size(s)
ans =
     2     8
```

Version améliorée de l'exemple 6.1:

```
function c = chrono()
% CHRONO  Formate la date en une chaine de caracteres
%         c = chrono  formate la date comme variable caractere
%
mois = str2mat('Janvier','Fevrier','Mars','Avril','Mai','Juin');
mois = str2mat(mois,'Juillet','Aout','Septembre','Octobre');
mois = str2mat(mois,'Novembre','Decembre');
t = clock;
j = [int2str(t(3)),' ',deblank(mois(t(2),:)),' ',int2str(t(1))];
h = [int2str(t(4)),'h ',int2str(t(5)),'min ',int2str(fix(t(6))),'sec'];
c = [j,' -- ',h];
```

D'autres fonctions MATLAB liées à la manipulation de chaînes de caractères sont: `blanks`, `deblank`, `upper`, `lower`, `findstr`, `strcmp`, `strrep` et `strtok`.

Conversion et impression

Pour imprimer des résultats dans un format particulier, autre que le défaut de MATLAB, il est nécessaire de convertir des valeurs numériques en chaînes de

caractères. Les fonctions `num2str`, `int2str`, `sprintf` et `fprintf` permettent cette conversion. En général les variables converties sont ensuite concatenées en de chaînes de caractères.

La fonction `int2str` convertit des entiers en une chaîne de caractères et `num2str` convertit des réels en une chaîne de caractères.

```
for i = 1:10
    disp(['Variable y',int2str(i)])
    ...
end
```

Pour contrôler entièrement le format d'output on utilise `sprintf`. La syntaxe est:

```
sprintf(' format',x)
```

où *format* spécifie un texte et la conversion de la variable `x`.

```
j = 3; b(j) = exp(pi);
t = sprintf('beta(%i) = %5.2f',j,b(j));
disp(t)
```

```
beta(3) =  23.14
```

D'autres spécificateurs de conversion sont `%e` la notation exponentielle et `%g` la notation en point flottant. Par exemple `%6.2f` convertit un nombre dans le champ suivant ␣␣␣␣.␣␣. En convertissant le nombre 1.3256 avec ce format on obtient ␣␣`1.33` et `%-6.2f` aligne l'output à gauche `1.33`␣␣.

La fonction `fprintf` produit la même conversion que `sprintf` mais écrit la chaîne de caractères dans un fichier. On a

```
fid = fopen(' nom_fichier',' mode');
fprintf(fid,' format',x);
```

La fonction `fopen` établit la correspondance entre le fichier et l'unité logique `fid`. L'argument *mode* définit le type d'accès au fichier. Le défaut est la lecture `'r'`. Parmi les autres options on a `'w'` pour écriture et `'a'` pour append. Si l'on omet de spécifier *nom_fichier* le résultat est affiché à l'écran. L'output correspondant à `fprintf` s'affiche en suivant la position courante du curseur. Pour provoquer un retour à la ligne on utilise `\n` et `\t` a l'effet d'un tabulateur.

Fonction "eval"

On peut créer dynamiquement n'importe quelle chaîne de caractères et utiliser ensuite la fonction `eval` pour l'interpréter et exécuter comme une instruction MATLAB. Par exemple on peut écrire:

```
t = 'x = z^2 + 7'; z = 3;
eval(t)
x =
    16
```

Exemple 6.6 Utilisation de la fonction `eval` pour la résolution d'un système d'équation non-linéaires avec la méthode de Gauss-Seidel:

```
eq = str2mat('x(1)=-.3*x(3)+2.5;','x(2)=-1.3*x(1)+.26*x(3)-1;', ...
             'x(3)=x(1)*(2+5/x(2));');
tol = .001; itmax = 50; it=0;  not_converged = 1;
eq_ordre = [1 3 2];  x = ones(3,1);
while not_converged
     x0 = x;
     for i = 1:3
         eval(eq(eq_ordre(i),:));
     end
     it = it + 1; if it > itmax, error('...  Maxit !'), end
     not_converged = any(abs(x-x0)./(abs(x0)+1) > tol);
end
```

7

OPTIMISATION D'UN PROGRAMME

Pour évaluer la performance d'un programme on peut mesurer le temps nécessaire à son exécution et compter le nombre d'opérations élémentaires exécutées.

La fonction `toc` retourne le temps écoulé depuis le dernier appel à `tic`. La fonction `flops` permet d'obtenir le nombre à peu près exact d'opérations élémentaires $(+ - */)$ exécutés par un programme sans pour autant en dégrader la performance. Voici ce nombre pour quelques opérations:

Opération	flops
`A + B`	n^2
`A * B`	$2n^3$
`A^100`	$99 \times 2n^3$

Voici un exemple comment mesurer le nombre d'opérations élémentaires et le temps d'exécution:

```
tic; cf = flops;
    instructions
fprintf('\n%i flops %8.2f sec',flops-cf,toc);
```

Les instructions d'un programme MATLAB ne sont pas compilées et traduites en code machine comme c'est le cas par exemple pour un programme FORTRAN. MATLAB interprète les expressions puis les exécute. Lorsqu'on appelle une fonction MATLAB qui est codée l'exécution est de plusieurs ordres plus rapide que s'il agit d'un programme comportant des `for` ou `while` qui doivent être interprétés à chaque passage. D'ou les règles suivantes:

- Toujours préférer des expressions vectorielles au constructions `for`;
- Minimiser le nombre de fois qu'une expression doit être interprétée;
- Eviter l'allocation dynamique d'une matrice.

L'effet en termes d'efficacité de ces pratiques est illustré avec les exemples qui suivent[1].

Exemple 7.1 Comparaison de deux évaluations de l'expression $\sum x_i^2$, l'une calculée au moyen d'une boucle `for` et l'autre au moyen d'une évaluation vectorielle.

```
tic; cf = flops;
x = rand(40000,1); s = 0;
for i = 1:40000
     s = s + x(i) * x(i);
end
fprintf('\n%i flops %8.2f sec',flops-cf,toc);

     80000 flops    2.86 sec

tic; cf = flops;
x = rand(40000,1);
s = x' * x;
fprintf('\n%i flops %8.2f sec',flops-cf,toc);

    80000 flops    0.05 sec
```

Le code vectoriel est environ 60 fois plus rapide à l'exécution. La performance est d'environ 2 Mflops par seconde.

Exemple 7.2 Soit deux boucles `for` imbriquées qui definissent les éléments d'une matrice $A \in \mathbb{R}^{40000\times 2}$:

```
clear A; tic; cf = flops;
for i = 1:40000
     for j = 1:2
          A(i,j) = i + j;
     end
end
fprintf('\n%i flops %8.2f sec',flops-cf,toc);

    80000 flops  646.96 sec
```

En plaçant la boucle sur `j` à l'extérieur le deuxième `for` ne doit être interprété que deux fois au lieu de 40000.

[1] Calculs effectués avec PC Pentium 133 MHz.

```
clear A; tic; cf = flops;
for j = 1:2
     for i = 1:40000
          A(i,j) = i + j;
     end
end
fprintf('\n%i flops %8.2f sec',flops-cf,toc);
```

```
    80000 flops  272.32 sec
```

En définissant la matrice `A` on évite l'allocation dynamique pendant l'exécution de la boucle.

```
clear A; tic; cf = flops; A = zeros(40000,2);
for j = 1:2
     for i = 1:40000
          A(i,j) = i + j;
     end
end
fprintf('\n%i flops %8.2f sec',flops-cf,toc);
```

```
    80000 flops   3.74 sec
```

On remarque que la différence de la vitesse d'exécution entre la première et la dernière version du code est de deux ordres de magnitude.

8

FONCTIONS ORIENTÉES COLONNES

MATLAB possède un ensemble de fonctions qui agissent sur les colonnes d'une matrice. Certaines constituent des fonctions statistiques élémentaires. L'utilisation de ces fonctions permet souvent d'éviter la programmation des boucles `for`.

Voici cette liste de fonctions:

Fonctions agissant sur les colonnes	
`max`	Maximum
`min`	Minimum
`mean`	Valeur moyenne
`median`	Valeur médiane
`std`	Ecart quadratique moyen
`sort`	Tri
`sum`	Somme de éléments
`prod`	Produit des éléments
`cumsum`	Somme cumulée des éléments
`diff`	Différence des éléments
`corrcoef`	Coefficients de corrélation
`cov`	Matrice de covariance

Si l'argument de ces fonctions est un vecteur, il est indifférent qu'il s'agisse d'un vecteur ligne ou d'un vecteur colonne.

```
A = [9 1 8; 3 7 5; 6 4 2];
mx = max(A)
mx =
     9     7     8
mo = mean(A)
```

```
mo =
     6     4     5

[B I] = sort(A)
B =
     3     2     2
     6     4     5
     9     7     8
I =
     2     1     3
     3     3     2
     1     2     1
```

Exemple 8.1 Centrage et normalisation des colonnes d'une matrice X:

```
Z = ( X - ones(n,1) * mean(X) ) ./ ( ones(n,1) * std(X) )
```

$$Z = (X - \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \begin{bmatrix} m_1 & \dots & m_k \end{bmatrix}) ./ \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \begin{bmatrix} \sigma_1 & \dots & \sigma_k \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} - m_1 & \dots & x_{1k} - m_k \\ \vdots & & \vdots \\ x_{n1} - m_1 & \dots & x_{nk} - m_k \end{bmatrix} ./ \begin{bmatrix} \sigma_1 & \dots & \sigma_k \\ \vdots & & \vdots \\ \sigma_1 & \dots & \sigma_k \end{bmatrix}$$

Exemple 8.2 Genérer k entiers uniforment distribuées dans l'intervalle $[a, b[$, puis les trier dans l'ordre croissant.

```
a = 5; b = 15; k = 10;
u = fix( (b - a) * rand(1,k) ) + a ;
[s,p] = sort(u);
```

Le vecteur s correspond au vecteur u auxquel on a appliqué la permutation p. Ainsi on a $s = u(p)$. Pour obtenir la permutation inverse q qui permet de revenir de s à u on procède comme:

```
q(p) = 1:k;
u = s(q);
```

9

FONCTIONS MATRICIELLES

La clef de la puissance de MATLAB, ainsi que de la qualité numérique des résultats, réside essentiellement dans ses fonctions matricielles. En partie, il s'agit de routines LINPACK et EISPACK codées directement en MATLAB ou de fichiers `.m`. Ces fonctions sont de quatre types:

- Factorisation triangulaire
- Factorisation orthogonale
- Décomposition singulière
- Décomposition des valeurs propres

Fonctions liées aux décompositions matricielles	
`lu`	Factorisation triangulaire
`chol`	Factorisation de Cholesky
`inv`	Inverse d'une matrice
`det`	Déterminant
`qr`	Factorisation orthogonale
`null`	Espace nul
`orth`	Base orthonormale
`svd`	Décomposition singulière
`pinv`	Pseudoinverse
`rcond`	Condition d'une matrice
`rank`	Rang d'une matrice
`norm`	Normes d'une matrice
`eig`	Valeurs propres et vecteurs propres

9.1 FACTORISATION LU

Une factorisation fondamentale pour le calcul numérique consiste à exprimer une matrice carrée comme le produit de deux matrices, soit

$$A = LU$$

avec L une matrice triangulaire inférieure et U une matrice triangulaire supérieure. Cette factorisation est souvent appellée *factorisation LU* (parfois LR). L'algorithme à la base de cette factorisation est l'élimination de Gauss.

L'utilisation de la factorisation LU est souvent transparente dans MATLAB. On y recourt notamment dans les situations suivantes:

- Lorsqu'on calcule l'inverse d'une matrice avec la commande `inv(A)`, MATLAB fait usage de la relation suivante:

  $$A^{-1} = U^{-1}L^{-1} \quad \text{étant donné que} \quad (AB)^{-1} = B^{-1}A^{-1}$$

  MATLAB factorise donc d'abord la matrice A, puis cherche l'inverse de deux matrices triangulaires pour former ensuite le produit.

- Le déterminant `det(A)` est calculé à partir de la relation

  $$\det(A) = \det(L) \times \det(U)$$

  avec $\det(L) = 1$, étant donné que les éléments de la diagonale de L sont tous égaux à 1. Pratiquement MATLAB cherche la factorisation LU d'une permutation de PA de la matrice A et de ce fait le déterminant correspond à `prod(diag(U)) * det(P)`.

- La résolution d'un système linéaire $Ax = b$ s'obtient avec la commande `x = A\b`. MATLAB factorise la matrice A et résoud le système $LUx = b$ en deux étapes ($L[\underbrace{Ux}_{y}] = b$). D'abord on résoud le système triangulaire

  $$Ly = b \quad \text{avec} \quad y = Ux$$

  puis le système triangulaire

  $$Ux = y \quad .$$

La fonction `lu` de MATLAB calcule une matrice L qui est une permutation d'une matrice triangulaire. Par exemple

```
A = [1 1 2; 3 4 5; 6 7 8];
[L,U] = lu(A)

L =
    0.1667   -0.3333    1.0000
    0.5000    1.0000         0
    1.0000         0         0

U =
    6.0000    7.0000    8.0000
         0    0.5000    1.0000
         0         0    1.0000
```

On vérifie bien que $LU = A$. On peut obtenir la matrice L dans la forme triangulaire inférieure en factorisant une permutation de la matrice A. En appelant `lu` comme suit:

```
[L,U,P] = lu(A)
```

on obtient la matrice P de permutation et on vérifie que dans ce cas matrice L est triangulaire inférieure et que le produit LU correspond à une factorisation de la permutation PA de A, c'est-à-dire

$$LU = PA$$

9.2 FACTORISATION `QR`

La factorisation orthogonale s'applique à des matrices carrées et à des matrices rectangulaires. Elle consiste à trouver deux matrices Q et R, telles que

$$A = QR$$

avec Q une matrice orthogonale et R une matrice triangulaire supérieure. Cette factorisation permet de résoudre des systèmes linéaires avec plus d'équations que d'inconnues. Soit le système linéaire

$$Ax = b \quad \text{et la factorisation} \quad A = QR$$

on remplace A par la factorisation

$$QRx = b$$

et comme Q est orthogonale on peut écrire

$$Rx = Q'b$$

et x est la solution d'un système triangulaire $Rx = y$ avec $y = Q'b$.

Soit le système linéaire suivant et sa solution:

```
A = [1 2; 3 4; 5 6]; b = [2 3 4]';
x = A\b

x =
   -1.0000
    1.5000
```

Lorsque MATLAB rencontre l'opérateur \ il détermine d'abord quel type de système linéaire doit être résolu. Dans ce cas il procède ensuite de la façon suivante:

```
[Q,R] = qr(A)

Q =                                    R =
   -0.1690    0.8971    0.4082            -5.9161  -7.4374
   -0.5071    0.2760   -0.8165                  0   0.8281
   -0.8452   -0.3450    0.4082                  0        0

y = Q'*b

y =
   -5.2400
    1.2421
    0.0000

x = R\y

x =
   -1.0000
    1.5000
```

9.3 DÉCOMPOSITION SINGULIÈRE SVD

La décomposition singulière est une technique particulièrement importante dans de nombreux problèmes de calcul matriciel. Etant donné une matrice A, elle consiste à trouver trois matrices U, S et V, telles que l'on vérifie

$$A = USV'$$

Les matrices U et V sont des matrices orthogonales et S est une matrice diagonale dont les éléments correspondent aux valeurs singulières.

```
A = [1 2; 3 4; 5 6];
[U,S,V] = svd(A)

U =                                    S =
    0.2298    0.8835    0.4082            9.5255         0
    0.5247    0.2408   -0.8165                 0    0.5143
    0.8196   -0.4019    0.4082                 0         0

V =
    0.6196   -0.7849
    0.7849    0.6196
```

9.4 VALEURS PROPRES

Les valeurs propres et les vecteurs propres d'une matrice carrée A sont définies par la relation

$$AV = VD$$

avec V la matrice des vecteurs propres et D la matrice diagonale des valeurs propres. La matrice A verifie la décomposition

$$A = VDV^{-1} \quad .$$

```
A = [1 2; 3 4];
[V,D] = eig(A)
```

```
V =                              D =
   -0.8246  -0.4160                 -0.3723         0
    0.5658  -0.9094                       0    5.3723
```

10

FONCTIONS DE FONCTIONS

MATLAB possède quelques fonctions particulières dont les arguments ne sont pas constitués par des matrices, mais par des fonctions. Ces fonctions concernent:

- Optimisation non linéaire
- Intégration numérique
- Equations différentielles

Voici une liste de quelques-unes de ces fonctions:

Fonctions de fonctions	
`fmin`	Minimum d'une fonction d'une variable
`fmins`	Minimum d'une fonction de plusieurs variables
`fzero`	Zéro d'une fonction d'une variable
`fsolve`	Solution d'un système de fonctions non linéaires
`quad`	Intégration numérique
`ode23`	Solution d'une équation différentielle

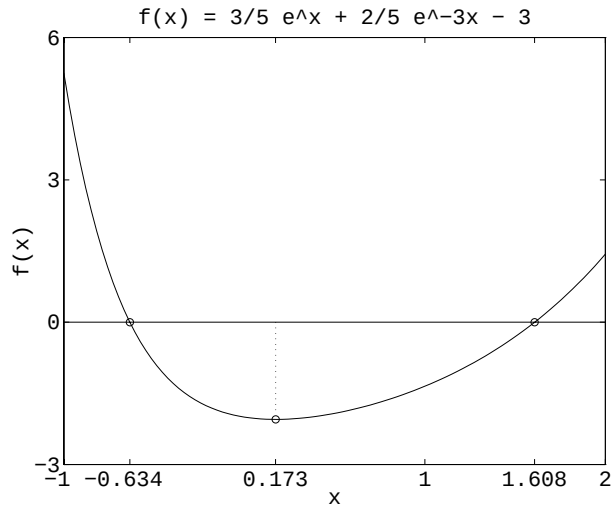
Exemple 10.1 Définissons une fonction MATLAB pour évaluer

$$y = \frac{3}{5}e^{x} + \frac{2}{5}e^{-3x} - 3.$$

On appellera cette fonction `fexpo` et on aura un fichier `fexpo.m` avec les commandes suivantes:

```
function y = fexpo(x)
y = 3./5*exp(x) + 2./5*exp(-3*x) - 3;
```

Lorsque x est un vecteur la fonction `fexpo` retourne y qui est aussi un vecteur. Le graphique de cette fonction dans l'intervalle $[-1, 2]$ peut être obtenu avec la commande `fplot('fexpo',[-1 2])` et est donné ci-après:



Dans les exemples d'utilisation de `fmin`, `fzero` et `quad` qui suivent on passe comme argument un nom de fonction sous forme de chaîne de caractères. Dans notre cas ce nom est `fexpo`.

Chercher le minimum dans l'intervalle $[-1, 2]$:

```
xmin = fmin('fexpo',-1,2)

xmin =
    0.1733
```

Chercher les zéros de la fonction dans le voisinage de $x = -.5$ et $x = 1.5$:

```
z1 = fzero('fexpo',-.5);
z2 = fzero('fexpo',1.5);

disp([z1 z2])
   -0.6343    1.6084
```

Intégrer `fexpo` de `z1` à `z2`:

```
s = quad('fexpo',z1,z2)

s =
   -3.1564
```

Pour la fonction `quad` il est impératif que la fonction donné comme argument retourne un vecteur.

Exemple 10.2 Résolution du système d'equations de l'exemple 6.6 a l'aide de la fonction `fsolve`. On doit d'abord créer une fonction qui evalue le système d'equations dans sa forme implicite, c'est-à-dire $F(x) = 0$. Soit `mod3eq` cette fonction:

```
function f = mod3eq(x)
% Systeme d'equations sous forme implicite ( f(x) = 0 )
%
f(1) = -0.3 * x(3) + 2.50 - x(1);
f(2) = -1.3 * x(1) + 0.26 * x(3) - 1 - x(2);
f(3) = x(1) * (2 + 5/x(2)) - x(3);
```

Ci-après des appels successifs à la fonction `fsolve` pour des valeurs initiales de x différentes. On remarque que la solution trouvé varie en fonction de la valeur initiale.

```
x = fsolve('mod3eq',[-3 -3 -3])
x =
    2.1406   -3.4713    1.1980

x = fsolve('mod3eq',[-2 -2 -2])
x =
    0.3930    0.3151    7.0232

x = fsolve('mod3eq',[23 23 23])
x =
    2.1406   -3.4713    1.1980
```

11

VISUALISATION

La visualisation de données au moyen de présentations graphiques constitue un outil puissant d'analyse.

Les fonctions de MATLAB permettent de produire des graphiques très élaborés dans lesquels l'utilisateur a le contrôle sur pratiquement tous les aspects de la présentation. Pour l'utilisateur, il n'est cependant pas nécessaire de spécifier les détails de la présentation étant donné que, pour chaque fonction, il existe une initialisation par défaut des paramètres qui définissent la présentation. Par la suite on n'entrera pas dans le détail de ce contrôle d'un graphique mais on ne présentera que les résultats produits par défaut.

Enumérons d'abord des catégories de fonctions qui contrôlent trois aspects d'un graphique, c'est-à-dire les coordonnées, l'annotation des axes et la dimension de la fenêtre graphique.

	Définition des coordonnées
`plot`	Coordonnées linéaires
`fplot`	Coordonnées linéaires
`loglog`	Coordonnées logarithmiques
`semilogx`	Coordonnées logarithmiques pour l'axe des x
`semilogy`	Coordonnées logarithmiques pour l'axe des y
`polar`	Coordonnées polaires
`mesh`	Graphique 3D
`contour`	Lignes de niveaux
`bar`	Diagramme en btons
`stairs`	Diagramme en escaliers
`hist`	Histogramme

ANNOTATIONS DU GRAPHIQUE	
`title`	Entête du graphique
`xlabel`	Annotations de l'axe des x
`ylabel`	Annotations de l'axe des y
`text`	Annotations dans le graphique
`gtext`	Annotations dans le graphique à l'aide de la souris
`grid`	Maillage du graphique

CADRAGE ET AUTRES CONTRÔLES	
`axis`	Cadrage manuel
`hold`	Surimpression
`clg`	"clear graph screen"
`subplot`	Définition de sous-fenêtres graphiques
`ginput`	Saisie de coordonnées avec la souris
`zoom`	Grossir le graphique avec la souris

11.1 GRAPHIQUES 2D

La commande `plot` crée un graphique avec une échelle linéaire pour l'axe des x et l'axe des y.

```
y = [.1 .5 1 3.5 4.5 2];
plot(y)
```

Sous cette forme, l'axe des x correspond aux indices des éléments de y. Si l'on désire obtenir des échelles logarithmiques ou polaires, on remplace simplement la commande `plot` par la commande `semilogy`, `loglog` ou `polar` par exemple.

MATLAB ajuste l'échelle des axes automatiquement aux données et affiche le dessin dans sa fenêtre graphique.

Pour deux vecteurs `x` et `y` de même longueur, la commande `plot(x,y)` affiche le graphique qui correspond aux points donnés par les coordonnées définies dans `x` et `y`.

```
x = linspace(1,8);
y = (7-x).*(x-3).^(1/3);
plot(x,y)
zoom on
```

Pour ce graphique, on remarque que 100 points ne suffisent pas pour obtenir une précision satisfaisante de la fonction dans un voisinage pour $x = 3$. On peut spécifier `x = linspace(1,8,400)`.

Une alternative est la fonction `fplot` qui calcule l'espacement des points en fonction de la courbure de la fonction. Dans ce cas, la fonction à dessiner est defini par une "function" MATLAB (fichier `.m`).

Exemple 11.1 Soit une fonction defini par la fonction MATLAB `drom.m`:

```
function y = drom(x)
y = (7-x)*(x-3)^(1/3);
```

L'appel à `fplot` se fait alors comme suit:

```
fplot('drom',[1 8])      ou    fplot('drom',[1 8 -1 3.5])
```

La fonction `fplot` peut aussi être utilisée sous la forme `fplot('log',[ ... ])` où `fplot('1/x',[ ... ])`. Si la fonction appellée par `fplot` évalue une matrice, chaque colonne de la matrice est dessinée.

Annotation du graphique

Après que la commande `plot` s'est exécutée, on peut annoter le graphique au moyen des commandes `title`, `xlabel` et `ylabel`.

```
title('Fonction drom')
xlabel('variable independante x')
ylabel('f(x)')
```

La commande `grid` permet de dessiner le maillage qui correspond aux échelles des axes. A l'aide de la fonction `gtext`, on peut annoter le graphique en choisissant l'emplacement avec la souris.

```
grid
gtext('<- non derivable')
```

Traçage de lignes multiples

Il est aussi possible de tracer plusieures lignes dans la même fenêtre graphique. Soit le vecteur `x` et la matrice `Y` définis comme suit:

```
x = (1:10)'; Y = x * linspace(1,3,6);
plot(log(x),Y)
semilogx(x,Y)      (produit le même résultat)
```

MATLAB trace successivement les colonnes de `Y`, en changeant de couleur pour chaque nouvelle courbe (le nombre de couleurs est limité). On peut aussi donner la commande `plot(log(x),Y')` étant donné que MATLAB vrifie si ce sont les lignes ou les colonnes de `Y` qui ont la même dimension que le vecteur `x`.

Comme le vecteur `x` prend des valeurs de 1 à 10, les deux commandes suivantes produisent le même résultat:

```
plot(x,Y)
plot(Y)
```

On peut encore utiliser la commande `plot` comme suit:

```
plot(Y,log(x))
X = randn(10,5); Y = randn(10,5);
plot(X,Y)
plot(X(:,1),Y(:,1),X(:,3),Y(:,3),X(:,5),Y(:,5))
```

Contrôle du style de lignes et de la couleur

Avec des arguments supplémentaires, on peut spécifier, dans la commande `plot`, le style de la ligne tracée ainsi que sa couleur. Dans l'exemple suivant, on marque d'abord les points d'un `+`, sans relier les points marqués, puis on trace la deuxième courbe avec une ligne pointillée en rouge.

```
plot(x,Y(:,1),'+',x,Y(:,2),':r')
```

Voici la liste des symboles définissant couleur et type de ligne:

Traits de ligne		Types de points		Couleurs	
`-`	continu	`.`	point	`y`	yaune
`--`	tiret	`+`	plus	`m`	magenta
`:`	pointillé	`*`	étoile	`c`	cyan
`-.`	tiret-point	`o`	cercle	`r`	rouge
		`x`	croix	`g`	vert
				`b`	bleu
				`w`	blank
				`k`	noir

Commande "axis" et "hold"

MATLAB définit automatiquement l'échelle de la fenêtre graphique en fonction de x_{inf}, x_{sup}, y_{inf} et y_{sup}. L'utilisateur peut rédéfinir cette fenêtre en spécifiant le vecteur $[x_{\text{inf}}\ x_{\text{sup}}\ y_{\text{inf}}\ y_{\text{sup}}]$. Par exemple, on peut retracer le graphique précédent dans la fenêtre suivante:

```
clg
plot(log(x),Y)
axis([-10 20 -5 20])
```

Chaque commande `plot` efface d'abord le graphique précédent puis établit les nouvelles dimensions de la fenêtre graphique en fonction des nouvelles données à tracer. La commande `hold` permet de conserver une ancienne fenêtre graphique et d'y rajouter un nouveau tracé de lignes.

Exemple 11.2 Les commandes suivantes, qui permettent de tracer l'évolution d'une densité lognormale $f(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp^{-\frac{(\log x - \mu)^2}{2\sigma^2}}$ en fonction du paramètre σ, illustrant l'utilisation de la commande `hold`.

```
f='exp(-(log(x)-mu).^2/(2*sigma^2))./(x*sigma*sqrt(2*pi))'
n = 8;
mu = 1.8; s = linspace(1,0.1,n) ; x = linspace(0.1,15);
hold off, clg
sigma = s(1);
plot(x,eval(f),'r'), hold on
for i = 2:n
     sigma = s(i);
     plot(x,eval(f)), pause(1)
end
title(['Lognormale (mu=1.5)  sigma = ',num2str(s(1)),'  a  ', ...
num2str(s(n))]);
```

Pour forcer MATLAB à dessiner une fenêtre graphique qui représente un carré on exécute la commande `axis('square')`.

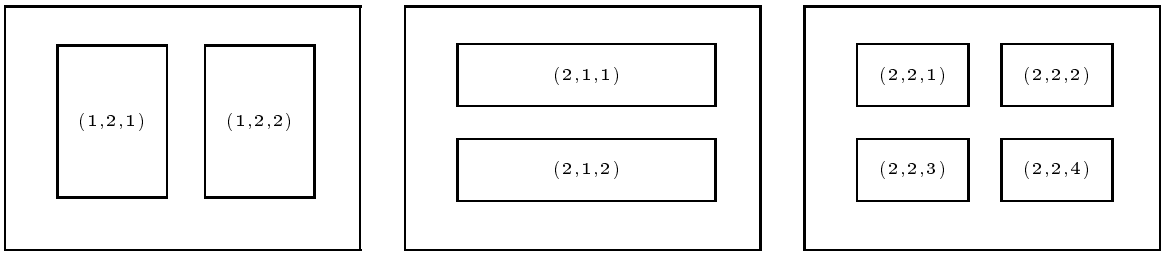
Partition de la fenêtre graphique

MATLAB dimensionne la fenêtre graphique par defaut de sorte qu'elle remplisse la fenêtre defini à l'écran ou qu'elle tienne sur une demi page lors de l'impression. Avec la commande `subplot`, il est possible de partitionner cet espace en n sous-fenêtres vericales et m sous-fenêtres horizontales. La commande

`subplot` doit précéder la commande `plot` comme montré dans l'exemple suivant:

```
subplot(n,m,q), plot(x,y)
```

Le paramètre q définit la position de la sous-fenêtre dans l'odre des lignes. La figure suivante montre quelques partitions avec les paramètres correspondants.



11.2 DIAGRAMMES PARTICULIERS

Les fonctions `hist`, `bar` et `stairs` permettent de dessiner des histogrammes, diagrammes en bâtons et diagrammes en escaliers.

Exemple 11.3 Illustration de quelques-unes des fonctionnalités des fonctions `bar`, `stairs` et `hist`.

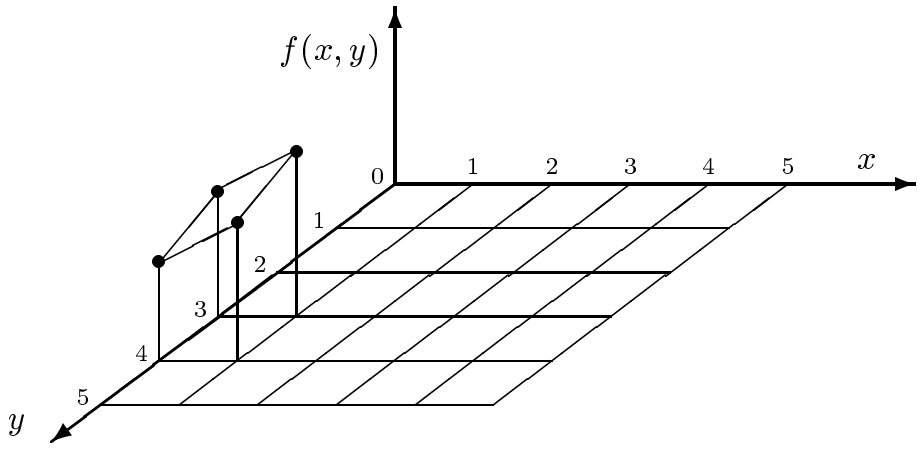
```
clg
y = randn(50,1);
subplot(2,2,1), bar(y)
x = 1:2:100;
subplot(2,2,2), bar(x,y)
[xb,yb] = bar(x,y);
subplot(2,2,3), plot(xb,yb,'.')      % equivalent avec bar(x,y)
x = 3:13; y = exp(sqrt(x));
subplot(2,2,4), stairs(y)

clg
y = randn(1000,1);
subplot(2,2,1), hist(y)     % 10 classes par defaut
subplot(2,2,2), hist(y,20)  % 20 classes
x = [-3 -1 0 1 3];
subplot(2,2,3), hist(y,x)
[n,x] = hist(y);
subplot(2,2,4), plot(x,n,'x')
```

11.3 GRAPHIQUES 3D ET LIGNES DE NIVEAUX

Pour produire le graphique d'une fonction à deux variables $z = f(x,y)$, on quadrille le domaine de définition, puis on fait correspondre aux noeuds du quadrillage une matrice Z dont les éléments correspondent à la valeur de la fonction en ses différents noeuds. Dans le plan qui correspond au domaine de définition, ces éléments représentent des hauteurs et la commande `mesh` relie ces points par des lignes.

Soit la fonction $z = x^a y^b$ que l'on désire représenter dans le domaine défini par $x \in [0,5]$ et $y \in [0,5]$. On choisit un quadrillage avec une largeur de maille d'une unité. La variable x prend donc les valeurs $x = [0\ 1\ 2\ 3\ 4\ 5]$ et la variable y les valeurs $y = [0\ 1\ 2\ 3\ 4\ 5]$.



Il s'agit maintenat de calculer la matrice Z dont les 36 éléments sont définis $Z_{ij} = f(x_i, y_j)$ pour $i,j = 1,\ldots,6$. On pourraît le faire avec deux boucles `for`, une pour l'indice i et l'autre pour l'indice j.

Cette expression peut cependant aussi être évaluée vectoriellement, ce qui est plus efficace. Pour ce faire, il faut définir deux matrices

$$X = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \end{bmatrix}$$

Ensuite on peut évaluer tous les éléments de la matrice Z avec l'expression

```
Z = X.^a .* Y.^b
```

Les matrices X et Y peuvent être construites à l'aide de la commande `meshgrid` et le maillage est obtenu à l'aide de la fonction `mesh`. Pour notre exemple on programme:

```
[X,Y] = meshgrid(0:5,0:5);
mesh(Z)
```

La fonction `view` permet la modification de la perspective de l'objet représenté. Par défaut on a une rotation horizontale de 37.5 dégrées dans le sens d'une montre (-37.5 azimut) et une élévation de 30 dégrées.

```
subplot(1,2,1), mesh(Z)
subplot(1,2,2), mesh(Z)
view([-10 40])
```

Exemple 11.4 Dessin de la fonction $f(x,y) = -(x^2+y-11)^2 - (x+y^2-7)^2$ en $\mathbb{R}^3$. Le domaine pour les variables x et y est defini comme $x \in [-5,5]$ et y $[-5,5]$.

```
clg
[X,Y] = meshgrid(linspace(-5,5,40),linspace(-5,5,40));
Z = -(X.^2 + Y - 11).^2 - (X + Y.^2 - 7).^2;
mesh(Z)
```

On obtient les lignes de niveaux de la même fonction pour des niveaux allant de 0 à -100 par pas de 8 avec la commande:

```
contour(Z,0:-8:-100)
```

La commande `surfc(Z)` produit le maillage et les lignes de contour.

Voici un exemple de création de quatre dessins differants de la même fonction. La commande `surfl` permet d'obtenir une surface lisse par interpolation des points ainsi que des ombres genéres par une source lunineuse.

```
subplot(2,2,1), mesh(Z)
subplot(2,2,2), contour(Z,0:-8:-100)
subplot(2,2,3), surfc(Z)
subplot(2,2,4), surfl(Z), shading interp; colormap(pink);
```

Exemple 11.5 Une autre façon de visualiser l'evolution de la densité lognormale de l'exemple 11.2.

```
[x sigma]=meshgrid(linspace(0.1,15,40),linspace(1,0.1,40));
mu=1.8;
f=exp(-(log(x)-mu).^2./(2*sigma.^2))./(x.*sigma*sqrt(2*pi));
mesh(sigma,x,f); title('mesh(sigma,x,f)'); figure(gcf); pause
waterfall(f); title('waterfall(f)'); figure(gcf); pause
surfc(f);     title('surfc(f)');     figure(gcf); pause
surfc(-f);    title('surfc(-f)');    figure(gcf); pause
surfl(f); shading interp; colormap(pink);
title('surfl(f); shading interp; colormap(pink);'); figure(gcf);
```