Chapitre 5

Les vecteurs et matrices

I. Vecteurs et matrices
- I.1. Les vecteurs
  - I.1.1. Addition et soustraction
  - I.1.2. Transposition
  - I.1.3. Opérations élément par élément
  - I.1.4. Génération de vecteurs
  - I.1.5. Opérations relationnelles sur les vecteurs
- I.2. Les matrices

II. Fonctions sur les vecteurs et matrices
- II.1. Quelques fonctions sur les matrices
- II.2. Concaténation
- II.3. Extraction d'une partie d'une matrice, extension d'une matrice
- II.4. Comparaison de matrices
- II.5. Typage des données numériques
- II.6. Transformations de vecteurs et matrices

III. Fonctions propres aux matrices
- III.1. Produit de matrices
- III.2. Inversion de matrices
- III.3. Division de matrices
- III.4. Exponentielle, logarithme et racine carrée d'une matrice
- III.5. Test du type des éléments d'une matrice

IV. Matrices particulières et spéciales

V. Factorisation et décomposition de matrices

VI. Matrices creuses et fonctions associées

VII. Applications
- VII.1. Moyenne et variance d'une série de mesures
- VII.2. Dérivée d'une fonction
- VII.3. Calcul d'intégrales
- VII.4. Résolution d'un système d'équations linéaires
- VII.5. Résolution d'un système sous-dimensionné ou indéterminé
- VII.6. Régression linéaire
- VII.7. Régression non linéaire

I. Vecteurs et matrices

L'élément de base pour MATLAB est une matrice à éléments complexes. Ainsi tout nombre réel est considéré comme une matrice à une ligne et une colonne dont le seul élément est à partie imaginaire nulle. Un vecteur n'est autre qu'une matrice à une ligne ou à une colonne. Les vecteurs servent aussi à représenter les polynômes et les chaînes de caractères. Les tableaux multidimensionnels sont des matrices concaténées selon certaines directions.

N. Martaj, M. Mokhtari, *MATLAB R2009, SIMULINK et STATEFLOW pour Ingénieurs, Chercheurs et Etudiants*, DOI 10.1007/978-3-642-11764-0_5, © Springer-Verlag Berlin Heidelberg 2010

I.1. Les vecteurs

I.1.1. Addition et soustraction

L'addition et la soustraction de vecteurs de mêmes dimensions se font élément par élément. Soit les vecteurs lignes x et y suivants :

```
>> x = [0 5 2];
```

```
>> y = [3 5 7];
```

Les opérations élémentaires sur ces vecteurs se font tout simplement comme suit :

```
>> x-y
ans =
    -3      0     -5
```

```
>> x+y
ans =
     3     10      9
```

Tant qu'on n'a pas affecté le résultat de l'opération à une variable, MATLAB crée la variable `ans` pour `answer` (réponse).

```
>> resultat = 2*x-y
resultat =
    -3      5     -3
```

Multiplier ou diviser un vecteur par un scalaire revient à diviser chaque élément du vecteur par ce scalaire.

```
>> 3*x-y/2
ans =
   -1.5000   12.5000    2.5000
```

I.1.2. Transposition

Pour réaliser certaines opérations vectorielles, on est amené à transformer un vecteur ligne en vecteur colonne et inversement.
La transposition d'un vecteur est réalisée en faisant suivre son nom par une apostrophe.

```
>> z = [2 5 -7 4];
```

```
>> zT = z'
zT =
     2
     5
    -7
     4
```

La norme d'un vecteur z est égale à la racine carrée de la somme des carrés de ses éléments. Ceci peut s'obtenir à partir du produit scalaire du vecteur z par lui-même.

Norme d'un vecteur

```
>> norme_z = sqrt(z*z') % sqrt(N)
norme_z =
    9.6954
```

On peut directement utiliser la fonction *norm*.

```
>> norm(z)
ans =
    9.6954
```

Pour tester la taille d'un vecteur, sa nature ligne ou colonne, nous pouvons utiliser les commandes `length` (longueur) et *size* (taille).

```
>> size(z)
ans =
     1     4
```

```
>> length(z)
ans =
     4
```

z est un vecteur ligne de 4 éléments.

La fonction `numel` donne directement le nombre d'éléments.

```
>> numel(z)
ans =
     4
```

Pour afficher un vecteur, il suffit de faire appel à son nom ou utiliser la commande `disp`.

```
>> z
z =
     2     5    -7     4

>> disp(z)
     2     5    -7     4
```

I.1.3. Opérations élément par élément

Les opérateurs sont précédés du signe "point" lorsqu'on veut réaliser des opérations entre les éléments de deux vecteurs, pris un à un.

```
>> x = [-1 4 3]; y = [5 4 -3];
```

```
>> z = x.*y
z =
    -5    16    -9

>> x./y
ans =
   -0.2000    1.0000   -1.0000

>> x.^2
ans =
     1    16     9
```

I.1.4. Génération de vecteurs

La commande `linspace` permet de générer un ensemble de `n` éléments (vecteur ligne) en spécifiant la première et la dernière valeur. La commande suivante génère un vecteur de `5` composantes équidistantes allant de `0` à `1`.

```
>>  t = linspace(0,1,5)
t =
         0    0.2500    0.5000    0.7500    1.0000
```

On peut aussi spécifier la première et la dernière valeur ainsi que le pas. Par exemple, on crée le vecteur w dont ces valeurs sont respectivement de 0 et 4.

```
>>  w = 0:0.5:4
w =
    0      0.5000      1.0000      1.5000      2.0000      2.5000
3.0000    3.5000    4.0000
```

Si on ne spécifie pas la valeur du pas, il est choisi, par défaut, égal 1.

```
>> w = 0:4
w =
     0     1     2     3     4
```

Si on omet de spécifier le pas, la commande `linspace(x1,x2)` génère `100` valeurs également espacées entre `x1` et `x2`.

```
>> length(linspace(0,5)) % length : longueur du vecteur
ans =
     100
```

Si on désire un espacement logarithmique, on dispose de la commande :

`logspace (d1,d2,N)`

qui génère `N` valeurs espacées d'un pas logarithmique entre 10^{d1} et 10^{d2}, si `N` est omis sa valeur par défaut est `50`.

```
>> t = logspace(1,2.5,5)
t =
   10.0000   23.7137   56.2341  133.3521  316.2278
```

I.1.5. Opérations relationnelles sur les vecteurs

La commande `isvector` retourne la valeur 1 si l'argument est un vecteur (ligne ou colonne) et fausse (valeur 0) dans le cas contraire (matrice). Les dimensions doivent être 1 x n ou n x 1 avec n>=0.

```
>> isvector(z)
ans =
     1
```

On vérifie que même si z possède 2 dimensions, l'une d'elles vaut toujours 1 comme le donne la commande `size`.

```
>> ndims(z)
ans =
     2

>> size(z)
ans =
     1     3
```

Pour déterminer si une donnée est un scalaire, nous avons la commande `isscalar`.

```
>> isscalar(z)
ans =
     0
```

```
>> x=5 ;
>> isscalar(x)
ans =
     1
```

La variable x=5 est bien un scalaire et pas le vecteur z.

Les vecteurs peuvent contenir des valeurs infinies et des valeurs complexes. Ainsi le vecteur w contient les valeurs 3+2i et l'infini.

```
>> w = [3+2i inf]
w =
   3.0000 + 2.0000i      Inf
```

```
>> isfinite(w)
ans =
      1   0
```

La première composante est bien une valeur finie et pas la deuxième.

```
>> isreal(w)
ans =
     0
```

```
>> isinf(w)
ans =
     0   1
```

La deuxième composante est infinie et pas la première.

Un vecteur peut contenir une valeur vide, symbolisée par la valeur [].

```
>> x = [];
>> isempty(x)
ans =
     1
```

Le vecteur x est bien vide.

I.2. Les matrices

Le tableau à 2 dimensions est l'élément de base de MATLAB. Un vecteur n'est autre qu'une matrice à une ligne ou à une colonne.
Un simple scalaire est vu comme une matrice à 1 ligne et 1 colonne.

```
>> x=5.3;
>> size(x)
ans =
     1     1
```

Une matrice peut être écrite comme une suite de vecteurs lignes, séparés par des points-virgules qui symbolisent des sauts de ligne.

```
>> x = [0 5; 3 5; 6 1]
x =
     0     5
     3     5
     6     1
```

```
>> size(x)
ans =
     3     2
```

```
>> [m, n]=size(x)
m =
     3
n =
     2
```

La variable x est une matrice à 3 lignes et 2 colonnes.

```
>> length(x)   % plus grande taille de la matrice x
ans =
     3
```

Comme pour les vecteurs, on dispose de plusieurs modes d'affichage de matrices : un appel direct de la variable matricielle ou l'utilisation de la commande disp, qui sert aussi à l'affichage d'une chaîne de caractères.

```
>> x
x =
     0     5     3
     1     2     6

>> disp(x)
     x =
     0     5     3
     1     2     6
```

II. Fonctions sur les vecteurs et matrices

II.1. Quelques fonctions sur les matrices

MATLAB dispose de fonctions qui opèrent directement sur les vecteurs ou les colonnes d'une matrice. Parmi celles-ci, on peut citer :

- mean : valeur moyenne,
- std : écart type,
- sum : somme,
- cumsum : somme cumulée,
- cumprod : produit cumulé,
- min : valeur minimale,
- max : valeur maximale,
- diff : différence des éléments successifs.
- prod : produit,
- sort : ordre croissant ou décroissant des éléments du vecteur.

Considérons la matrice rectangulaire X suivante :

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

X =
     1     2     3
     4     5     6
```

La moyenne et l'écart type de chaque colonne sont :

```
>> moyenne = mean(X)
moyenne =
    2.5000    3.5000    4.5000
```

```
>> ecart_type = std(X)
ecart_type =
    2.1213    2.1213    2.1213
```

Les fonctions s'appliquent aux éléments des colonnes d'une matrice

Si l'on désire calculer, par exemple, la moyenne, la somme, l'écart type de l'ensemble des éléments de la matrice, il faut, au préalable, transformer cette dernière en un vecteur colonne par l'opérateur "`:`". Cette même transformation doit être réalisée pour l'utilisation des fonctions `sum`, `min` et `max`. Nous pouvons aussi, dans certains cas, appliquer deux fois la même fonction.

```
>> X(:)'
ans =
     1     4     2     5     3     6
```

```
>> ec_type_X = std(X(:))
ec_type_X =
    1.8708
```

```
>> moy_X = mean(X(:))
moy_X =
    3.5000
```

Cette valeur peut-être obtenue directement en doublant la fonction `mean` :

```
>> mean(mean(X)
ans =
    3.5000
```

Produit scalaire de 2 vecteurs

```
>> c = [1 4 3]; d = [5 2 1];
>> dot(c,d)
ans =
    16
```

Quelque soit le type ligne ou colonne de l'un ou des deux vecteurs le résultat ne change pas.

```
>> dot(c',d)
ans =
    16
```

Valeurs minimale et maximale de chaque colonne de X

```
>> mn = min(X)
mn =
     1     2     3
>> mx = max(X)
mx =
     4     5     6
```

Comme précédemment, le minimum et le maximum de toutes les valeurs de la matrice sont calculés des deux façons suivantes :

```
>> min_max1 = [min(X(:)) max(X(:))]
min_max1 =
                 1     6
```

```
>> min_max2=[min(min(X)) max(max(X))]
min_max2 =
              1     6
```

Somme des éléments de chaque colonne de X

```
>> som = sum(X)
som =
     5     7     9
```

Somme des éléments de X

```
>> somX = sum(sum(X))
somX =
    21
```

L'application des fonctions `min`, `max` et `sum` au vecteur `X(:)` donne les mêmes résultats.

Il en est de même pour la somme qui s'applique d'abord à tous les éléments de chaque colonne. La somme de tous les éléments de la matrice se fait comme précédemment. L'expression relationnelle d'égalité suivante, valant 1, démontre l'égalité des deux expressions :

```
>> sum(X(:))==sum(sum(X))
ans =
     1
```

La fonction `prod` donne le produit des éléments d'un vecteur. On peut s'en servir, par exemple, pour calculer directement la factorielle d'un entier `n`.

`n ! = n (n-1) ... 1`

Factorielle de 5

```
>> n = 5;
>> fact_5 = prod(1:n)
```

```
fact_5 =
   120
```

Ce même calcul peut être réalisé en utilisant la récursivité de la factorielle n ! = n (n-1) ! (voir chapitre « Programmation »).

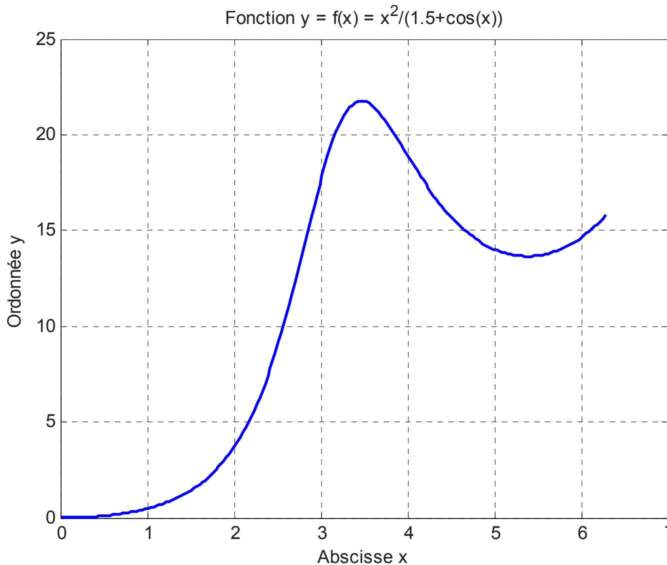
Une majeure partie des fonctions MATLAB est prévue pour accepter directement comme arguments d'appel, des vecteurs ou matrices. Les arguments de retour peuvent être alors, des vecteurs ou des matrices de dimensions analogues aux arguments d'entrée. Ceci est illustré par l'exemple qui suit.

fichier vect_poly.m

```
% domaine de variation, vecteur x
x = 0:pi/100:2*pi;

% vecteur y de la fonction
y = x.^2./(1.5+cos(x)); % argument d'appel : vecteur x

% tracé de la courbe de f
plot(x,y), grid
title('Fonction y = f(x) = x^2/(1.5+cos(x))')
xlabel('Abscisse x')
ylabel('Ordonnée y')
```



On précède l'opérateur mathématique par le point pour effectuer des opérations élément par élément entre vecteurs.

Produit cumulé des éléments d'un vecteur

Considérons le vecteur x suivant :

```
>> x = [2  5  3]
x =
     2     5     3

>> cumprod(x)
ans =
     2    10    30
```

Le résultat du produit obtenu par deux éléments successifs est multiplié à son tour par le suivant et ainsi de suite jusqu'au dernier.

On peut aussi s'en servir pour le calcul de la factorielle d'un nombre entier.

```
>> N=5;
>> ProdCumul=cumprod(1 :5)
>> fact_N=ProdCumul(length(ProdCumul))

ProdCumul =
     1     2     6    24   120

fact_N =
   120
```

II.2. Concaténation

Une matrice peut être construite par la concaténation d'autres matrices. La concaténation doit respecter la condition sur les matrices qui doivent être de mêmes tailles.

Soit les matrices A et B suivantes :

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

On crée la matrice C en concaténant B à droite de A, comme suit :

```
>> C = [A B]
C =
     1     5     3     0     6     7
     5     2     6     8     4     3
```

La concaténation peut se faire aussi verticalement grâce à un saut de ligne.

```
>> D = [A;B]
D =
```

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

La commande `cat(dims, A, B)` permet de concaténer deux ou plusieurs matrices selon la dimension spécifiée dans l'entier *dims* = 1 (verticalement), 2 (horizontalement) et 3 (par pages, voir tableaux multidimensionnels).

```
>> C = cat(1,A,B)
C =
     1     5     3
     5     2     6
     0     6     7
     8     4     3
```

Les concaténations, horizontale et verticale, peuvent être effectuées par les commandes `horzcat` et `vertcat`.

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

```
>> C=vertcat(A,B)
C =
     1     5     3
     5     2     6
     0     6     7
     8     4     3
```

```
>> C=horzcat(A,B)
C =
     1     5     3     0     6     7
     5     2     6     8     4     3
```

II.3. Extraction d'une partie d'une matrice, extension d'une matrice

A partir d'une matrice, on peut extraire une autre matrice, un vecteur ou l'un de ses éléments.

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

L'élément de la deuxième ligne et de la troisième colonne peut être récupéré en écrivant :

```
>> x(2,3)
```

```
ans =
     5
```

L'instruction suivante permet de récupérer une partie de la matrice `x`. Cette partie sera composée de toutes les lignes de `x` (représentées par le signe `:`) et des colonnes `2` à `3`.

```
>> x1 = x(:,2:3)
x1 =
     1     2
     4     5
```

Cette matrice peut aussi être obtenue en supprimant, de la matrice `x`, la première colonne.

Remplacement de la première colonne par une colonne vide

```
>> x(:,1) = []
x =
     1     2
     4     5
```

Extraction de la deuxième colonne

```
>> x(:,2)
ans =
     1
     4
```

Extraction de la deuxième ligne

```
>> x(2,:)
ans =
     3     4     5
```

II.4. Comparaison de matrices

La comparaison de deux vecteurs ou matrices de mêmes dimensions donne un vecteur ou matrice de dimensions analogues, composés de `0` et de `1`. Cette opération se fait élément par élément. Le résultat de la comparaison de deux éléments est `1` lorsque la condition est vérifiée et `0` dans le cas contraire.

Considérons les deux matrices `A` et `B` suivantes :

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

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

On remarque que seul l'élément `A(2,1)` ne satisfait pas à la condition donnée.

Soit la matrice C suivante :

```
>> C = [0 2; 4 6];
```

La commande `isequal (A, C)` donne 1 si les matrices A et C sont identiques et 0 autrement.

```
>> isequal(A,C)
ans =
     0
```

Le test d'égalité donne 1 pour chaque élément identique et 0 dans le cas contraire. Dans le cas des matrices A et C, seuls les éléments (1,2) et (2,1) sont égaux, comme le montre le test suivant :

```
>> A==C
ans =
     0     1
     1     0
```

Pour vérifier si au moins un élément d'un vecteur ou tous les éléments de celui-ci sont nuls, on utilisera respectivement les commandes `any` et `all`. Ces commandes s'appliquent aussi aux matrices.

L'application de la commande `all` à un vecteur retourne `1` si tous ses éléments sont non nuls et `0` dans le cas contraire. Dans le cas d'une matrice, elle retourne un vecteur ligne dont les composantes sont les résultats de cette commande appliquée à chacune des colonnes.

Pour vérifier si une matrice est symétrique, on la comparera à sa transposée.

```
>> isequal(A,A')
ans =
     0
```

Le résultat de l'égalité étant égal à 0, ceci implique que la matrice `A` n'est pas symétrique.

```
>> all(A == A')
ans =
     0     0
```

Ce test peut être largement simplifié en appliquant deux fois la commande `all`.

```
>> all(all(A == A'))
ans =
     0
```

Tout ce qui a été dit sur la commande `all` reste valable pour la commande `any`.
Par exemple, la réponse à la question : "les matrices `A` et `B` ont-elles des éléments identiques ?", sera donnée par la commande suivante :

```
>> any(any(A == B))
ans =
     0
```

La comparaison entre deux matrices de mêmes dimensions peut se faire en utilisant des opérateurs logiques relationnels classiques.

Opérateurs relationnels

Les opérateurs suivants, qui comparent la matrice A à la matrice B agissent élément par élément. Le résultat est une matrice de mêmes dimensions que A (ou B) contenant 1 à l'indice où la condition est vérifiée et 0 ailleurs.

A < B,   A > B,   A <= B,   A >= B,   A == B,   A ~= B

Considérons les matrices A et B suivantes :

```
>> A=[1 5; 3 2]
 A =
     1     5
     3     2

>> B=[0 6; 2 7]
B =
     0     6
     2     7
```

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

Il n'y a que les éléments (1,1) et (2,1) de la matrice A qui sont supérieurs ou égaux à ceux de B.

L'opérateur ~ est l'opérateur de la négation. L'instruction suivante qui recherche les éléments de A qui ne sont pas strictement plus petits que ceux de B donnera les mêmes résultats que précédemment.

```
>> ~(A<B)
ans =
     1     0
     1     0
```

Opérateurs logiques

Les symboles &, |, et ~ sont les opérateurs logiques correspondent aux portes logiques AND, OR et NOT.
Nous disposons aussi des fonctions or, not et xor pour implémenter le OU, le NOT et le « ou exclusif ».

Avec les matrices A et B précédentes nous avons :

```
>> A|B ;
ans =
     1     1
     1     1
```

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

Les valeurs de A et B non nulles sont considérées comme une valeur logique 1.

```
>> xor(A,B)
ans =
     1     0
     0     0
```

```
>> not(A&B)
ans =
     1     0
     0     0
```

Ces 2 commandes donnent le même résultat.

```
>> isequal(xor(A,B), not(A&B))
ans =
     1
```

Les opérateurs && et || sont dénommés ET et OU courts-circuits. Ils ont l'avantage d'éviter l'évaluation de toute une expression si l'opérande de gauche est considéré comme faux.

Considérons l'expression logique suivante. Le premier opérande est choisi volontairement à la valeur fausse. On utilisera successivement le & et le court-circuit && et on déterminera le temps de calcul mis pour évaluer la même expression logique.

```
>> clc
>> b=0;
>> tic;
>> x = (b ~= 0) & (randn(1000)> 1);
>> toc
Elapsed time is 0.060768 seconds..
```

Lorsqu'on utilise le ET court-circuit, le fait que b soit faux, l'expression `(randn(1000)>1)` n'est pas évaluée, ce qui diminue considérablement le temps de calcul.

```
>> tic;
>> x = (b ~= 0) && (randn(1000)> 1);
>> toc
Elapsed time is 0.001641 seconds.
```

Le temps de calcul est d'environ le tiers avec le && qu'avec le ET simple du &.

II.5. Typage des données numériques

Les mêmes valeurs numériques peuvent être forcées dans le type entier signé ou non signé, codé sur 8, 16, 32 ou 64 bits à point fixe ou réelles à virgule flottante.

La matrice A, bien que ne contenant que des entiers, est du type numérique, mais pas entier :

```
>> [isnumeric(A) isinteger(A)]
ans =
     1     0
```

Lorsqu'on force le type entier non signé sur 8 bits par `uint8`, elle devient du type entier, toujours numérique mais pas flottant.

```
>> C=uint8(A)
>> [isnumeric(C) isinteger(C) isfloat(C)]
ans =
     1     1     0
```

Pour déterminer le type d'un objet, on utilise la fonction `class`. On peut aussi forcer le type par la fonction `cast`.

```
>> a = int8(5);
>> b = cast(a,'uint8');
>> class(a)
ans =
int8
>> class(b)
ans =
uint8
```

Un nombre entier signé, codé sur n bits possède des valeurs entre $-2^n/2$ et $2^n/2 -1$ et le non signé entre 0 et $2^n -1$.

```
>> intmin(class(a))
ans =
 -128
```

```
>> intmax(class(a))
ans =
  127
```

II.6. Transformations de vecteurs et matrices

Les commandes `fliplr` et `flipud` réalisent respectivement un retournement de gauche à droite et de haut en bas du vecteur (ou de la matrice) donné en argument.

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

```
>> x_lr = fliplr(x)
x_lr =
     3     2     1
     6     5     4
```

```
>> x_ud = flipud(x)
x_ud =
     4     5     6
     1     2     3
```

La commande `rot90` réalise une rotation de `90°` d'un vecteur ou matrice. Le résultat obtenu, différent de la transposée, correspond aux applications successives de la transposition et de la commande `flipud`.

```
>> x_90 = rot90(x)
x_90 =
     3     6
     2     5
     1     4
```

```
>> x'
ans =
     1     4
     2     5
     3     6
```

On vérifie aisément que la commande `flipud`, appliquée à la transposée de x correspond à sa rotation par `rot90`.

```
>> all(all(flipud(x') == rot90(x)))
ans =
     1
```

```
>> isequal(flipud(x'), rot90(x))
ans =
     1
```

Les résultats de `flipud(x')` et `rot90`(x) sont identiques.

III. Fonctions propres aux matrices

MATLAB possède toutes les fonctions de calcul matriciel, qui font de lui, en plus de ses possibilités graphiques de très haut niveau, un langage de calcul numérique qui permet de résoudre, avec facilité, beaucoup de problèmes rencontrés en ingénierie.

III.1. Produit de matrices

Le produit d'une matrice de dimensions `(m,n)` par une matrice de dimensions `(p,q)` n'est possible que si `n = p`, le résultat donne une matrice de dimensions `(m,q)`.

```
>> A = [1 2; 3 4];
>> B = [1 2 3; 4 5 8];
```

```
>> A*B
ans =
     9    12    19
    19    26    41
```

Si la condition précédente n'est pas réalisée, MATLAB fournit un message annonçant une erreur dans les dimensions des matrices utilisées.

```
>> B*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

Le produit de Hadamard, ou produit élément par élément de 2 matrices de mêmes dimensions, s'obtient par l'opérateur `'.*'`.

```
>> A*A
ans =
     7    10
    15    22
```

```
>> A.*A
ans =
     1     4
     9    16
```

III.2. Inversion de matrices

Une matrice carrée est inversible si son rang est égal à sa dimension.

```
>> x = [1 2 ; 5 3];
```

Inversion de la matrice x

```
>> inv(x)
ans =
   -0.4286    0.2857
    0.7143   -0.1429
```

Rang de la matrice x

```
>> rank(x)
ans =
     2
```

Si la matrice possède un certain nombre de lignes ou de colonnes linéairement dépendantes, elle n'est pas inversible (rang plus faible que l'ordre).
MATLAB fournit dans ce cas, un message d'erreur, signalant que la matrice est singulière ou mal conditionnée.

```
>> x = [1 2; 3 6]
x =
     1     2
     3     6
```

```
>> rang = rank(x)
rang =
     1

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.551115e-018
```

```
>> x_inv = inv(x)
x_inv =
1.0e+016 *
   -1.8014    0.6005
    0.9007   -0.3002
```

Dans ce cas, si on effectue le produit de `x` par son inverse, on ne retrouve pas la matrice identité.

```
>> x_inv*x
ans =
   -1.0000   -2.0000
    0.5000    1.0000
```

III.3. Division de matrices

L'opérateur division "`/`" précédé du point, "`./`", permet une division élément par élément de deux matrices.

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

```
>> A_sur_B = A./B
A_sur_B =
    0.5000    1.0000
    1.0000    2.0000
```

On peut aussi utiliser l'opérateur slash "`/`", dans ce cas `A/B` correspond à `A*inv(B)`. Ceci peut être vérifié par l'instruction suivante :

```
>> isequal(A/B, A*inv(B)))
ans =
     1
```

L'opérateur antislash "`\`" réalise une division à gauche, `A\B = inv(A)*B`.

```
>> all(all(A\B == inv(A)*B))
ans =
     1
```

III.4. Exponentielle, logarithme et racine carrée d'une matrice

MATLAB dispose de 4 fonctions pour le calcul de l'exponentielle d'une matrice :

`expm` : à base des valeurs et vecteurs propres de la matrice,

`expm1` : à base de l'approximation de Pade,

`expm2` : à base des séries de Taylor,

`expm3` : à base des valeurs et vecteurs propres de la matrice.

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

```
>> y = expm(x)
y =
    2.7183    2.3504
         0    0.3679
```

Les différentes fonctions précédentes donnent le même résultat.
La fonction `exp` appliquée à une matrice, calcule l'exponentielle, élément par élément.

```
>> z = exp(x)
z =
    2.7183    7.3891
    1.0000    0.3679
```

Matrice logarithme

```
>> w = logm(x)
```

```
w =
        0                  0 - 3.1416i
        0                  0 + 3.1416i
```

Les éléments de la matrice résultat sont complexes si les valeurs propres de la matrice d'origine sont négatives, ce qui est le cas de la matrice `x`.

```
>> eig(x) % valeurs propres de x
ans =
     1
    -1
```

Logarithme des éléments d'une matrice

La fonction `log` appliquée à une matrice, calcule le logarithme népérien de chacun de ses éléments.

```
>> log(x)
Warning: Log of zero
ans =
        0                  0.6931
     -Inf                  0 + 3.1416i
```

```
>> logm(expm(x))  % on retrouve la matrice d'origine
ans =
    1.0000    2.0000
         0   -1.0000
```

On obtient le même résultat avec `expm(logm(x))`.

Matrice racine carrée

La matrice racine carrée `rA` d'une matrice `A` est définie par `rA*rA = A` et s'obtient par la fonction `sqrtm`.

```
>> rX = sqrtm(x)
rX =
   1.0000             1.0000 - 1.0000i
        0                  0 + 1.0000i
>> all(all(sqrtm(x)^2 == x))
ans =
     1
```

Pour obtenir les racines carrées des éléments d'une matrice, on utilisera la fonction `sqrt`.

```
>> sqrt(x)
ans =
   1.0000                  1.4142
   0                       0 + 1.0000i
```

D'une manière générale, on peut évaluer n'importe quelle fonction matricielle par la commande `funm` dont la syntaxe est :

```
fx = funm('f',x)
```

```
>> x = [1 -1;0 2];
>> fx = funm('exp',x)

fx =
    2.7183   -4.6708
         0    7.3891
```

On obtient le même résultat si on applique directement la fonction `expm` à la matrice x par :

```
>> expm(x)
ans =
    2.7183    2.3504
         0    0.3679
```

III.5. Test du type des éléments d'une matrice

Comme nous le verrons dans les chapitres correspondants, une matrice peut-être constituée d'éléments numériques ou chaînes de caractères. Seule une structure peut contenir divers types d'éléments. Il existe dans MATLAB, des commandes pour tester le type des éléments d'une matrice.

Soit la matrice `x` :

```
>> x=[100 1/0 0/0 ; inf/0 5 0/inf]
x =
   100   Inf   NaN
   Inf     5     0
```

Les divisions 1/0 et Inf/0 donnent la valeur Inf qui symbolise l'infini tandis que 0/0 produit la valeur `NaN` qui n'est pas un nombre.

```
>> isnan(x)
ans =
     0     0     1
     0     0     0
```

Il n'y a que l'élément (1,3) provenant de la division 0/0 qui ne soit pas un nombre, donc du type `NaN`.

Pour déterminer l'existence d'une valeur infinie, nous utilisons la commande `isinf`.

```
>> isinf(x)
```

```
ans =
     0     1     0
     1     0     0
```

Seuls les éléments (1,2) et (2,1) sont infinis. Le résultat inverse est donné par la commande `isfinite`.

```
>> isfinite(x)
ans =
     1     0     0
     0     1     1
```

Nous obtenons une matrice de mêmes dimensions, avec des 1 là où l'élément est fini et 0 ailleurs. Dans le cas de cette matrice x, les éléments `Nan` ou `Inf` ne sont pas finis.

```
>> isnumeric(x)
ans =
     1
```

La commande `isnumeric` teste uniquement si les éléments d'une matrice sont de type numérique ou chaîne de caractères.
Bien que la matrice x contienne des valeurs infinies ou des éléments `NaN` (`Not a number`), ils restent néanmoins, de type numérique. Comme la matrice ne peut contenir qu'un seul type d'éléments, la réponse est 1 dans le cas numérique et 0 dans l'autre cas.

Nous pouvons transformer une matrice sous forme de chaînes de caractères par la commande `mat2str`. Prenons le cas de la même matrice x.

```
>> x_chaine = mat2str(x)
x_chaine =
[100 Inf NaN;Inf 5 0]
```

La matrice, quelles que soient ses dimensions, est transformée en une seule chaîne de caractères.

```
>> size(x_chaine)
ans =
     1    21
```

La chaîne ainsi obtenue possède 21 caractères.
Les caractères « `NaN` » correspondent aux éléments allant du $9^{ème}$ au $12^{ème}$ caractère de la chaîne.

```
>> x_chaine(9:12)
ans =
 NaN
```

La commande `isnumeric` donne 0, la valeur 100 est vue comme une chaîne de caractères.

```
>> isnumeric(x_chaine)
ans =
     0
```

Pour tester le type chaîne de caractères, nous disposons de la commande `isstr`.

```
>> isstr(x_chaine)
ans =
     1
```

Pour déterminer le type « chaîne de caractères >>, nous utilisons la commande `ischar`.

```
>> ischar(x_chaine)
ans =
     1
```

Les commandes `isfloat` et `isinteger`, testent respectivement les types à virgule flottante et le type entier.

Tableau logique (Logical Array)

Il existe un autre type de tableau : le tableau logique (`logical Array`) dont on peut se servir pour indexer un autre tableau.
Le type `logical` qui n'accepte que les valeurs 0 et 1 est un type particulier dont on se sert pour indexer un tableau pour la recherche.

Soit la matrice x suivante :

```
>> x=[1 2 6; 8 4 3]
x =
     1     2     6
     8     4     3
```

Si l'on veut récupérer les éléments (1,1), (2,1) et (2,3) on se servira, pour indexation, de la matrice index suivante.

```
>> index=[1 0 0; 1 0 1]
index =
     1     0     0
     1     0     1
```

Bien que ne comportant que les valeurs 0 et 1, cette matrice doit être convertie en type logique par la commande `logical` pour servir à l'indexation.

```
>> x(index)
??? Subscript indices must either be real positive integers
or logicals.
```

```
>> index=logical(index)

index =
     1     0     0
     1     0     1
```

Bien qu’ayant les valeurs 0 et 1, le type de cette matrice n’est pas numérique.

```
>> isnumeric(index)

ans =
     0
```

Le type logique des éléments de cette matrice est déterminé par la commande `islogical`.

```
>> islogical(index)

ans =
     1
```

Les valeurs de la matrice x, indexées par la matrice index sont :

```
>> x(index)

ans =
     1
     8
     3
```

Nous retrouvons bien les valeurs des index spécifiés.

IV. Matrices particulières et spéciales

Certaines matrices reviennent souvent dans les calculs scientifiques. MATLAB offre la possibilité de générer ce type de matrices spéciales telles que la matrice identité, le carré magique, la matrice de Pascal, la matrice de Hadamard, etc.

Pour les matrices rectangulaires, on précisera le nombre de lignes et de colonnes et pour les matrices carrées, on spécifiera uniquement l'ordre.

Matrice identité

Matrice identité d'ordre 3

```
>> identite = eye(3)

identite =
     1     0     0
     0     1     0
     0     0     1
```

Matrice nulle

Matrice nulle rectangulaire

```
>> zero = zeros(2,3)
zero =
     0     0     0
     0     0     0
```

Matrice unité

Matrice unité rectangulaire

```
>> un = ones(2,3)
un =
     1     1     1
     1     1     1
```

Matrices aléatoires

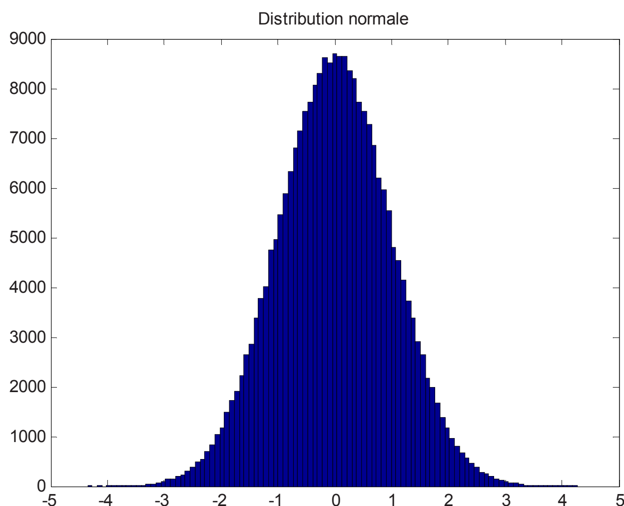
On peut générer des matrices aléatoires dont les éléments sont distribués normalement avec une moyenne nulle et une variance unité à l'aide de la commande `randn(m,n)`. Pour une distribution uniforme, on utilisera la commande `rand(m,n)`. Les paramètres `m` et `n` désignent respectivement le nombre de lignes et de colonnes. Les instructions suivantes génèrent 2 matrices aléatoires, `A_uni` et `A_norm` dont les éléments sont respectivement distribués uniformément et normalement.

```
>> n = 500; A_normale = randn(n); A_uniforme = rand(n);
```

Tracé des histogrammes des distributions obtenues

Distribution normale

```
>> n_classes = 100; hist(A_normale(:),n_classes)
>> title('Distribution normale')
```



On vérifie bien que la moyenne de la distribution est nulle et que sa variance est égale à 1.

```
>> mean(A_normale(:))
ans =
   -0.0041
```

```
>> std(A_normale(:))^2
ans =
    1.0004
```

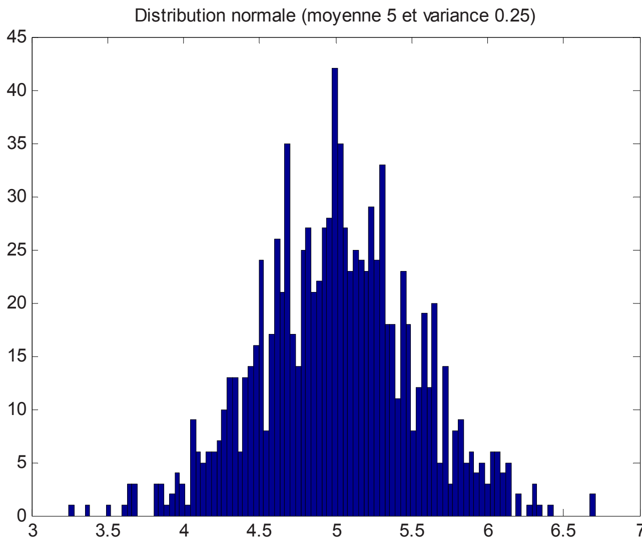
Si l'on désire une distribution normale de moyenne m et de variance σ^2, on opérera la transformation suivante :

```
A_norm2 = A_norm*σ + m
```

Exemple :

Distribution normale de 1000 éléments de moyenne 5 et de variance 4

```
>> A_norm2 = randn(1,1000)*0.5+5;
>> n_classes = 100;
>> hist(A_norm2,n_classes)
>> title('Distribution normale de moyenne 5, variance 0.25')
```



Distribution uniforme

La commande rand donne une distribution uniforme de valeurs entre 0 et 1 avec une moyenne de 1/2 et une variance de 1/12.

```
>> mean(A_uni(:))
ans =
    0.4999
```

```
>> std(A_uni(:))^2
ans =
    0.0827
```

Si l'on désire une distribution uniforme sur un intervalle [a, b], on opérera la transformation suivante :

```
A_uni2 = (b-a)*A_uni + a
```

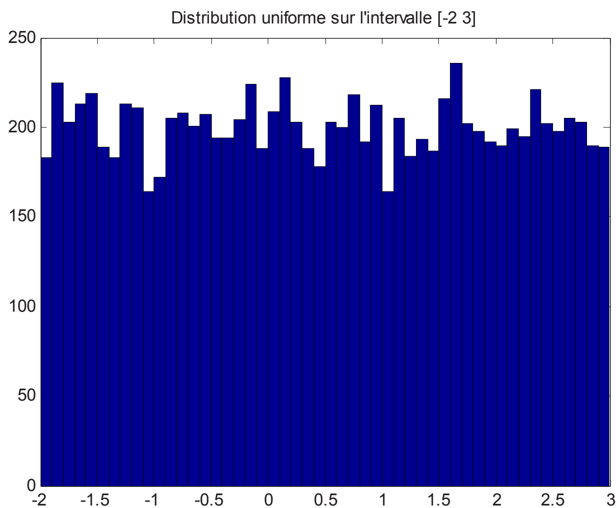
Exemple : distribution uniforme de 10000 éléments sur l'intervalle [-2 3]

```
>> a = -2; b = 3;
>> A_uni2 = (b-a)*rand(1,10000)+a;
>> n_classes = 50;
>> hist(A_uni2,n_classes)
>> title('Distribution uniforme sur l''intervalle [-2 3]')
```

```
>> mean(A_uni2)

ans =
    0.5169

>> std(A_uni2)^2
ans =
    2.0804
```



Carré magique

La commande `magic(n)` crée une matrice carrée d'ordre `n` dont les éléments prennent les valeurs allant de `1` à n^2. La somme des éléments de chaque ligne ou de chaque colonne donne le même nombre.

```
>> magic4 = magic(4)
magic4 =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Sommes des colonnes

```
>> sum(magic4)
ans =
    34    34    34    34
```

Sommes des lignes

```
>> sum(magic4')
ans =
    34    34    34    34
```

Matrice de Pascal

C'est une matrice d'entiers naturels, définie positive, symétrique, construite à partir du triangle de Pascal.

Matrice de Pascal carrée d'ordre 4

```
>> pasc_mat1 = pascal(4)
pasc_mat1 =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

```
>> inv(pasc_mat1)
ans =
    4.0000   -6.0000    4.0000   -1.0000
   -6.0000   14.0000  -11.0000    3.0000
    4.0000  -11.0000   10.0000   -3.0000
   -1.0000    3.0000   -3.0000    1.0000
```

Son inverse est aussi à éléments entiers.
La commande `pascal(n,1)` crée une matrice carrée d'ordre `n`, triangulaire inférieure et identique à son inverse (matrice involutive).

```
>> pasc_41 = pascal(4,1)
pasc_41 =
     1     0     0     0
     1    -1     0     0
     1    -2     1     0
     1    -3     3    -1
```

Matrice de Hadamard

Une matrice `H` de Hadamard d'ordre `n` est une matrice dont les éléments sont égaux à `1` ou `-1`. Le produit par sa transposée donne une matrice égale à `n` fois la matrice identité.

```
>> hadam4 = hadamard(4)
hadam4 =
     1     1     1     1
     1    -1     1    -1
     1     1    -1    -1
     1    -1    -1     1
```

```
>> hadam4'*hadam4
ans =
     4     0     0     0
     0     4     0     0
     0     0     4     0
     0     0     0     4
```

Matrice Compagnon

Si `p` désigne un polynôme de degré `n`, la commande `compan(p)` crée une matrice dite compagnon d'ordre `n-1`, dont `p` est le polynôme caractéristique (les racines de `p` sont égales à ses valeurs propres).

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

```
>> comp = compan(p)
comp =
    -2     1
     1     0
```

On vérifie bien l'égalité des racines de `p` et des valeurs propres de la matrice compagnon associée.

```
>> eig(comp) == roots(p)

ans =
     1
     1
```

D'autres commandes telles que `hankel`, `hilb`, `toeplitz`, `vander`, etc. permettent respectivement la génération des matrices de `Hankel`, `Hilbert`, `Toeplitz` et de `Vandermonde`.

V. Factorisation et décomposition de matrices

Factorisation de Cholesky

La factorisation de Cholesky d'une matrice `A` définie positive consiste en une décomposition de type :

`A = B'*B`

`B` : matrice triangulaire inférieure régulière,
`B'` : transposée de `B`.

MATLAB prévoit la fonction `chol` pour cette factorisation.

```
>> A = [1 1 1 ; 1 2 3 ; 1 3 6];

>> B = chol(A)
B =
     1     1     1
     0     1     2
     0     0     1
```

On vérifie bien que `B' * B = A`

```
>> B'*B == A
ans =
     1     1     1
     1     1     1
     1     1     1
```

Décomposition QR

La fonction `qr` produit la décomposition QR d'une matrice `A` :

`A = Q * R`

`R` : matrice triangulaire supérieure, de mêmes dimensions que `A`,
`Q` : matrice unitaire (la valeur absolue du déterminant est égal à 1).

```
>> A = [1 5 8;3 5 6; 2 7 3];
>> [Q,R] = qr(A)

Q =
   -0.2673    0.6344   -0.7253
   -0.8018   -0.5639   -0.1978
   -0.5345    0.5287    0.6594
```

```
R =
   -3.7417   -9.0869   -8.5524
         0    4.0532    3.2778
         0         0   -5.0113
```

Décomposition de Schur

Toute matrice `A` peut s'écrire :

`A = U * T * U'`

`U :` matrice unitaire,
`T :` matrice triangulaire supérieure.

Cette décomposition s'obtient par la fonction `schur`.

```
>> A = [2 7 8;-4 5 6;0 3 -5];
>> [U,T] = schur(A)
U =
   -0.9542    0.2860   -0.0874
    0.2979    0.9345   -0.1947
   -0.0260    0.2118    0.9770
T =
    1.5374   -7.0207   -4.2882
    5.0213    7.3740    5.5319
         0         0   -6.9114
```

Valeurs singulières d'une matrice

Pour toute matrice `A`, il existe deux matrices unitaires `U` et `V` et une matrice diagonale `S` dont les coefficients diagonaux sont les valeurs singulières de `A`.

`A = U*S*V'`

Cette décomposition sera obtenue par la fonction `svd.`

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

U =
    0.6022    0.7983
   -0.7983    0.6022

S =
    5.8549         0
         0    4.4407

V =
    0.9903   -0.1387
    0.1387    0.9903
```

Factorisation triangulaire LU

Pour toute matrice carrée régulière `A`, on peut écrire la décomposition suivante :

```
P * A = L * U
```

`U :` matrice triangulaire supérieure,
`L :` matrice triangulaire inférieure à diagonale unité,
`P :` matrice de permutation.

Cette décomposition peut être obtenue par la fonction `lu`.

```
>> A = [1 3 2; 2 4 5  ; 3 5 6];
>> [L,U,P] = lu(A)
L =
    1.0000         0         0
    0.3333    1.0000         0
    0.6667    0.5000    1.0000
U =
    3.0000    5.0000    6.0000
         0    1.3333    0.0000
         0         0    1.0000
P =
     0     0     1
     1     0     0
     0     1     0
```

MATLAB propose d'autres méthodes de factorisation et de décomposition de matrices, pour cela, il vous est conseillé de consulter le manuel de référence ou d'exécuter la commande "`help matfun`".

VI. Matrices creuses et fonctions associées

Dans certains problèmes de modélisation de phénomènes physiques, on aboutit très souvent à la résolution de systèmes linéaires qui peuvent être de très grandes dimensions mais dont la matrice possède très peu d'éléments non nuls. Une telle matrice est dite "creuse" (`sparse` en anglais).

Pour résoudre de tels systèmes, il convient de mettre en oeuvre des techniques permettant d'éviter de stocker des termes nuls ou d'effectuer des opérations dont l'un des termes est nul.

Un modèle à n composantes fait intervenir une matrice carrée d'ordre n dont la résolution nécessite n^2 mots mémoire et un temps de calcul proportionnel à n^3.

MATLAB propose un certain nombre de fonctions pour le stockage et la manipulation de matrices creuses. Ces fonctions permettent de réduire l'espace mémoire nécessaire et le temps de calcul.

Ci-après, on crée la matrice x dont les valeurs sont distribuées selon la loi de Gauss.

```
>> x=randn(8) ;
```

On décide, pour la rendre creuse, de ne garder que les valeurs supérieures à 1 et de remplacer toutes les autres par 0.

```
>> x=x.*(x>1)
```

La représentation de x en tant que matrice creuse se fait grâce à la commande `sparse`.

```
>> x_creuse=sparse(x)
x_creuse =
   (6,2)       1.5352
   (2,4)       1.5929
   (3,4)       1.0184
   (8,5)       2.1122
   (3,6)       1.0378
   (7,6)       1.5532
   (1,7)       1.9574
   (3,7)       1.8645
   (7,7)       1.1902
```

Toutes les opérations et fonctions matricielles MATLAB peuvent s'appliquer aux matrices creuses. Les opérations sur les matrices creuses retournent des matrices creuses.
Pour obtenir la matrice pleine on utilisera la fonction `full`.

```
>> x_pleine = full(x_creuse);
```

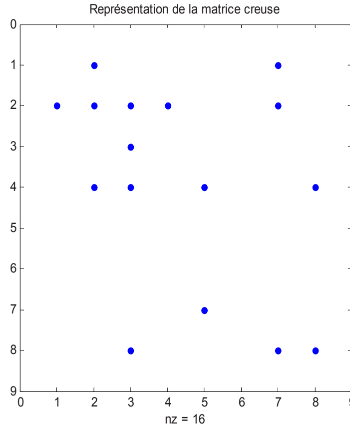
On peut transformer la matrice pleine originelle en une matrice creuse par la commande `sparse`, qui renvoie la liste des éléments non nuls avec leurs indices.
L'exécution de la commande `whos` permet d'observer l'occupation mémoire des variables `x_creuse` et `x_pleine`. Nous proposons ci-dessous un extrait du résultat de cette commande.

```
>> whos
  Name            Size              Bytes  Class     Attributes

  x               8x8                 512  double
  x_creuse        8x8                 156  double    sparse
  x_pleine        8x8                 512  double
```

On remarque bien que la matrice creuse occupe 30% environ de l'espace mémoire nécessaire au stockage de la matrice pleine correspondante.
Nous utiliserons la commande `spy` pour une visualisation graphique de la densité d'une matrice.

```
>> spy(x_creuse)
>> title('Représentation de la matrice creuse')
```



Le nombre d'éléments non nuls, est `nz`=16 sur les 64 éléments de la matrice originale.

Les commandes `nnz` et `nonzeros` donnent respectivement le nombre d'éléments non nuls d'une matrice et la liste de ces valeurs.

Pour tester si une matrice est creuse, on dispose de la commande `issparse` qui retourne `1` dans ce cas et `0` dans le cas contraire.

```
>> issparse(x_creuse)
ans =
     1
```

Nous disposons d'autres commandes et fonctions pour la manipulation de matrices creuses.

`spfun('fonction',x_creuse)` applique la fonction donnée aux seuls éléments non nuls de x_`creuse`.

```
>> exp(x_creuse)
ans =
   (1,1)        1.0000
   (2,1)        1.0000
   (3,1)        1.0000
   (4,1)        1.0000
   (5,1)        1.0000
   (6,1)        1.0000
   (7,1)        2.8815
   (8,1)        1.0000
   (1,2)        1.0000
   (2,2)        1.0000
```

L'exponentielle est ici appliquée à tous les éléments, y compris ceux qui sont nuls.

```
>> spfun('exp',x_creuse)

ans =
   (7,1)        2.8815
   (5,2)        3.7065
   (6,4)        3.6421
   (8,4)        3.6000
   (8,5)        2.9562
   (3,6)        3.8171
   (7,6)        5.0518
   (3,7)        5.6728
   (4,7)        6.9411
   (5,7)        5.1298
   (5,8)        2.9508
   (6,8)       10.7258
```

Dans ce cas, la fonction `exp` n'est appliquée qu'aux seuls éléments non nuls, comme le montre l'affichage précédent.

Nous disposons de plusieurs méthodes pour la génération de matrices creuses.

Matrice creuse identité

```
>> speye(4)

ans =
   (1,1)         1
   (2,2)         1
   (3,3)         1
   (4,4)         1
```

Matrices creuses aléatoires

`sprandn(m,n,densite)` : génère une matrice creuse aléatoire à `m` lignes et `n` colonnes de densité spécifiée. Les valeurs sont normalement distribuées autour de zéro,

`sprandn(x_creuse)` : génère une matrice aléatoire de même structure que `x_creuse`,

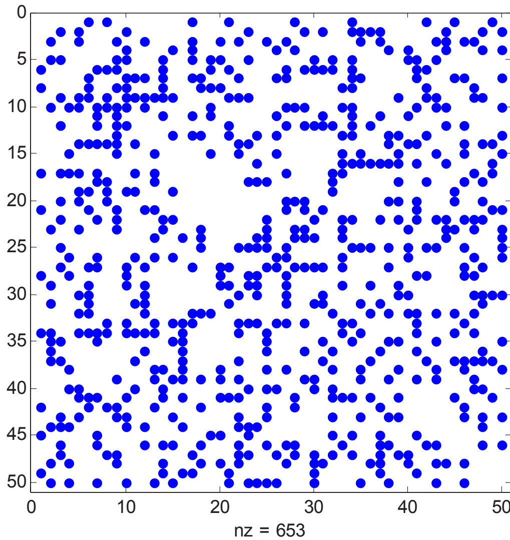
`sprandsym(n,densite)` : génère une matrice creuse, carrée, symétrique et aléatoire d'ordre `n`.

```
>> sprandn(x_creuse)

ans =
   (1,1)       -0.8654
   (4,1)       -0.8087
   (2,2)       -1.6858
   (5,3)        0.3582
   (4,4)       -0.6719
```

```
   (2,5)       -1.0210
   (3,5)        2.4989
   (6,6)       -0.6320
   (7,7)        0.7838
   (4,8)        0.2405
   (8,8)       -1.4705
   (9,9)        0.3349
  (10,10)       1.5990
```

```
>> mr = sprandsym(50,0.3);
>> spy(mr)
>> title('Matrice creuse symétrique aléatoire')
```



La saisie d'une matrice creuse peut se faire en indiquant seulement les éléments non nuls de la matrice originelle, par l'intermédiaire de la fonction `sparse` qui obéit à la syntaxe :

`sparse(i,j,s,m,n)`

`i` et `j` sont des vecteurs qui contiennent les indices lignes et colonnes des éléments non nuls, dont les valeurs sont données par le vecteur `s`.

Si `n` et `m` ne sont pas spécifiés, la matrice obtenue a pour nombre de lignes la valeur maximale de `i` et pour nombre de colonnes la valeur maximale de `j`.

Considérons la matrice suivante dont la saisie sera réalisée dans le fichier `sparse1.m`.

fichier sparse1.m

```
i = [4 8 10 2 3 6 8 4  7];  j = [1 1  2 5 5 5 6 9 10];
s=[-17.70  -16.55  19.56  17.39  -15.87  -15.55  -23.59  -17.55
17.87];
% taille matrice, n lignes et m colonnes
n = 10;
m = 10;
m_creuse = sparse(i,j,s,n,m)
```

```
>> sparse1
m_creuse =
   (4,1)      -17.7000
   (8,1)      -16.5500
  (10,2)       19.5600
   (2,5)       17.3900
   (3,5)      -15.8700
   (6,5)      -15.5500
   (8,6)      -23.5900
   (4,9)      -17.5500
   (7,10)      17.8700
```

VII. Applications

VII.1. Moyenne et variance d'une série de mesures

On dispose des 5 mesures suivantes dont on désire calculer la moyenne et la variance :

$$\begin{bmatrix} 1.12 & 1.05 & 1.25 & 1.26 & 1.39 \end{bmatrix}$$

Si m désigne la moyenne, l'estimation non biaisée de la variance est donnée par :

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - m)^2$$

```
>> x = [1.12, 1.05, 1.25, 1.26, 1.39];
>> moy_x = sum(x)/(length(x));
moy_x =
    1.2140
```

Centrage de toutes les composantes de x, élévation au carré, sommation et variance

```
>> x_centre = x-moy_x(x)
x_centre =
   -0.0940   -0.1640    0.0360    0.0460    0.1760
```

```
>> x2 = x_centre.^2
x2 =
    0.0088    0.0269    0.0013    0.0021    0.0310
```

```
>> somme_x2 = sum(x2)
```

```
somme_x2 =
    0.0701
```

```
>> var = somme_x2/(length(x)-1)
var =
    0.0175
```

On peut réduire le nombre d'instructions par composition de commandes.

```
>> var = sum((x-moy_x).^2)/(length(x)-1);
```

Pour récupérer la moyenne et la variance, il suffit d'invoquer les noms des variables, soit individuellement, soit dans un tableau.

```
>> mv = [moy_x var]
mv =
    1.2140    0.0175
```

Nous voyons ici toute la puissance de MATLAB par rapport aux langages dits évolués tels Pascal ou C pour lesquels nous avons besoin de 2 boucles, une pour le calcul de la moyenne et l'autre pour celui de la variance.

MATLAB facilite encore plus la programmation dans la mesure où beaucoup de fonctions sont déjà prédéfinies ; la moyenne et la variance sont directement accessibles en utilisant les fonctions `mean` et `std`.

```
>> moy = mean(x)
moy =
    1.2140
```

```
>> variance = std(x)^2
variance =
    0.0175
```

D'autres fonctions statistiques permettent de calculer la médiane, les valeurs minimale et maximale d'une série de valeurs.

```
>> [min(x) max(x) median(x)]
ans =
    0     8     2
```

VII.2. Dérivée d'une fonction

La fonction `diff` appliquée à un tableau `x` de taille `n` donne un tableau de taille `(n-1)` dont chaque élément correspond à la différence de deux éléments successifs de `x`.

```
>> x = [0 3 8 1];
>> diff(x)
ans =
     3     5    -7
```

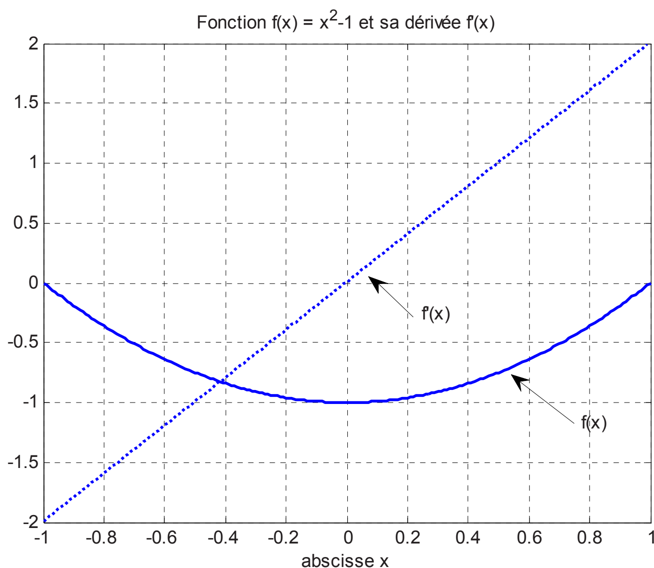
Cette fonction peut être utilisée pour le calcul de la dérivée d'une fonction mathématique `f(x)` par la formule suivante :

```
f'(x) = df/dx = diff(f)./diff(x)
```

Le programme suivant, illustre le calcul et le tracé de la courbe représentative de la dérivée de la fonction $f(x) = x^2-1$.

fichier derivee.m

```
clc, clear all, close all
% dérivée de la fonction f(x) = x^2-1
x = -1:.01:1;    % intervalle de la variable x
f = x.^2-1; % fonction à dériver
plot(x,f);  % tracé de la fonction f(x)
hold on
dx = diff(x);    % incréments dx de la variable x
df = diff(f);    % incréments df de la fonction f(x)
df_dx = df./dx; % dérivée de f(x)
% le vecteur f' possède un élément de moins que f
plot(x(1:length(x)-1),df_dx,':')
axis([-1 1 -2 2])
gtext('f(x)')    % légende de la courbe de f(x)
gtext('f''(x)') % légende de la courbe de f'(x)
title('Fonction f(x) = x^2-1 et sa dérivée f''(x)'), grid
```



Dans le cas de fonctions polynomiales, MATLAB permet de calculer plus simplement la dérivée en utilisant la fonction `polyder`.

La fonction $f(x)=x^2-1$ peut être représentée par le polynôme p suivant :

```
>> p = [1 0 -1];
```

Le polynôme représentant la fonction dérivée est donné par :

```
>> p_deriv = polyder(p)
p_deriv =
     2     0
```

L'évaluation du polynôme sur l'ensemble des valeurs, donné par le vecteur x, est réalisée par la fonction polyval.

```
>> polyval(p_deriv,x)
ans =
     2     4
```

VII.3. Calcul d'intégrales

On cherche à calculer les intégrales de la forme suivante : $I(x) = \int_{-1}^{x} f(t)\, dt$

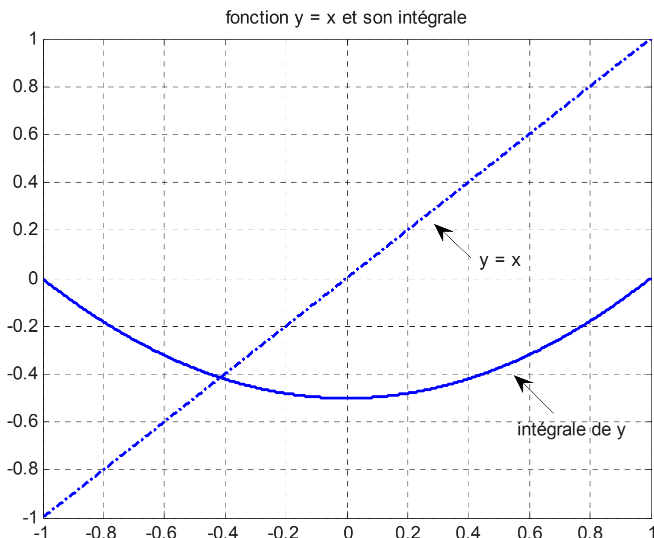
Nous nous intéresserons au cas où $f(t) = t$, dont le résultat théorique est :

$$I(x) = \left[\frac{t^2}{2}\right]_{-1}^{x} = \frac{x^2-1}{2}$$

Pour calculer cette intégrale à l'aide de MATLAB, on peut utiliser la méthode des rectangles qui consiste à approcher l'aire sous la courbe par celle des rectangles de longueur f(x) et de largeur dx (le pas d'intégration). La valeur de l'intégrale dans l'intervalle [a,b] est la somme cumulée des surfaces des rectangles élémentaires. La somme cumulée est réalisée par la fonction cumsum.
La valeur obtenue par cette méthode est d'autant plus proche de la valeur théorique que le pas d'intégration est plus faible. Dans cet exemple, ce pas vaut 0.001.

fichier integ.m

```
clear all, close all, clc
dx = 0.001; % pas d'intégration
a = -1; b = 1; % limites du domaine d'intégration
x = a:dx:b; y = x;
g = cumsum(y.*dx); % somme cumulée des aires des rectangles
% tracé de l'intégrale
plot(x,g), hold on
% tracé de la fonction y = f(x)
plot(x,y,'-.'), grid, gtext('y = x'), gtext('intégrale de y')
title('fonction y = x et son intégrale'), hold off
```



La courbe obtenue correspond parfaitement au résultat théorique.
La fonction `polyquad`, proposée dans le chapitre « `polynômes` », réalise l'intégration d'un polynôme. L'instruction suivante donne la primitive du polynôme, s'annulant en `x = 0`.

$$p(x) = 6x^2 + 2x + 1$$

```
>> p_integ = polyquad([6 2 1])
p_integ =
     2     1     1     0
```

VII.4. Résolution d'un système d'équations linéaires

Nous proposons le système suivant, comme exemple de résolution de systèmes linéaires.

$$\begin{cases} 2\,x_1 + 3\,x_2 = 8 \\ x_1 - 2\,x_2 = -3 \end{cases}$$

Ce système peut être mis sous la forme matricielle `A X = B`.

fichier systlin.m

```
% solution X = inv(A) * B ou X = A\B
A = [2 3;1 -2];
B = [8 -3]';
```

```
X = inv(A)*B;
disp('Solutions:')'
disp(['x1 = ',num2str(X(1)),' et x2 = ',num2str(X(2))])
```

```
>> systlin
Solutions :
x1 = 1 et  x2 = 2
```

Les solutions peuvent être obtenues par la méthode des moindres carrés en utilisant la fonction `nnls`.

```
>> x = nnls(A,B)
x =
    1.0000
    2.0000
```

VII.5. Résolution d'un système sous-dimensionné ou indéterminé

C'est le cas où le nombre d'inconnues est supérieur à celui des équations. Considérons le cas de l'exemple suivant :

$$\begin{cases} 2x_1 + x_2 - 3x_3 = 1 \\ x_1 - 2x_2 + x_3 = 2 \end{cases}$$

Le système se présente sous la forme matricielle `Ax = b`.

$$\begin{bmatrix} 2 & 1 & -3 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Un tel système possède une solution si le rang de la matrice `A` est égal à celui de la matrice augmentée `Ab`.
La matrice `Ab` est formée de la matrice `A` à laquelle on ajoute une quatrième colonne formée des composantes du vecteur `b`.

Matrices du système

```
>> A = [2 1 -3;1 -2 1];
>> b = [1;2];
```

Calcul du rang de la matrice A

```
>> rang_A = rank(A)

rang_A =
     2
```

Construction de la matrice augmentée Ab

```
>> Ab = [A b]
```

```
Ab =
     2     1    -3     1
     1    -2     1     2
```

Rang de la matrice augmentée

```
>> rang_Ab = rank(Ab)
rang_Ab =
     2
```

Une solution du système est donnée par :

```
>> X = A\b
X =
         0
   -1.4000
   -0.8000
```

`A\b` est équivalent à `inv(A)*b` si `A` inversible.
La matrice `A` étant rectangulaire, donc non inversible, on peut utiliser sa pseudo-inverse donnée par la fonction `pinv`.

On obtient alors, une autre solution pour le système indéterminé.

```
>> X = pinv(A)*b

X =
    0.7333
   -0.6667
   -0.0667
```

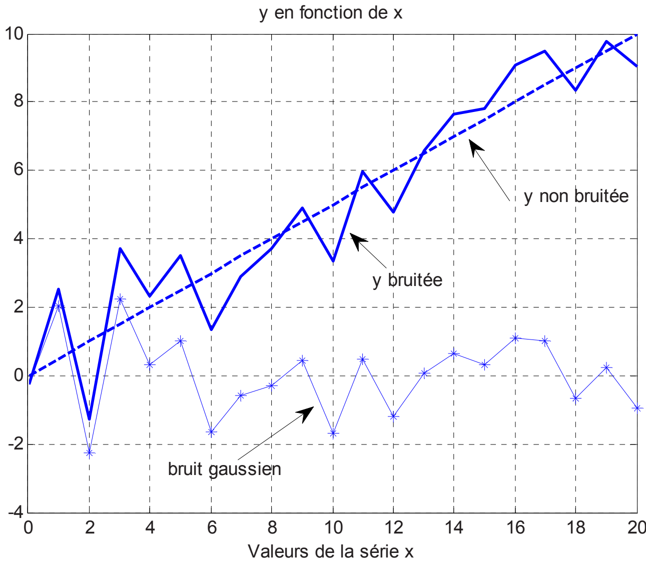
Si une matrice est carrée et inversible, alors son inverse est identique à sa pseudo-inverse.

VII.6. Régression linéaire

On s'intéresse à la recherche du modèle linéaire `Y = f(X)` liant `2` séries de mesures `X` et `Y`.
Comme exemple, on simulera la fonction `y = f(x)` par la relation `y = 0.5 x + 2`, à laquelle on superposera un bruit gaussien centré et de variance unité.

fichier reg_lin.m

```
close all
x = 0:20; y = 0.5*x;
b=randn(1,21); yb=y+b; plot(x,y), hold on
plot(x,yb), plot(x,b), grid
title('y en fonction de x')
gtext('y non bruitée'), gtext('y bruitée')
gtext('bruit gaussien'), xlabel('Valeurs de la série x')
```



Chaque mesure y_i est liée à la mesure x_i par la loi $y_i = a\ x_i + b$.
Le modèle appliqué à toutes les mesures donne le système matriciel suivant :

$$\begin{bmatrix} y(1) \\ y(2) \\ . \\ . \\ y(21) \end{bmatrix} = \begin{bmatrix} x(1) & 1 \\ x(2) & 1 \\ . & . \\ . & . \\ x(21) & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

que l'on note :

$$Y = \Phi\ \theta$$

Le vecteur optimal des paramètres qui minimise la variance de l'erreur entre les mesures et le modèle est donné par :

$$\theta = (\Phi^T \Phi)^{-1}\ \Phi^T\ Y$$

Construction de la matrice Φ (suite du fichier reg_lin.m)

```
phi = ones(21,2);
phi(:,1) = x'; % 1-ère colonne de phi = vecteur x
```

Construction du vecteur Y et du vecteur θ optimal (suite du fichier reg_lin.m)

```
Y = y';
% algorithme des moindres carrés
```

```
teta = inv(phi'*phi)*phi'*Y;

a = teta(1,1);
b = teta(2,1);
disp(['paramètres : a = 'num2str(a) 'b = ' num2str(b)]);

a =
    0.4653

b =
    2.0187
```

Tracé du nuage des points de mesures et de la droite de régression (suite fichier reg_lin.m)

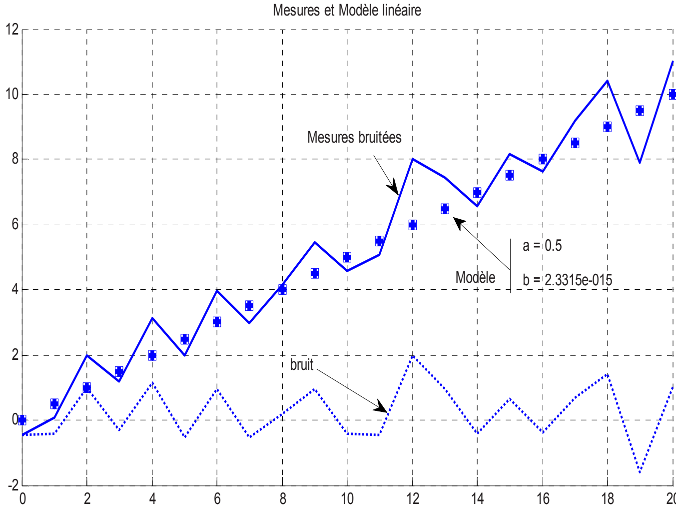
```
close all
x = 0:20;
y = 0.5*x;
br=randn(1,21);
yb=y+br;
hold on
plot(x,yb)
grid

phi = ones(21,2);
phi(:,1) = x';  % 1-ère colonne de phi = vecteur x

Y = y';
% algorithme des moindres carrés
teta = inv(phi'*phi)*phi'*Y;

a = teta(1,1);
b = teta(2,1);
gtext(['a = ' num2str(a)]);
gtext(['b = ' num2str(b)]);
ym = a*x+b;

% mesures bruitées
hold on
plot(x,y,'+','LineWidth',3)
plot(x,br)
% sortie du modèle
plot(x,ym,'s')
plot(x,y)
gtext('Mesures bruitées')
gtext('Modèle')
title('Mesures et Modèle linéaire'), grid on
```



A partir des matrices `X` et `Y`, on dispose dans MATLAB, de la fonction `nnls` pour "`non negative least squares`" permettant le calcul immédiat du vecteur des paramètres du modèle par la méthode des moindres carrés.

Le coefficient `b` du modèle linéaire étant très faible ($2.33\ 10^{-15}$), le signal non bruité est confondu avec celui du modèle obtenu par la commande `nnls` des moindres carrés.

VII.7. Régression non linéaire

On dispose des 2 séries de mesures x et y suivantes :

x	*0.1*	*0.2*	*0.5*	*1.0*	*1.5*	*1.9*	*2.0*	*3.0*	*4.0*	*6.0*
y	*0.95*	*0.89*	*0.79*	*0.70*	*0.63*	*0.58*	*0.56*	*0.45*	*0.36*	*0.28*

1. On cherche une approximation exponentielle liant y à x, sous la forme :

$$y = A\, e^{Bx}$$

Les coefficients A et B, calculés par la méthode de Newton, sont donnés par les relations suivantes :

$$B = \frac{\sum[x\, ln\, y] - \sum x \sum[ln\, y / n]}{\sum x^2 - (\sum x)^2 / n} \qquad A = \frac{\sum(e^{Bx} / y)}{\sum(e^{2Bx} / y^2)}$$

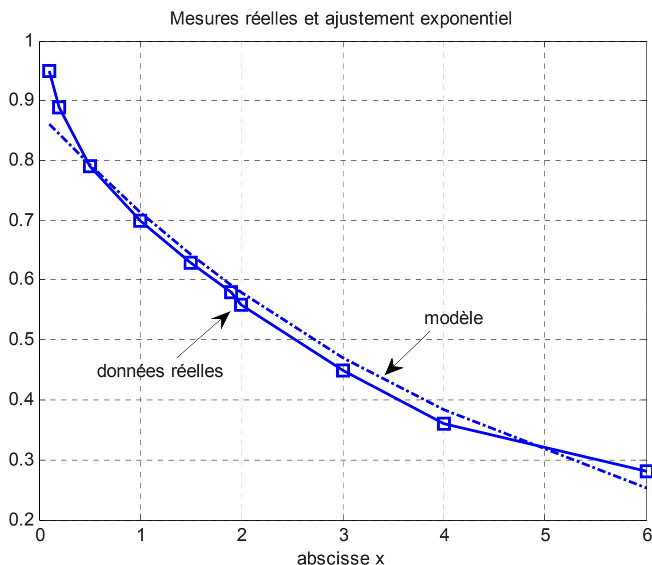
En utilisant les fonctions propres aux vecteurs, nous allons calculer les valeurs des coefficients A et B. Nous tracerons sur un même graphique, la courbe des données réelles et celle du modèle exponentiel obtenu.

fichier ajustexp1.m

```
x = [0.1 0.2 0.5 1.0 1.5 1.9 2.0 3.0 4.0 6.0];
y = [0.95 0.89 0.79 0.70 0.63 0.58 0.56 0.45 0.36 0.28];
n = length(x);
B=(sum(x.*log(y))-sum(x)*sum(log(y)/n))/(sum(x.^2)-
sum(x)^2/n);
A = sum(exp(B*x)./y)/sum(exp(2*B*x)./y.^2);
ym = A*exp(B*x);     % modèle

% tracé des données réelles et du modèle
close all, clc
plot(x,y), hold on, plot(x,y,'s'), plot(x,ym,'-.')
title('Mesures réelles et ajustement exponentiel')
xlabel('abscisse x')
gtext('données réelles')
gtext('modèle'
grid
```

```
>> A
A =
    0.8795
>> B
B =
   -0.2080
```



2. On désire obtenir les coefficients A et B du modèle exponentiel précédent par application de la méthode des moindres carrés.

Le modèle linéarisé est : $\ln y = \ln A + Bx$. Ce modèle appliqué à toutes les données donne le système matriciel suivant :

$$\begin{bmatrix} \ln y(1) \\ \ln y(2) \\ . \\ . \\ \ln y(10) \end{bmatrix} = \begin{bmatrix} 1 & x(1) \\ 1 & x(2) \\ . & . \\ . & . \\ 1 & x(10) \end{bmatrix} \begin{bmatrix} \ln A \\ B \end{bmatrix}$$

que l'on note :

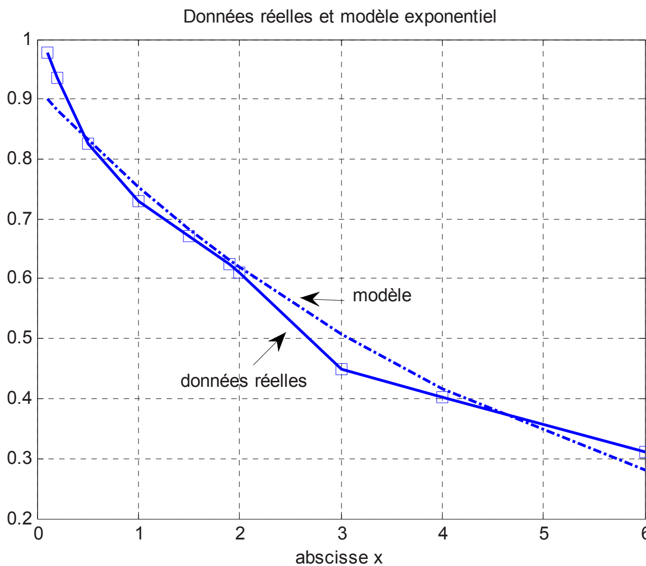
$$Y = \Phi \ \theta$$

Le vecteur paramètres est obtenu par :

$$\theta = (\Phi^T \Phi)^{-1} \Phi^T Y$$

fichier ajstexp2.m

```
x = [0.1 0.2 0.5 1.0 1.5 1.9 2.0 3.0 4.0 6.0];
y   =   [0.95   0.89   0.79   0.70   0.63   0.58   0.56   0.45   0.36
0.28]+0.05*rand(size(x));
% construction de la matrice de mesures phi
phi = [ones(length(x),1) x'] ; b = log(y);
Y = b' % vecteur Y
% calcul du vecteur teta optimal
teta = inv(phi'*phi)*phi'*Y;
% extraction des coefficients A et B du modèle exponentiel
A = exp(teta(1,1)); B = teta(2,1);
ym = A*exp(B*x) % modèle exponentiel
% tracé des données réelles et du modèle
plot(x,y), hold on, plot(x,y,'s'),plot(x,ym,'-.'), grid
gtext('données réelles'), gtext('modèle')
```



```
>> A
A =
    0.9180

>> B
B =
   -0.1972
```

3. Le système linéarisé appliqué à chaque couple de données (x_i, y_i) permet d'écrire :

$$\ln y_i = \ln A + B x_i$$

que l'on peut mettre sous la forme :

$$f_i = \theta_1 + \theta_2 x_i$$

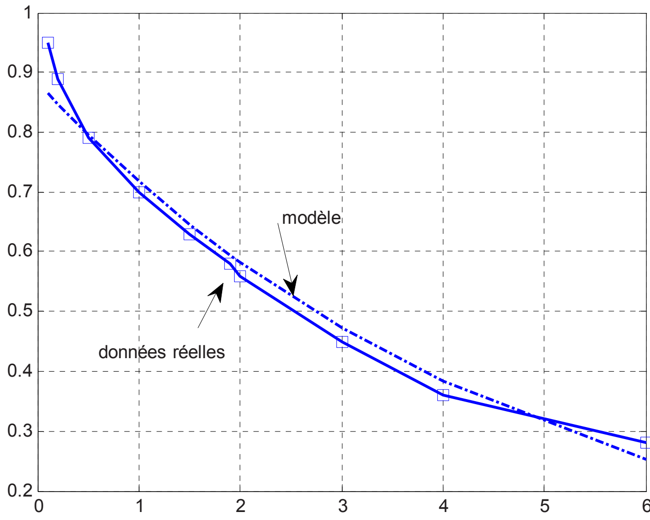
$$\theta_1 = \frac{\begin{vmatrix} \Sigma f_i & \Sigma x_i \\ \Sigma f_i x_i & \Sigma x_i^2 \end{vmatrix}}{\begin{vmatrix} n & \Sigma x_i \\ \Sigma x_i & \Sigma x_i^2 \end{vmatrix}} \quad , \quad \theta_2 = \frac{\begin{vmatrix} n & \Sigma y_i \\ \Sigma x_i & \Sigma x_i y_i \end{vmatrix}}{\begin{vmatrix} n & \Sigma x_i \\ \Sigma x_i & \Sigma x_i^2 \end{vmatrix}}$$

La résolution par la méthode de Cramer permet d'avoir les paramètres θ_1 et θ_2 .
Nous nous proposons de calculer ces paramètres en programmant les relations ci-dessus.

fichier ajstexp3.m

```
x = [0.1 0.2 0.5 1.0 1.5 1.9 2.0 3.0 4.0 6.0];
y = [0.95 0.89 0.79 0.70 0.63 0.58 0.56 0.45 0.36 0.28];
% construction de la matrice de mesures phi
f = log(y);
num_teta1 = det([sum(f) sum(x);sum(x.*f) sum(x.^2)]);
den_teta1 = det([length(x) sum(x);sum(x) sum(x.^2)]);
teta1 = num_teta1/den_teta1;
num_teta2 = det([length(x) sum(f);sum(x) sum(x.*f)]);
den_teta2 = den_teta1; teta2 = num_teta2/den_teta2;
% extraction des coefficients A et B du modèle
A = exp(teta1); B = teta2;
% calcul sortie du modèle
ym = A*exp(B*x);
% tracé des données réelles et du modèle
plot(x,y), hold on, plot(x,ym,'-.'), grid
gtext('données réelles'), gtext('modèle')
```

```
>> A
A =
    0.8835
>> B
B =
   -0.2080
```



1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
1
2
3
4
5
6
modèle
données réelles