Introduction à *MATLAB*

Christophe Labourdette et Emmanuel Motchane

Septembre 2000

Table des matières

1 Présentation du logiciel

MATLAB est un logiciel d'utilisation facile permettant de faire des calculs numériques sans avoir besoin de programmer dans un langage traditionnel de type C, C++, Fortran ou Pascal. C'est un langage interprété, ce qui signifie que l'on peut taper des commandes qui s'exécutent immédiatement sans avoir à passer par une phase de compilation. Néanmois il est possible d'écrire des programmes qui sont alors une suite de commandes que l'on met dans un fichier.

Le coeur de *MATLAB* , inaccessible aux utilisateurs a été écrit en C, ce qui permet une optimisation des opérations les plus courantes. On peut ensuite bâtir indéfiniment sur ce noyau en utilisant les fonctions de base pour définir ses propres fonctions, ou utiliser celles écrites par d'autre personnes (certaines bibliothèques de fonctions sont vendues avec *MATLAB* , d'autres peuvent être acquises moyennant une majoration de prix, toolbox).

Le nom *MATLAB* signifie "matrix laboratory" et contient toute la philosophie de *MATLAB* . Le principal type de donnée dans *MATLAB* est la matrice (réelle ou complexe). Le mot matrice s'entend dans le sens habituel, i.e. un tableau rectangulaire de m lignes et n colonnes. Les scalaires sont des matrices à 1 ligne et 1 colonne et les vecteurs des matrices à n lignes et une colonne. Les noms de variables et de fonctions sont composées de lettres et de chiffres, seuls les 19 premiers caractères sont significatifs.
Attention! *MATLAB* fait la distinction entre les majuscules et les minuscules, les variables A et a sont donc différentes.

2 Le démarrage de *MATLAB*

Chaque utilisateur va être amené à créer des fichiers *MATLAB* . Il est donc conseillé de créer un répertoire Matlab et d'invoquer *MATLAB* à l'intérieur de ce répertoire à chaque utilisation. Pour démarrer *MATLAB* , il suffit de taper

```
> matlab
```

Notez que ci-dessus ainsi que dans toute la suite le caractère > représente le prompt, il ne faut donc évidemment pas le taper.
Deux fichiers sont exécuté au lancement de la session:

- **matlabrc.m** est le fichier de démarrage principal, qui fixe la plupart des paramètres et des chemins par défaut, il invoque en général le fichier de démarrage secondaire s'il est présent dans les chemins de recherche.
- **startup.m** est le fichier de démarrage secondaire, qui se trouve généralement dans un répertoire de l'utilisateur inclu dans les chemins de recherche.

Ils peuvent tous les deux être personnalisés. Il existe un outil graphique permettant de modifier les chemins de recherche il se nomme `pathtool`.

3 Edition de la ligne de commandes

La ligne de commande peut être éditée à l'aide des commandes suivantes (le `^` signifie qu'il faut appuyer sur la touche *ctrl* en même temps que sur le caractère qui suit) :

↑ *ou* `^P`	rappeler la ligne précédente
↓ *ou* `^N`	rappeler la ligne suivante
← *ou* `^B`	déplacer le curseur d'un caractère vers la gauche
→ *ou* `^F`	déplacer le curseur d'un caractère vers la droite
`^L`	déplacer le curseur d'un mot vers la gauche
`^R`	déplacer le curseur d'un mot vers la droite
`^A`	déplacer le curseur en début de ligne
`^E`	déplacer le curseur en fin de ligne
`^U`	effacer la ligne courante
`^T`	passer du mode insertion au mode écrasement et vice-versa
`^D`	effacer le caractère sous le curseur
`^K`	effacer la fin de la ligne après le curseur

Remarque 3.1 En tapant `> plo` ↑, la dernière ligne commençant par "plo" est affichée. ■

On peut lancer une commande unix à l'intérieur de *MATLAB* en la faisant précéder d'un `!`. Néanmoins, si l'on peut travailler sur plusieurs fenêtres, il vaut mieux utiliser une autre fenêtre pour les commandes unix (comme lancer un éditeur de textes par exemple). D'autre part, des commandes comme `cd`, `dir`, `delete`,... sont intégrées dans *MATLAB* .

4 Quelques mots sur la syntaxe

Une commande *MATLAB* est généralement de la forme

$$variable = expression$$

expression étant composée de variables et d'opérateurs ou de fonctions. *MATLAB* évalue la valeur de *expression* et crée une matrice qui sera affectée à *variable*. Si on tape juste

$$expression$$

la valeur de *expression* est affectée à la variable par défaut qui s'appelle *ans*.

Exemple:
```
> a=1+1
  → a=2
> 2+2
  → ans=4
```

Pour entrer une expression on tape *return* et la valeur de l'expression est automatiquement affichée sur l'écran. Pour éviter que ce soit le cas on peut mettre un " ; " à la fin de l'expression :

Exemple:
```
> b=1+2;
  → ( aucun résultat n'est affiché )
> b
  → 3
```

Ceci est utile dans les programmes pour éviter que tous les résultats intermédiaires soient affichés.

Si une commande est trop longue pour une ligne on utilise la commande de continuation ... (3 points avec un espace avant) et on la continue sur la ligne suivante.

Exemple:
```
> somme=1+2+3+4+ ...
  5+6+7;
```

Un nom de variable *MATLAB* est une lettre suivie d'un nombre quelconque de chiffres et de lettres (mais *MATLAB* ne lit que les 19 premiers !)

De même que sous Unix *MATLAB* fait la différence entre majuscules et minuscules, toutes les commandes sont donc en minuscules :

Exemple: > inv(A) calcule l'inverse de A
> INV(A) n'est pas défini (à moins qu'on l'ait fait)

- Il y a dans *MATLAB* des variables permanentes que l'on ne peut pas effacer, mais dont on peut changer la valeur :

  Exemple:

ans	variable par défaut
eps	epsilon machine (plus petit réel positif non nul pour la machine)
realmax	plus grand réel
realmin	plus petit réel
i,j	$\sqrt{-1}$

- Informations sur l'espace de travail :
  - Pour voir quelles sont les variables déjà utilisées on tape

    ```
    > who
    ```
  - Des informations plus détaillées (taille des matrices, place mémoire, ...) sont données en tapant

    ```
    > whos
    ```

5 L'aide en ligne

La commande *help* permet d'avoir une description des fonctions *MATLAB* et des fichiers .m (fichiers liés à *MATLAB*) vendus avec le logiciel ou créées par l'utilisateur (à condition qu'ils soient documentés).

```
> help
```

donne une liste des principaux thèmes.

```
> help sujet
```

donne une information sur `sujet`. Si `sujet` est une fonction, on a la description de la fonction. Si `sujet` est un répertoire le contenu du fichier *Contents.m* de ce répertoire est affiché.

Remarque 5.2 Si l'on crée des fonctions *MATLAB* il est conseillé de les regrouper par thème dans des répertoires et de créer des fichiers *Contents.m* sur le modèle de ceux qui existent. ■

La documentation d'une fonction se fait dans les lignes de commentaires :

Exemple : Supposons que l'on veuille créer une fonction *MATLAB* qui calcule le discriminant d'un polynôme du second degré. On crée alors un fichier disc.m qui contient :

```
function d=disc(a,b,c)
% DISC discriminant d'un polynôme du second degré
% disc(a,b,c) calcule le discriminant du polynôme
% $ax^2+bx+c$ par la formule $d=b^2-4ac$.
d=b^2-4a*c;
```

```
> help disc
```

va afficher toutes les lignes de commentaires contigues à partir de la deuxième ligne du fichier *disc.m*.

Remarque 5.3 Essayez de mettre le plus d'informations possible dans la première ligne, car il existe une fonction *lookfor*

```
> lookfor motcle
```

qui recherche motclé dans toutes les premières lignes de description des fichiers .m ■

Remarque 5.4 Pour éviter qu'une longue description défile sur l'écran taper

```
> more on
```

pour activer l'affichage page à page (on tape alors sur la barre d'espace pour avoir la page suivante)

```
> more off
```

désactive ce mode. ■

6 Un peu plus d'aide ...

Il existe maintenant une fenêtre d'aide qui permet de se promener dans la documentation en ligne, elle est appellée par

```
> helpwin
```

Il existe de nombreux utilisateurs de *MATLAB* dans le monde qui sont prêts à faire partager leurs codes ou expériences, on peut donc trouver sur le web des tutoriels, des librairies, etc ...

7 Sauvegarde de l'espace de travail

On quitte une session *MATLAB* en tapant

```
> quit
```

ou

```
> exit
```

ceci efface toutes les variables. Pour les sauver on peut taper

```
> save
```

ou

```
> save nom_fich
```

sauve les différentes variables dans le fichier `matlab.mat` ou `nom_fich.mat`. On les récupère en tapant

```
> load
```

ou

```
> load nom_fichier
```

Si l'on veut uniquement sauver les variables x et y dans un fichier `temp.mat` on tape

```
> save temp x y
```

8 Les nombres complexes

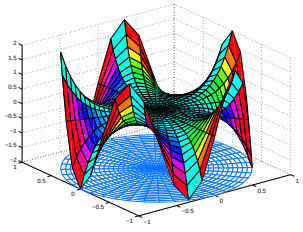
L'imaginaire pur i ($i^2 = -1$) est noté indifféremment i ou j. Dans ses réponses Matlab utilise toujours i. Un nombre complexe z se déclare comme suit :

$$z = 3 + 2i$$

On peut également utiliser la notation exponentielle : $z = r * exp(j * theta)$.
On peut alors extraire:
la partie réelle : $a = real(z)$,
la partie imaginaire : $b = imag(z)$
ou construire le complexe conjugué : $zc = conj(z)$.
On peut également obtenir le module, $r = abs(z)$ ainsi que l'argument $theta = angle(z)$ du complexe z. Bien entendu tous les calculs classiques (puissance, multiplication, addition, soustraction, etc) se font de manière transparente.
La fonction *plot* appliquée à un nombre complexe, trace son image dans le plan cartésien. La fonction *compass* trace un segment orienté partant de l'origine du plan complexe. Pour dessiner une fonction complexe $y = f(z)$ dans le plan unité (en coordonnées polaires) on utilise les fonctions $cplxgrid(n)$ et *cplxmap* , la première construit un découpage de l'espace en n domaines dans le système de coordonnées polaires, la deuxième trace la surface en prenant la partie réelle de y comme hauteur et la partie imaginaire de y comme couleur.

Exemple on veut tracer la fonction $2z^5$.

```
> z = cplxgrid(20);
> cplxmap(z,'2*z.^5')
```



9 Le calcul matriciel et vectoriel

Pour *MATLAB* l'entité de base est une matrice à éléments complexes, un nombre réel est donc considéré comme une matrice à une ligne et une colonne ne possédant qu'une partie réelle.
La plupart des fonctions de *MATLAB* acceptent comme arguments des vecteurs ou des matrices, elles retournent alors souvent des vecteurs ou des matrices de dimensions analogues aux arguments d'entrée.

9.1 Les vecteurs

Il y a en fait peu de fonctions spécifiques aux vecteurs, qui sont souvent considérés comme des matrices dont l'une des dimensions est nulle. Il suffit donc d'utiliser une fonction matricielle en spécifiant que l'une des dimension est égale à 1. Il existe pourtant deux fonctions permettant de générer de manière spécifique des vecteurs:

- > $u = linspace(debut, fin, nombre)$; génère *nombre* éléments entre *debut* et fin de manière uniforme (si *nombre* n'est pas précisé il est pris à 100).
- > $u = logspace(debut, fin, nombre)$; génère *nombre* éléments espacés d'un pas logarithmique entre 10^{debut} et 10^{fin} (si *nombre* n'est pas précisé il est pris à 50).

Un certain nombre de fonctions opèrent sur des vecteurs ou sur les colonnes de la matrice donnée en argument, on peut citer:

- > $mean(u)$; valeur moyenne,
- > $std(u)$; écart-type,
- > $sum(u)$; somme,
- > $cumsum(u)$; somme cumulée,
- > $prod(u)$; produit,
- > $cumprod(u)$; produit cumulé,
- > $min(u), max(u)$; valeurs minimale et maximale,
- > $diff(u)$; différence des éléments successifs.

9.2 Les matrices

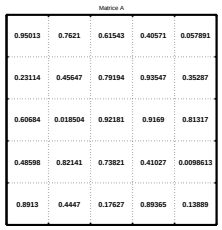
- Une expression définissant une matrice contient entre crochets tous les éléments de la matrice donnés ligne par ligne. Deux éléments successifs sont séparés par un blanc. Deux lignes successives sont séparées par un point-virgule ou un retour chariot.
  Exemple: > $A = [1\ 2; 3\ 4; 5\ 6]$
  représente la matrice
  $$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$
  La même matrice peut également être définie par
  ```
  > A=[1   2
       3   4
       5   6]
  ```
  On peut également rentrer une matrice terme à terme > $A(1,1) = 1$; $A(1,2) = 2$; $A(2,1) = 3$ ... L'affectation des tableaux est dynamique : il est inutile de prévoir de place à l'avance. Néanmoins le faire (en initialisant par exemple les tableaux à 0) peut faire gagner du temps lors de gros calculs.
- On peut charger une matrice directement à partir d'un fichier de données numériques rangées sous forme matricielle (éléments séparés par des espaces) en faisant
  > *load* nom_de_fichier $-ascii$ si le fichier est de format *ascii*. La matrice créée s'appelera alors $nomfich$.
  Il existe évidemment d'autres formats de fichiers que l'on peut lire avec *MATLAB* .
- L'opérateur : permet de créer des matrices de façon incrémentale
  - Si m et n sont deux entiers $n : m$ définit le vecteur
    $[n\ n+1\ n+2 ... m]$.

- Si m, i et n sont trois entiers $n : i : m$ définit le vecteur
  $[n\ n+i\ n+2i ... n+ki]$, où k est le plus grand entier tel que $n+ki \leq m$.
  exemple > $x = 2 : 3 : 11$
  représente $x = [2\ 5\ 8\ 11]$.
  Notez que ces représentations peuvent également s'appliquer pour des réels.
- On peut construire des matrices à partir de sous-matrices, à condition bien sûr que les dimensions soient compatibles : > $A = [A_1\ A_2;\ A_3\ A_4]$
- Il existe des commandes pour créer des matrices particulières:
  - Une matrice (pxq) remplie de zéro > $A = zeros(p, q)$;
  - Une matrice (pxq) remplie de un > $A = ones(p, q)$;
  - Une matrice identité (p) > $A = eye(p)$;
  - Une matrice aléatoire (normale) (pxq) > $A = randn(p, q)$;
  - Une matrice aléatoire (uniforme) (pxq) > $A = rand(p, q)$;
  - Un carré magique (p) > $A = magic(p)$;
  - Une matrice de Hadamard (p) > $A = hadamard(p)$;
  - Une matrice de Pascal (p) > $A = pascal(p)$;
  - Une matrice compagnon (p) > $A = compan(p)$;

9.3 Affichage

Il existe différents modes d'affichage de matrices :
- *disp* est la plus simple elle effectue un retour chariot après chaque ligne.
- *printmat* permet d'afficher une matrice en donnant un label pour les lignes et les colonnes.
- *pltmat* affiche la matrice dans une fenêtre graphique.

Cette commande demande comme argument:
- la matrice,
- le nom de la matrice,
- la couleur de la grille,
- la taille de la police de caractères utilisée pour les valeurs de la matrice.




```
> A = rand(5);
> pltmat(A,'Matrice A','b',12)
```

9.4 Extraction de sous-matrices

Soient v et w deux vecteurs. $A(v, w)$ est la sous-matrice de A contenant les lignes correspondantes aux indices de v et les colonnes correspondantes aux indices de w.
Exemple $v = [3\ 1]$ $w = [2\ 3]$ entraîne que

$$B = A(v, w) = \begin{pmatrix} A(3,2) & A(3,3) \\ A(1,2) & A(1,3) \end{pmatrix}$$

Si on utilise "**:**" à la place de v (resp. w) cela signifie que l'on veut garder toutes les lignes (resp. colonnes).
> $A(:, 3)$ est la troisième colonne de A
> $A(1 : 5, :)$ sont les cinq premières lignes de A

9.5 Les opérateurs

Il existe dans *MATLAB* des opérateurs qui font des opérations matricielles dans le sens classique du terme et des opérateurs qui agissent terme à terme. En voici la description :

opérateur	version matricielle	version terme à terme
addition	+	+
soustraction	-	-
multiplication	*	.*
division à droite	/	./
division à gauche	\	.\
puissance	^	.^

Remarque 9.5 La division matricielle à droite (resp. à gauche) est définie par : b/A est l'élément x tel que $xA = b$ (resp. $A\backslash b$ est l'élément x tel que $Ax = b$). ■

Les opérations mathématiques usuelles telles que sin, cos, log ... s'appliquent terme à terme sur les matrices.
Transposition: La transposée d'une matrice A est obtenue en tapant > A'

9.6 Comparaisons

On peut également utiliser des opérateurs logiques pour comparer deux matrices de même dimension (terme à terme). On peut aussi comparer une matrice à un scalaire, ce qui équivaut à comparer chaque élément de la matrice à ce scalaire. Les opérateurs de comparaison sont :

opérateur	définition
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
~=	différent de

Le résultat est une matrice logique composée de 0 et de 1, les 0 correspondant aux termes ne vérifiant pas la relation logique et les 1 correspondant aux termes vérifiant la relation logique. La fonction *all* retourne 1 si tous ses éléments sont non nuls et 0 dans le cas contraire.

9.7 Transformations

Il est possible de transformer des matrices à l'aide des commandes suivantes:
- > $fliplr(A)$ échange les colonnes de gauche à droite,
- > $flipud(A)$ echange les lignes de haut en bas,
- > $rot90(A)$ tourne la matrice de 90 degré,
- > $reshape(A)$ redimensionne une matrice (n,m) en une matrice (p,q) en prenant les éléments dans le sens des aiguilles d'une montre. Bien entendu $m * n$ doit être égal à $p * q$.

9.8 Quelques fonctions

- > $expm(A)$ calcule l'exponentielle d'une matrice
  - > $expm(A)$ et $expm3(A)$ à base des valeurs et des vecteurs propres de la matrice.
  - > $expm1(A)$ à base de l'approximation de Pade.
  - > $expm2(A)$ à base des séries de Taylor.
- > $logm(A)$ calcule le logarithme d'une matrice.
- > $rank(A)$ calcule le rang d'une matrice.
- > $sqrtm(A)$ calcule la racine d'une matrice.

10 Algèbre linéaire

La décomposition LU d'une matrice A est obtenue par

$> [L, U, P] = lu(A)$

On obtient une matrice triangulaire inférieure L avec des 1 sur la diagonale, une matrice supérieure U et une matrice de permutation P.

Il est aussi possible d'entrer

$[L, U] = lu(A)$

auquel cas U sera triangulaire supérieure et L sera le produit d'une matrice triangulaire supérieure avec une matrice de permutations.

La décomposition LU sert à $MATLAB$ pour faire d'autres opérations sur les matrices telles que :

- Calcul de l'inverse : > $X = inv(A)$
- Calcul du déterminant : > $d = det(A)$
- Résolution du système $Ax = b$: > $x = A\backslash b$

$MATLAB$ peut également faire la décomposition QR d'une matrice : > [Q, R] = $qr(A)$ où Q est une matrice unitaire et R une matrice triangulaire supérieure. La factorisation QR est utilisée pour résoudre des systèmes non déterminés (plus d'équations que d'inconnues) au sens des moindres carrés

En outre $MATLAB$ permet de calculer le conditionnement d'une matrice, différentes normes matricielles, le rang d'une matrice et d'effectuer la plupart des opérations d'algèbre matricielle.

11 Les tabeaux multidimensionnels

Il est possible dans les versions récentes de $MATLAB$ d'utiliser des tableaux à n dimensions. Chaque dimension utilise un indice, pour un tableau à 3 dimensions, nous aurons donc un indice des lignes, un indice des colonnes et un indice des pages.

On crée généralement ces tableaux en utilisant judicieusement le "**:**" ou la commande cat. Les fonctions vectorielles telles que sum, std, $mean$, etc peuvent être appliquées à des vecteurs que l'on extrait d'un tableau multidimensionnel. Il est également possible de donner en paramètre la direction dans laquelle on veut effectuer l'opération (1 selon les lignes, 2 selon les colonnes, 3 selon la profondeur).

Il est possible de modifier les dimensions du tableau multidimensionnel avec la commande $reshape$.

12 Les chaînes de caractères

Pour $MATLAB$ une chaîne de caractères c'est un vecteur ligne de la taille du nombre de caractères. Elle se déclare avec des apostrophes et non des guillemets.

```
> c = 'Bonjour tout le monde';
```

Il existe de nombreuses fonctions pour manipuler les chaînes de caractères.

- > $length(c)$ calcule la longueur d'une chaîne.
- > $abs(c)$ renvoie les codes ASCII des caractères composant la chaîne.
- > $isstr(c)$ renvoie 1 si la variable testée est une chaîne de caractère et 0 sinon.
- > $num2str(c)$ converti un nombre entier ou réel en une chaîne de caractères.
- > $int2str(c)$ converti un nombre entier en une chaîne de caractère.
- > $dec2hex(c)$ transforme un nombre entier en un nombre hexadécimal sous forme de chaîne de caractères.
- > $mat2str(m, p)$ converti une matrice m de nombres en une chaîne de caractères avec une précision donnée p.
- > $str2num(c)$ transforme une chaîne de caractères valide en une matrice de nombres.
- > $hex2num(c)$ converti un nombre hexadécimal en un flottant double précision.
- > $hex2dec(c)$ converti un nombre hexadécimal en un nombre entier.

- > *isletter*(*c*) retourne un tableau contenant 1 pour les lettres de l'alphabet et 0 pour le reste.
- > *strcmp*(*c*) compare deux chaînes de caractères et retourne 1 si elles sont identiques.
- > *upper*(*c*) retourne en majuscule la chaîne spécifiée.
- > *lower*(*c*) retourne en minuscule la chaîne spécifiée.
- > *eval*(*c*) évalue une chaîne de caractères comportant des commandes.

12.1 Dates et Heures

La commande *date* retourne une chaîne de caractères représentant la date du jour.
Plus précisement on peut obtenir la date et l'heure du jour en utilisant la fonction

```
> t = clock
```

ou `t = [annee mois jour heure minutes secondes]` . On peut ensuite calculer la différence en secondes entre deux dates

```
> etime(t1,t2)
```

les vecteurs sont bien entendu du même type que le vecteur t ci-dessus.
La commande *cputime* donne le temps CPU en secondes utilisé par le processus *MATLAB* .
A l'intérieur d'une application l'exécution de la commande *tic* lance un compteur et la commande *toc* permet alors d'obtenir le temps écoulé depuis le *tic*.

13 Opérations sur les polynômes

Les polynômes sont rangés dans des vecteurs dans l'ordre des puissances décroissantes :
Exemple `p=[1 3 0 2]` représente le polynôme $x^3 + 3x^2 + 2$
Les fonctions mettant en jeu des polynômes sont :

poly(A)	polynôme caractéristique de la matrice A, ou construction à partir des racines
roots(p)	racines du polynôme p
polyval(p,x)	évaluation du polynôme p au point x
polyder(p)	polynôme dérivée du polynôme p
c=conv(a,b)	multiplie les polynômes a et b
[q,r] =deconv(c,a)	quotient et reste de la division polynômiale de c par a

14 Opérations sur les fonctions

14.1 Définition d'une fonction numérique

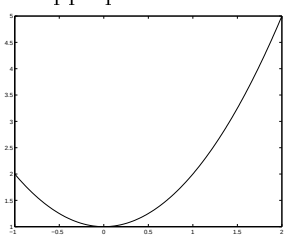
On ne peut pas définir de fonctions analytiquement avec *MATLAB* , on peut simplement donner un certain nombre des valeurs qu'elle prend. Ainsi, pour définir une fonction on crée un fichier .m contenant sa représentation analytique, ce qui permettra de l'évaluer au points voulus. **Exemple** On veut tracer un graphe de la fonction $f(x) = x^2 + 1$. On crée alors le fichier *f.m* dans lequel on écrit :

```
function y=f(x)
y=x.^2+1;
```

Il faut prendre soin de bien utiliser l'opérateur terme à terme `.^` car on va appliquer la fonction à des vecteurs.

On peut alors tracer le graphe de f en calculant f en un vecteur de points :

```
> x=-1:.01:2;
> plot(x,f(x))
```



Remarque 14.6 La fonction f est nommée par le fichier qui la contient (sans l'extension .m) et non pas par son nom dans le fichier, mais il vaut mieux prendre deux noms identiques pour éviter les confusions. ■

14.2 Objets inline

Il est possible de définir des fonctions (ou des objets) *inline*. Celles ci sont créées dans l'espace de travail et n'ont donc qu'une existence temporaire. La syntaxe est simple :

```
> f = inline('3*x + 2*y')
```

définit la fonction *inline* $f(x, y) = 3 * x + 2 * y$. Il n'y a pas de prototypage obligatoire, la fonction recherche les arguments parmis les lettres différentes de i et j, par défaut l'argument sera x.
On peut connaître les arguments d'un objet *inline* grâce à la commande *argnames* :

```
> argnames(f)
```

et en revoir la définition avec la commande *formula* :

```
> formula(f)
```

Il est également possible de vectorialiser un fonction pour qu'elle puisse accepter des arguments vectoriels au lieu d'arguments scalaires à l'aide de la commande *vectorize* :

```
> g = vectorize(f)
```

14.3 Extrema

La boite à outil optimisation fournit des méthodes pour chercher les minima de fonctions à une ou plusieurs variables ce sont les fonctions *fminbnd* et *fminsearch*. Par exemple pour une fonction à une variable on peut utiliser une fonction *inline* comme suit :

```
> f = inline('sin(x)+3');
> fminbnd(f,2,5)
```

la fonction *fminbnd* renvoie alors x_1 qui est un minimum local avec $2 < x_1 < 5$.
Bien entendu la fonction peut être définie dans un fichier.
Il est également possible de modifier les paramètres par défaut de la recherche en utilisant la commande *optimset*.
Il existe aussi la commande *fzero*, qui calcule le zéro d'une fonction au voisinage d'un point par la méthode de Newton :

```
> x=fzero('fonction',1)
```

14.4 Intégration numérique (méthode de Simpson)

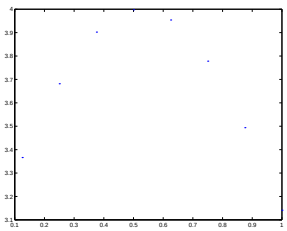
On calcule numériquement une intégrale comme $\int_0^1 f(x)\,dx$ à l'aide de la commande :

```
> q=quad('f',0,1)
```

où f est une fonction définie dans un fichier .m. Il y aussi la commande *quad8* qui permet de faire une quadrature plus précise (méthode récursive de Newton-Cotes 8), mais plus lente. On peut préciser en option la tolérance, en général un vecteur $[rel, abs]$, l'erreur sera alors une combinaison de *rel* (l'erreur relative) et de *abs* (l'erreur absolue), on peut également demander le tracé d'un graphique représentant la progression du calcul, en donnant comme paramètre supplémentaire une valeur non nulle.

Exemple : On veut calculer l'intégrale $\int_0^1 (sin(3x) + 3)dx$ avec la tolérance par défaut et en tracant la progression:

```
> f = inline('sin(3*x)+3');
> quad(f,0,1,[],1)
```



Pour les intégrales dépendant d'un paramètre (ou de plusieurs), il faut passer la valeur du paramètre après les arguments optionnels.

14.5 Résolution numérique d'une équation différentielle

Les méthodes de Runge-Kutta d'ordre 2,3 et 4,5 sont implémentées dans *MATLAB* . On les appelle à l'aide des commandes *ode23* et *ode45*. Ces commandes permettent de résoudre les systèmes différentiels d'ordre 1. Et bien sûr les systèmes d'ordre supérieurs peuvent se ramener à des systèmes d'ordre 1.

Exemple : On veut résoudre l'équation de Van der Pol qui est une équation différentielle ordinaire d'ordre 2

$$\ddot{x} + (x^2 - 1)\dot{x} + x = 0.$$

D'abord on la transforme en un système d'ordre 1

$$\begin{aligned} \dot{x}_1 &= x_1(1 - x_2^2) - x_2 \\ \dot{x}_2 &= x_1 \end{aligned}$$

on a alors un système de la forme

$$\dot{Y} = F(X, t)$$

On définit un fichier vdpol.m pour F comme suit

```
function xpoint=vdpol(t,x)
xpoint=zeros(2,1);
xpoint(1)=x(1).*(1-x(2).^2)-x(2);
xpoint(2)=x(1);
```

La deuxième ligne de ce fichier correspond à l'initialisation de la fonction. Il reste maintenant à résoudre l'EDO entre t0=0 et tf=20, et avec les conditions initiales x(0)=0, x'(0)=0.25 (par exemple)

> t0=0; tf=20;
> x0=[0,0.25]'

On calcule alors x pour des temps entre t0 et tf avec

> [t,x]=*solveur*('vdpol',[t0 tf], x0, *options*);

solveur est le nom de la commande de résolution. On dispose de

ode45 : Utilise la méthode de Runge-Kutta à un pas (Dormand-Prince (4,5)). C'est la première fonction à appliquer pour rechercher la solution d'un problème différentiel.

ode23 : Basé sur la méthode de Runge-Kutta à un pas (Bogacki-Shampine (2,3)).

ode113 : Méthode à ordre variable, basée sur le solveur Adams-Bashforth-Moulton. C'est une méthode à pas multiples.

ode15s : Méthode à ordre variable et à pas multiple, basée sur la formue de différentiation numérique et la méthode de Gear.

ode23s : Méthode basée sur la formule modifiée de Rosenbrock d'ordre 2. C'est une méthode à un pas. Elle présente d'excellentes propriétés de stabilité.

L'argument optionnel *options* représente les paramètres de résolution, fixés à l'aide de la commande *odeset*.

Les grandeurs t et x sont des vecteurs, et $x(n)$ est la solution à l'instant $t(n)$.

15 Graphiques

Il est devenu essentiel de visualiser rapidement des données, cette facilité a grandement contribué au succès et à l'extension de *MATLAB* .
En général on distingue deux catégories, les graphiques 2D d'une part et les graphiques 3D d'autre part. Les dessins s'affichent cependant dans un environnement Graphique que *MATLAB* nous permet de personnaliser.

15.1 Environnement

Par défaut toutes les opérations graphiques se font sur une seule fenêtre graphique qui s'appelle *figure 1*. On peut créer une nouvelle fenêtre graphique en tapant

```
> figure
```

son nom sera *figure n+1* si *figure n* était le nom de la dernière fenêtre ouverte. Pour que la fenêtre courante, c'est-à-dire la fenêtre sur laquelle les graphiques sont affichés, soit *figure n*, on entre la commande

```
> figure(n)
```

On peut également fermer une fenêtre graphique grace à la commande *close*. Plus particulièrement

```
close(n)
```

ferme la fenêtre *figure n*. *MATLAB* utilise une approche objet pour tout ce qui est graphique. On dispose donc d'une hiérarchie d'objet possédant un certain nombre de propriétés. Chaque objet est identifié par un nombre appelé "handle", l'écran possède le "handle" 0 c'est la racine de la hiérarchie. Il est possible de consulter les propriétés de l'objet portant le "handle" n en tapant,

```
> get(n)
```

Toutes les propriétés ne sont pas modifiables, on peut avoirla liste de celle qui le sont en tapant,

```
> set(n)
```

Pour modifier une propriété d'un objet possédant n comme "handle", on utilise la commande `set` comme suit

```
> set(n, 'Propriete', 'Valeur')
```

Où l'on veut assigner `Valeur` à `Propriete` .

15.2 Graphiques 2D

Si y est un vecteur

```
> plot(y)
```

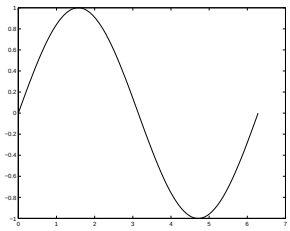
trace les points de y en fonction de l'indice, i.e. $y(i)$ en fonction de i. Si x et y sont deux vecteurs de même dimension

```
> plot(x,y)
```

trace le graphe de y en fonction de x.

Exemple : On veut tracer le graphe de sin entre 0 et 2π :

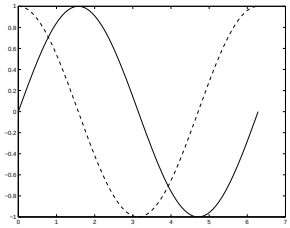
```
> t=0:pi/100:2*pi;
> x=sint(t);
> plot(t,x)
```



De manière plus générique la commande `plot(x,y,s)` trace le ou les graphiques de y en fonction de x. s est un paramètre (une chaîne de 1 à 3 caractères) permettant de choisir,

- La couleur:
  jaune `y`, magenta `m`, cyan `c`, rouge `r`, vert `g`, bleu `b`, blanc `w`, noir `k`.
- Le symbole utilisé en cas de tracé discontinu:
  point `.`, cercles `o`, croix `x`, plus `+`, étoiles `*`.
- Le symbole utilisé en acs de tracé continu (par défaut):
  trait continu `-`, pointillés `:`, trait-point `-.`, trait-trait `--`.

On peut tracer plusieurs courbes sur la même figure



Exemple : On suppose t et x définis par les commandes ci-dessus et on définit en plus :

```
> y=cos(t)
> plot(t,x,'r-',t,y,'g--'))
```

pour tracer sin avec une ligne continue rouge et cos avec une ligne pointillée verte.

On peut également faire

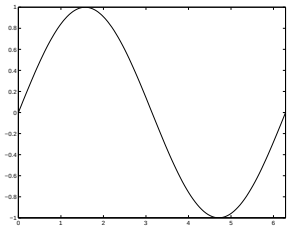
```
> plot(X,Y)
```

où Y est une matrice et X un vecteur colonne ou une matrice de même dimension. Dans ce cas, on trace les colonnes de Y en fonction des colonnes de X. Il y aura donc autant de courbes que de colonnes dans X et Y.

Normalement, à chaque fois qu'on réutilise *plot* la fenêtre graphique est effacée. Si on veut éviter cela pour tracer plusieurs graphes sur la même fenêtre avec des commandes différentes, on peut faire

```
> hold on
```

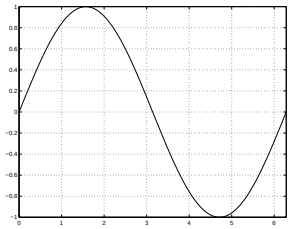
Graphe d'une fonction avec *fplot*



```
> fplot('sin',[0 2*pi])
```

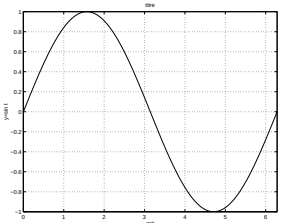
choisit de manière adaptative des points pour que la courbe soit significative. En pratique, elle prend plus de points là où la pente de la courbe est raide.

Affichage d'une grille



On obtient une grille sur le graphe par la commande **grid**. Par exemple sur la fonction sinus entre 0 et 2π.

Affichage de texte sur une fenêtre graphique



On peut afficher un titre à l'aide de la commande

```
> title('titre')
```

des légendes sur les axes avec

```
> xlabel('x=t')
> ylabel('y=sin t')
```

un texte peut être affiché sur une fenêtre graphique à l'aide de la commande *text* ou de la commande *gtext* si l'on veut le placer avec la souris. Depuis la version 5 de *MATLAB* il est possible d'utiliser les notations de TEXpour insérer des expressions mathématiques dans une chaîne de caractères. On peut alors écrire,

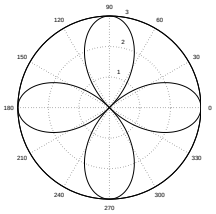
```
> xlabel('x varie de -\pi a \pi')
> ylabel('y = sin \alpha x')
```

Coordonnées Polaires

Il est possible de dessiner des courbes en coordonnées polaires à l'aide de la commande `polar(theta, rho)` qui représente `rho` en fonction de `theta` .

Exemple: On peut dessiner une rosace,

```
> theta = -pi:0.01:pi;
> rho = 3*cos(2*theta);
> polar(theta,rho)
```



Histogramme

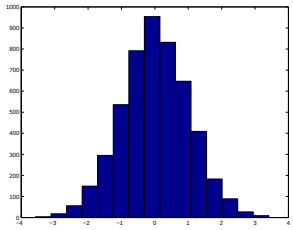
On peut également dessiner un histogramme en tapant,

```
> hist(x,n)
```

on obtient alors `n` classes du vecteur `x` .

Exemple: En prenant un échantillon de 5000 valeurs tirées aléatoirement selon une loi normale.

```
> x = randn(5000,1);
> hist (x,15)
```



15.3 Graphiques 3D

Si x,y et z sont des vecteurs

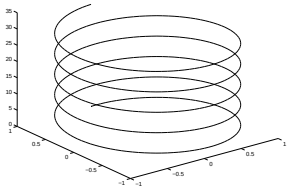
```
> plot3(x,y,z)
```

trace la courbe passant par les points de coordonnées $(x(i), y(i), z(i))$ **en projection.**

Exemple:

```
> t=0:pi/50:10*pi;
> plot3(sin(t),cos(t),t)
```

trace une hélice.



Comme dans le cas 2D si x,y et z sont des matrices avec le même nombre de colonnes, on a obtient une courbe par colonne.

Graphe d'une fonction de 2 variables

On réalise un tel graphique en deux étapes :

1. On définit une grille pavant le plan (x, y) avec *meshgrid*

   Exemple: Si on veut tracer un graphique dans l'intervalle $[-8, 8]$ on définit :

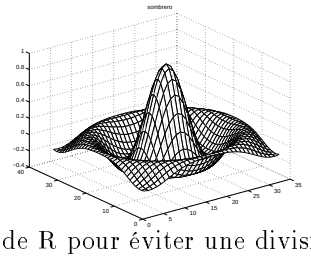
   ```
   > x=-8:0.5:8;
   > y=x;
   > [X,Y]=meshgrid(x,y);
   ```

   X est alors une matrice ayant n lignes identiques égales à x et Y est une matrice ayant n colonnes identiques égales à y (où n est la taille du vecteur x).

2. On trace le graphe avec *mesh* :

   Exemple: On veut tracer $z = sin(r)/r$, une grille de points étant définis par les matrices X et Y de l'exemple précédent :

   ```
   > R=sqrt(X.^2+Y.^2)+eps;
   > z=sin(R)./R;
   > mesh(z)
   ```



notez qu'on ajoute *eps* qui est le epsilon machine à l'expression de R pour éviter une division par 0. $z(i, j)$ définit la hauteur d'une surface au-dessus d'une grille plane (i, j).

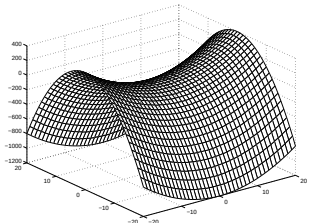
Les surfaces paramétrées

```
> mesh(X,Y,Z,C)
```

trace la surface dont le point situé aux coordonnées $X(i,j)$, $Y(i,j)$, $Z(i,j)$ a la couleur $C(i,j)$, où C définit une échelle de couleurs.

Exemple:

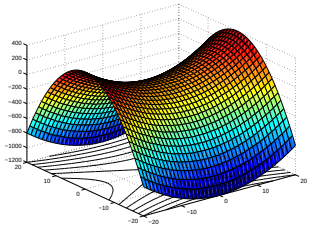
```
> x = -20:1:20;
> y = x;
> [X,Y] = meshgrid(x,y);
> Z = X.^2 - 3 * Y.^2;
> mesh(X,Y,Z)
```



On peut également dessiner la surface colorié et non plus en fil de fer grace à la commande `surf`. Il existe des commandes `meshc` et `surfc` qui projettent les contours de la surface considéré.

Exemple:

```
> x = -20:1:20;
> y = x;
> [X,Y] = meshgrid(x,y);
> Z = X.^2 - 3 * Y.^2;
> surfc(X,Y,Z)
```



Subdivision d'une fenêtre graphique en sous-fenêtres

Ceci peut se faire à l'aide de la commande *subplot*:

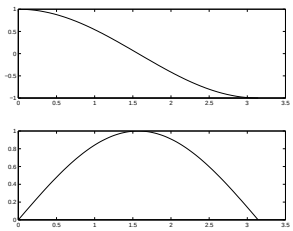
```
> subplot(m,n,p)
```

sépare la fenêtre graphique en m lignes et n colonnes de sous-fenêtres, p étant la sous-fenêtre courante.

Exemple:

```
> t=0:pi/100:pi;
> subplot(2,1,1)
> plot(t,cos(t))
> subplot(2,1,2)
> plot(t,sin(t))
```

dessine cos dans la sous-fenêtre du haut et sin dans la sous-fenêtre du bas de la fenêtre courante.



Les images: *MATLAB* est capable de lire ou écrire des images dans les formats suivants (bmp, pcx, tiff, jpeg, hdf,xwd) On peut alors lire une image avec

```
> X = imread(nomfichier, format)
```

De même pour écrire on utilise la fonction `imwrite`.
Les commandes

```
> pcolor(y)
> image(y)
```

permettent d'afficher une image en associant chaque valeur de y, qui correspond à un point du dessin à une échelle de valeurs sur une échelle de couleurs. Il y des échelles de couleur prédéfinies que l'on invoque à l'aide de la commande

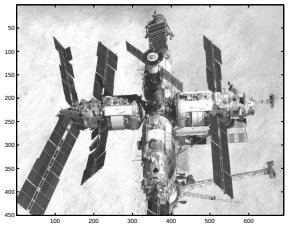
```
> colormap(nom_echelle)
```

où `nom_echelle` est le nom d'une échelle de couleur existante comme *hot, grayscale, cold ....* On peut bien évidemment modifier les échelles de couleurs.

Exemple:

```
> X = imread('mir_sts89.jpg','jpeg');
> image(X)
```

affiche cette image de la station MIR.



16 Impression de figures *MATLAB* sur du papier

Cette opération s'effectue à l'aide de la commande *print*

```
> print
```

envoie le dessin de la fenêtre graphique courante sur l'imprimante.

```
> print nom_fich
```

sauve le contenu de la fenêtre graphique courante en format postscript dans un fichier appelé `nom_fich`. Il existe des options pour définir des formats postscript particuliers.
En général il faut demander du postscript de niveau 2,

```
> print -dps2 nom_fich
```

Lorsque les dessins sont à inclure dans un document LaTeXil faut sauver le graphique en mode encapsulé postscript,

```
> print -deps2 nom_fich
```

17 La programmation avec *MATLAB*

17.1 Les contrôles de flot

- FOR

  ```
  for v=expression
  instruction
  end
  ```

  elle permet d'exécuter une opération pour un certain nombre d'éléments. `v` est une matrice (la plupart du temps un vecteur). `instruction` est effectué pour tous les éléments de `v` si `v` est un vecteur ou pour toutes les colonnes ce `v` si `v` est une matrice. Pour une efficacité maximale il faut essayer de faire le plus possible les opérations de manière vectorielle.

  Exemple

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

- WHILE

  ```
  while expression
  instruction
  end
  ```

  elle permet d'exécuter une instruction tant qu'un critère est vérifié.

  Exemple Calcul du plus petit entier n tel que $n!$ a plus de 100 chiffres :

  ```
  while prod(1:n)<100
  n=n+1;
  end
  n
  ```

- Les opérations conditionelles avec IF. La syntaxe est la suivante

  ```
  if condition
  instruction 1
  elseif
  instruction 2
  else
  instruction 3
  end
  ```

- L'instruction SWITCH permet d'éviter des if-elseif trop complexes

```
switch expression
    case expr1,
        instructions
    case expr2,
        instructions
    case {expr3,expr4,expr5,...}
        instructions
    otherwise
        instructions
end
```

- L'instruction BREAK permet de sortir d'une boucle

  Exemple

```
while 1
n=input('entrez n, fin si n negatif');
if n<= 0
break
else
instruction
end
end
```

  Notez que la première instruction `while 1` correspond à l'instruction toujours vraie : 1 est l'opérateur logique *true*.

17.2 Les fichiers .m (M-files)

Les fichiers .m contiennent des instructions *MATLAB* . Il y en a de deux sortes :

- Les fichiers *commande* qui contiennent une suite de commandes *MATLAB* .
- Les fichiers *fonction* qui définissent une nouvelle fonction *MATLAB* .

Les fichiers *fonction* permettent de créer de nouvelles fonctions *MATLAB* personnelles qui peuvent alors être exécutées comme les fonctions prédéfinies. Pour exécuter un fichier .m il suffit de taper son nom (sans le .m).

Exemple Supposons que le fichier *carre.m* contienne

```
for i=1:10
y=i^2
end
```

alors

```
> carre
```

affiche les carrés des dix premiers entiers.

Remarque 17.7 S'il y a des calculs que l'on désire faire à chaque fois que l'on lance *MATLAB* , on peut les mettre dans un fichier que l'on appellera *startup.m* que l'on place dans le répertoire d'où on lance *MATLAB* . Ce fichier est automatiquement exécuté à chaque fois que l'on démarre *MATLAB* . ■

Les fichiers définissant une fonction ont une première ligne de la forme

```
function y=nom_fonc(x)
```

Les variables utilisées dans une fonction sont locales et sont effacées à la fin de l'exécution de la fonction. Cependant, on peut imposer à des variables définies dans une fonction d'être globales à l'aide de la commande *global*

Exemple

```
function y=expalpha(x)
% Calcule exp(alpha*x)
```

```
global ALPHA
y=exp(ALPHA*x)
```

La variable ALPHA est globale, elle est donc connue à l'extérieur de la fonction et on peut ainsi modifier sa valeur sans toucher à la fonction.

Attention: Supposons que l'on definisse la variable ALPHA par

> ALPHA = 5

La variable ALPHA sera considérée comme une variable locale de *l'espace de travail*. Une execution de *expalpha* produira

```
ans =

     []
```

car la variable ALPHA n'est pas définie à l'intérieur de la fonction *expalpha*. La parade consiste a définir préalablement ALPHA par

> global ALPHA

Par convention, on note les variables globales avec des majuscules. D'autre part, une fonction peut avoir plusieurs arguments d'entrée ou de sortie.

Remarque 17.8 Comme nous l'avons vu dans la section 5, il est recommandé de documenter les fonctions que l'on crée pour pouvoir utiliser l'aide en ligne sur elles. Pour cela, il suffit d'écrire la documentation en commentaires sur des lignes contigues à partir de la deuxième. Une ligne de commentaires commence par le symbole %. ■

17.3 Comment augmenter la vitesse et la place mémoire

Il y a essentiellement deux idées fondamentales à retenir à ce sujet :

1. Ce qui va le plus vite dans *MATLAB* , ce sont les opérations vectorielles ou matricielles qui sont plus rapides notamment que les boucles scalaires.

   Exemple Il y a deux manières de procéder pour calculer le sinus de tous les termes du vecteur `t=0:0.01:10`. On peut utiliser une boucle scalaire :

   ```
   i=0;
   for t=0:.01:10
   i=i+1
   y(i)=sint(t);
   end
   ```

   ou faire une opération vectorielle :

   ```
   t=0:.01:10
   y=sint(t);
   ```

   La deuxième méthode est beaucoup plus efficace dans *MATLAB* .

2. Autant que possible faire l'allocation des vecteurs à l'avance, sinon *MATLAB* doit redimensionner un vecteur à chaque fois qu'il s'aggrandit, ce qui est coûteux en temps de calcul.

   Exemple

   ```
   y=zeros(1:100);
   for i=1:100
   y(i)=det(X^i);
   end
   ```

   La première ligne initialise le vecteur `y` à 0 et en même temps lui alloue la place mémoire nécessaire.

18 Les matrices creuses

La plupart des matrices que l'on rencontre lors de la résolution numérique d'équations aux dérivées partielles sont très grandes (n peut atteindre plusieurs millions pour des problêmes 3D), de sorte qu'il est impossible de stocker tous les termes de la matrice. Mais on sait que la plus grande partie des termes est nulle, d'où la terminologie de matrice creuse. Une matrice creuse est une matrice avec beaucoup de zéros. D'autre part même pour des matrices de taille plus raisonnable, on a intérêt à tirer avantage du fait qu'une matrice est creuse pour accélérer les calculs. En effet il est inutile de faire effectuer à l'ordinateur des multiplications par 0 car on connait déjà le résultat ! *MATLAB* contient des fonctions de stockage "creux", c'est-à-dire ne stockant que les termes non nuls et les informations pour y accéder.

On transforme le stockage plein d'une matrice A en un stockage creux avec la commande

```
> sparse(A)
```

sparse matrix est le terme anglais pour "matrice creuse". L'opération inverse se fait avec

```
> full(A)
```

On peut visualiser graphiquement les termes non nul d'une matrice creuse avec

```
> spy(A)
```

Pour créer une matrice creuse on définit deux vecteurs d'entiers i et j, un vecteur de réels s tous les trois de même longueur l et deux entiers m et n. Alors

```
> A=sparse(i,j,s,m,n)
```

définit la matrice creuse de dimension $m \times n$ et pour tous k entre 0 et l $A(i(k), j(k)) = s(k)$, tous les autre termes de A étant nuls. On peut aussi taper simplement

```
> sparse(i,j,s)
```

alors par défaut $m = max_k(i(k))$ et $n = max_k(j(k))$.

Exemple

```
A=sparse([1 2 3],[3 1 2], [11 22 33])
```

définit un stockage creux de la matrice

$$\begin{pmatrix} 0 & 0 & 11 \\ 22 & 0 & 0 \\ 0 & 33 & 0 \end{pmatrix}$$

Une matrice A creuse étant donnée, on peut trouver ses éléments non nuls à l'aide de la commande *find*

```
[i,j,k]=find(A)
```

donne les vecteurs i, j et k qui correspondent aux éléments non nuls avec leurs indices.