Notes introductives à Matlab

P. Ciarlet, E. Lunéville
ENSTA/UMA

Le logiciel **Matlab** consiste en un langage interprété qui s'exécute dans une fenêtre dite d'*exécution.* L'intérêt de **Matlab** tient, d'une part, à sa simplicité d'utilisation : pas de compilation, déclaration implicite des variables utilisées et, d'autre part, à sa richesse fonctionnelle : arithmétique matricielle et nombreuses fonctions de haut niveau dans divers domaines (analyse numérique, statistique, commande optimale, représentation graphique, ...). Il est à noter que toutes les commandes sont en anglais et l'aide en ligne également !
On peut utiliser **Matlab**, soit en mode *en ligne*, c'est-à-dire saisir des commandes dans la fenêtre d'exécution au fur et à mesure, soit en mode *programmation*, en écrivant dans des fichiers séparés *(*.m)* l'enchaînement des commandes. Ces fichiers s'appellent des *scripts* et on les construit à l'aide de n'importe quel éditeur de texte (par exemple *emacs*, …). Le mode en ligne permet d'obtenir des résultats simples qui ne sont pas sauvegardés. Le mode programmation, quant à lui, permet de développer des applications très complexes.

1. Lancement de Matlab

Lors de son lancement (via la commande **Matlab** par exemple) la fenêtre d'*exécution* s'ouvre. Il est alors possible d'exécuter différents types de commandes dans cette fenêtre, par exemple[*], la commande fondamentale d'aide en ligne :

```
help

HELP topics:

matlab\general       -  General purpose commands.
matlab\ops           -  Operators and special characters.
matlab\lang          -  Programming language constructs.
matlab\elmat         -  Elementary matrices and matrix manipulation.
matlab\elfun         -  Elementary math functions.
matlab\specfun       -  Specialized math functions.
matlab\matfun        -  Matrix functions - numerical linear algebra.
matlab\datafun       -  Data analysis and Fourier transforms.
matlab\polyfun       -  Interpolation and polynomials.
matlab\funfun        -  Function functions and ODE solvers.
matlab\sparfun       -  Sparse matrices.
matlab\graph2d       -  Two dimensional graphs.
matlab\graph3d       -  Three dimensional graphs.
matlab\specgraph     -  Specialized graphs.
matlab\graphics      -  Handle Graphics.
matlab\uitools       -  Graphical user interface tools.
matlab\strfun        -  Character strings.
matlab\iofun         -  File input/output.
matlab\timefun       -  Time and dates.
```

[*] dans tous les exemples qui suivent, apparaît après la commande le résultat de cette commande

```
matlab\datatypes     -  Data types and structures.
matlab\winfun        -  Windows Operating System Interface Files (DDE/ActiveX)
matlab\demos         -  Examples and demonstrations.
toolbox\tour         -  MATLAB Tour
toolbox\local        -  Preferences.

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

La commande suivante permet d'affiner l'aide sur les fonctions mathématiques élémentaires :

```
help elfun
```

```
  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
    tanh        - Hyperbolic tangent.
    atan        - Inverse tangent.
    atan2       - Four quadrant inverse tangent.
    atanh       - Inverse hyperbolic tangent.
    sec         - Secant.
    sech        - Hyperbolic secant.
    asec        - Inverse secant.
    asech       - Inverse hyperbolic secant.
    csc         - Cosecant.
    csch        - Hyperbolic cosecant.
    acsc        - Inverse cosecant.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acoth       - Inverse hyperbolic cotangent.

  Exponential.
    exp         - Exponential.
    log         - Natural logarithm.
    log10       - Common (base 10) logarithm.
    log2        - Base 2 logarithm and dissect floating point number.
    pow2        - Base 2 power and scale floating point number.
    sqrt        - Square root.
    nextpow2    - Next higher power of 2.

  Complex.
    abs         - Absolute value.
    angle       - Phase angle.
    conj        - Complex conjugate.
    imag        - Complex imaginary part.
    real        - Complex real part.
    unwrap      - Unwrap phase angle.
    isreal      - True for real array.
    cplxpair    - Sort numbers into complex conjugate pairs.

  Rounding and remainder.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
```

```
    mod         - Modulus (signed remainder after division).
    rem         - Remainder after division.
    sign        - Signum.
```

permettant ainsi de voir toutes les fonctions mathématiques élémentaires dont dispose Matlab. On peut maintenant préciser la recherche si l'on veut avoir une idée plus précise de la fonction **log** par exemple :

```
help log
```

```
 LOG    Natural logarithm.
    LOG(X) is the natural logarithm of the elements of X.
    Complex results are produced if X is not positive.

    See also LOG2, LOG10, EXP, LOGM.
```

Remarque : les commandes Matlab doivent toujours être tapées en **minuscules** même si dans l'aide en ligne elles apparaissent en majuscules.

♦ Quelques commandes d'environnement importantes

Pour que Matlab fonctionne correctement et en particulier retrouve vos scripts *.m, Matlab met à votre disposition plusieurs commandes d'environnement d'inspiration Unix.

path : permet de savoir quels sont les dossiers auxquels Matlab a accès et de spécifier de nouveaux dossiers Unix où se trouvent vos ressources personnelles. Par ailleurs, pour référencer un nouveau dossier, taper :

addpath ~/mesfichiersmatlab

indiquant à Matlab qu'il peut trouver des scripts dans le dossier **~/mesfichiersmatlab** durant la session en cours.

cd : positionne Matlab dans un dossier Unix, par exemple :

cd ~/mesfichiersmatlab

prenant en priorité les scripts se trouvant dans ce dossier**.**

dir ou **ls** : permet de faire la liste des objets du dossier courant, par exemple la commande suivante :

```
ls
```

```
.               coursIntro.bak  explicite.m.old transport.m
..              coursIntro.tex  oldexplicite.m  u0.m
burgers.m       explicite.m     predicteur.m
```

En outre, voici les deux principales commandes permettant d'effacer les objets générés par Matlab (variables, figures, …) :

clear all : efface tous les objets en mémoire

clf : détruit les figures

Pour les autres commandes d'environnement faire **help general**.

♦ Exécuter un script

Si *monscript.m* est un script Matlab que vous avez écrit et qui est accessible (par *path* ou *cd*) , il suffira de saisir dans la fenêtre d'exécution la commande :

monscript

Par défaut, Matlab inscrit les résultats à la suite de la commande.

Si l'on ne désire pas voir le résultat d'une commande il suffit de terminer cette commande par **;** :

monscript;

Attention à ne pas donner à vos scripts le nom d'une commande prédéfinie !

2. Les variables sous Matlab

Matlab gère les nombres entiers, réels, complexes, les chaînes de caractères ainsi que les tableaux de nombres de façon transparente. Il n'est pas utile de déclarer le type de la variable que l'on manipule, y compris les tableaux. Par ailleurs, toutes les variables utilisées restent présentes en mémoire et peuvent être rappelées. Ainsi les instructions suivantes, déclarent les variables lors de leur affectation :

```
a=1

a =
     1

b=1.01

b =
    1.0100

X=1.0e+05

X =
      100000

nom='    mon nom'

nom =
    mon nom

c=1+2i

c =
   1.0000 + 2.0000i
```

la constante **i** est le nombre imaginaire prédéclaré, de même que certaines constantes (**e,pi,**...).

On déclare un vecteur colonne de la façon suivante :

```
u=[1;3;-1]

u =
     1
     3
    -1
```

un vecteur ligne de la façon suivante :

```
v=[1,3,-1]

v =
     1     3    -1
```

et une matrice d'ordre 3x2 :

```
A=[1,2 ; -1, 3; 4, 0]

A =
     1     2
    -1     3
     4     0
```

la '**,**' sert à séparer les éléments d'une ligne et '**;**' les éléments colonnes. En fait, on peut remplacer la '**,**' par un espace pour améliorer la lisibilité :

```
vb=[1 3 -1]

vb =
     1     3    -1
```

Pour spécifier un élément d'un vecteur, d'une matrice, on utilise la syntaxe suivante :

```
u(2)

ans =
     3

v(3)

ans =
    -1

A(3,2)

ans =
     0
```

L'utilisation d'indice hors limite provoque une erreur, comme le montre cet exemple :

```
A(3,3)

???  Index exceeds matrix dimensions.
```

On peut se servir de raccourcis bien utiles et plus efficaces pour remplir des vecteurs ou des tableaux. En voici quelques-uns :

```
u1=1:10
```
(incrémentation automatique de 1 à 10 avec pas de 1)

```
u1 =
     1     2     3     4     5     6     7     8     9    10
```

```
v1=1:2:10
```
(incrémentation automatique de 1 à 10 avec pas de 2)

```
v1 =
     1     3     5     7     9
```

```
Id3=eye(3)
```
(matrice identité d'ordre 3)

```
Id3 =
     1     0     0
     0     1     0
     0     0     1
```

```
Un=ones(2)
```
(matrice constituée de 1 d'ordre 2)

```
Un =
     1     1
     1     1
```

```
Z=zeros(2,3)
```
(matrice nulle d'ordre 2x3)

```
Z =
     0     0     0
     0     0     0
```

De même, il existe des syntaxes particulières permettant d'extraire des lignes ou des colonnes de matrices :

```
A1=[11 12 13;21 22 23;31 32 33]

A1 =
    11    12    13
    21    22    23
    31    32    33
```

```
A1(:,1)
```
(colonne 1 de la matrice A1)

```
ans =
    11
    21
    31
```

```
A1(2,:)
```
(ligne 2 de la matrice A1)

```
ans =
    21    22    23
```

Les erreurs de dimensions des objets matriciels sont une des principales difficultés que rencontre le débutant. Pour vérifier ces tailles, on pourra utiliser la commande **size** :

```
size(A1)
```

```
ans =
     3     3

size(u)

ans =
     3     1
```

3. Opérations élémentaires sous Matlab

Les opérations sur les scalaires sont standards : addition **+**, soustraction **-**, multiplication *****, division /, puissance ^. La racine carrée s'obtient par la function **sqrt**. On dispose de toutes les fonctions usuelles sur les scalaires : faire **help elfun** pour de plus amples détails. Attention, les fonctions peuvent renvoyer des complexes même dans des situations anodines :

```
sqrt(-1)

ans =
        0 + 1.0000i

acos(2)  (function arc cosinus, ici on a affaire au prolongement dans le plan complexe de cette fonction !)

ans =
        0 + 1.3170i
```

En ce qui concerne les vecteurs et matrices ces opérateurs se prolongent au sens du calcul vectoriel et matriciel. En particulier, il faut veiller à la compatibilité des tailles des objets entre eux ! Voici quelques exemples :

```
u=[1 2 3]

u =
     1     2     3

v=[-1 1 1]

v =
    -1     1     1

w=u+v         (addition)


w =
     0     3     4

ut=u'         (transposition d'un vecteur ligne ou colonne)

ut =
     1
     2
     3
```

```
ut2=[ut ut]
```
(Concaténation en ligne de deux vecteurs colonnes qui donne une matrice 3x2)

```
ut2 =
     1     1
     2     2
     3     3
```

```
ut3=[ut; ut]
```
(Concaténation en colonne de deux vecteurs colonnes qui donne un vecteur 6x1)

```
ut3 =
     1
     2
     3
     1
     2
     3
```

```
ps=v*ut
```
(Produit qui conduit au produit scalaire)

```
ps =
     4
```

```
M=ut*v
```
(Produit qui conduit à une matrice)

```
M =
    -1     1     1
    -2     2     2
    -3     3     3
```

```
L=M+2*eye(3)

L =
     1     1     1
    -2     4     2
    -3     3     5
```

```
y=L\ut
```
(Résolution du système linéaire L.y=ut)

```
y =
    0.1667
    0.3333
    0.5000
```

```
e=u/L'
```
(Résolution du système linéaire e.L' =u)

```
e =
    0.1667    0.3333    0.5000
```

On prendra garde au sens de la division. Si la matrice n'est pas inversible, un message vous prévient.

On peut effectuer des opérations tensorielles sur les vecteurs et matrices par l'adjonction d'un . à l'opérande : par exemple le produit tensoriel de deux vecteurs colonne

```
ut.*y

ans =
```

```
    0.1667
    0.6667
    1.5000
```

et l'élévation à la puissance composante par composante

```
ut.^y

ans =
    1.0000
    1.2599
    1.7321
```

De même, Matlab autorise l'utilisation de toutes les fonctions scalaires dans un contexte vectoriel. Ainsi, si **h** est un vecteur de dimension **n**, **sin(h)** sera un vecteur de même dimension :

```
h=0:pi/4:pi

h =
         0    0.7854    1.5708    2.3562    3.1416

sin(h)

ans =
         0    0.7071    1.0000    0.7071    0.0000
```

Pour les nombreuses opérations sur les matrices (inverse, puissance, trace, déterminant, factorisation, ...) faire **help elmat** et **help matfun**.

Pour ce qui est des opérations sur les chaînes de caractères, ces dernières étant considérées comme des vecteurs ligne de caractères ascii, la concaténation de deux chaînes s'effectuera de la façon suivante :

```
c1='texte'

c1 =
texte

c2=' et suite de texte'

c2 =
 et suite de texte

c3=[c1 c2]

c3 =
texte et suite de texte
```

pour les autres opérations sur les chaînes de caractères voir **help strfun**.

4. Structure creuse des matrices

Une matrice creuse est une matrice présentant un grand nombre d'éléments nuls qu'il n'est pas nécessaire de stocker, permettant de gagner à la fois de la place mémoire et du temps de calcul. Matlab gère de façon transparente les matrices creuses à l'aide de pointeurs que nous ne décrirons pas ici.

Contrairement aux variables classiques on doit déclarer explicitement le type **sparse** pour spécifier qu'une matrice est creuse :

```
AC=sparse(1000,2000)

AC =
   All zero sparse: 1000-by-2000
```

qui crée une matrice 1000x2000 initialisée à 0. En fait la matrice n'occupe pas de place en mémoire (hormis la place occupée par le pointeur) mais la commande suivante renvoie la valeur nulle

```
AC(1,1)

ans =
     0
```

Pour affecter un élément de la matrice, on procède comme d'habitude. Par exemple,

```
AC(1,1)=1

AC =
   (1,1)        1
```

On remarquera que le résultat affiché comporte à la fois la position et la valeur des éléments non nuls.

```
AC(2,3:2:8)=5

AC =
   (1,1)        1
   (2,3)        5
   (2,5)        5
   (2,7)        5
```

L'affectation d'un élément à 0 ne modifie pas la structure creuse :

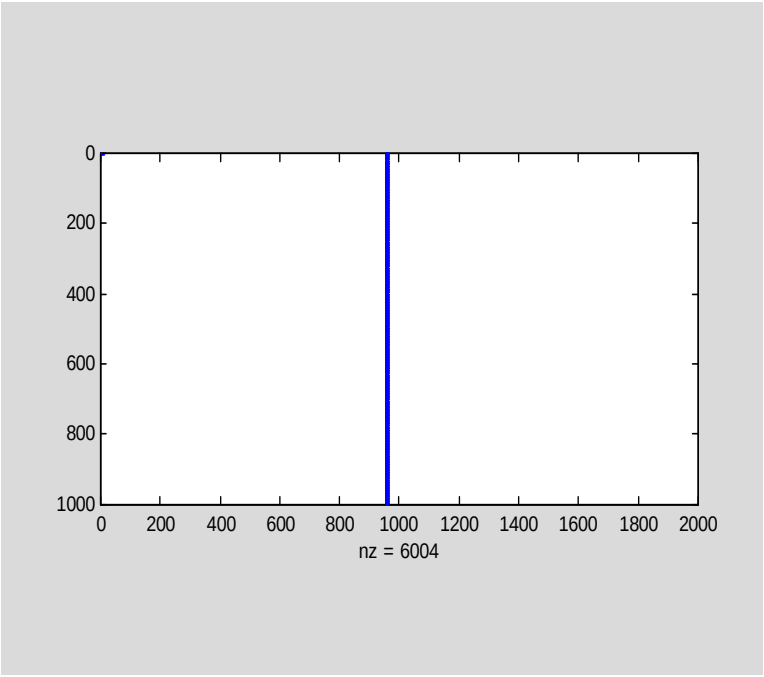
```
AC(1,3)=0

AC =
   (1,1)        1
   (2,3)        5
   (2,5)        5
   (2,7)        5
```

Remplissons complètement les colonnes 955 à 960 sans afficher le résultat :

```
AC(1:1000,955:960)=1;
```

On visualise sur une figure la structure creuse de la matrice AC à l'aide de la commande **spy :**

```
figure
spy(AC)
```



nz désigne le nombre d'éléments non nuls de la matrice.

La commande **figure** est optionnelle.

Toutes les opérations standard sur les matrices s'appliquent aux matrices creuses. Pour des informations supplémentaires sur le type creux taper **help sparfun.**

5. Contrôle de flux

Dans cette section, on indique les principales syntaxes permettant de contrôler le flux d'exécution d'une application Matlab. On les utilise essentiellement en mode programmation !

♦ Opérateurs booléens

Avant de décrire la syntaxe du test sous Matlab, indiquons les principaux opérateurs de relation ainsi que les opérateurs booléens qu'utilise Matlab.

<	strictement inférieur à
<=	inférieur ou égal à
>	strictement supérieur à
>=	supérieur ou égal à
==	égal à
~=	différent de
&	et logique (and)
|	ou logique (or)
~	non logique (not)

Le résultat d'un test est un booléen qui, sous Matlab, prend la valeur 1 pour vrai et 0 pour faux. Par exemple, on a les résultats suivants :

```
r=1<2

r =
     1

r=~((1>2)|(0~=0))          (traduction Matlab de l'expression logique : non (1>2 ou 0≠0))

r =
     1
```

Il existe d'autres fonctions booléennes, par exemple **xor**, **isfinite**, **isnan**, **isinf**,… dont on trouvera la description en faisant **help ops**.

♦ Syntaxe du test (**if**)

if *expression booléenne*
  instructions
end

if *expression booléenne*
  instructions
else
  instructions
end

if *expression booléenne*
  instructions
elseif *expression booléenne*
  instructions
else
  instructions
end

♦ Syntaxe du branchement (**switch**)

switch *expression* (*expression* est un scalaire ou une chaîne de caractères)
  case *value1*
    instructions (*instructions* effectuées si *expression=value1*)
  case *value2*
    instructions
  ...
  otherwise
    instructions
end

♦ Syntaxe de boucle (**while** et **for**)

while *expression*
  instructions
end

for *indice=debut:pas:fin* (si le pas n'est pas précisé, par défaut il vaut 1)
  instructions
end

Pour sortir d'un test ou d'une boucle, on utilise la commande **break** (voir l'aide en ligne).

6. Programmation et utilisation des fonctions

La notion de fonction existe sous Matlab. Sa syntaxe est la suivante :

function [*args1,args2,...***] =** nomfonction**(***arge1,arge2,...***)**
instructions

args1,args2,... sont les arguments de sortie de la fonction et peuvent être de n'importe quel type
arge1,arge2,... sont les arguments d'entrée de la fonction et peuvent être de n'importe quel type
instructions est un bloc d'instructions quelconque devant affecter les arguments de sortie *args1,args2,...*

Lorsqu'il n'y a qu'un seul argument de sortie, on peut utiliser la syntaxe plus simple :

function *args* = nomfonction**(***arge1,arge2,...***)**

Pour appeler une fonction on opère de la façon suivante :

[*vars1,vars2,...***] =** nomfonction**(***vare1,vare2,...***)**

avec compatibilité des variables d'entrées *vare1,vare2,...* avec les arguments d'entrée *arge1,arge2,...* et compatibilité des variables de sorties *vars1,vars2,…*, si elles ont déjà été créées, avec les arguments de sortie *args1,args2,....*

Remarque : il n'est pas obligatoire de fournir tous les arguments d'entrée et de sortie lors de l'appel d'une fonction, mais ceux que l'on omet doivent être les derniers des listes d'entrée ou de sortie. Ainsi, supposons que *nomfonction* soit une fonction à 2 arguments d'entrée et 2 arguments de sortie, on peut alors effectuer l'appel suivant :

[*vars1***]=** nomfonction**(***vare1***)** mais pas l'appel : **[***vars2***]=** nomfonction**(***vare2***)**.

Bien entendu, il faut gérer l'omission des arguments d'entrée. Matlab met à votre disposition la variable **nargin** qui indique le nombre d'arguments en entrée lors de l'appel de la fonction. Voici un exemple de calcul d'un produit scalaire ou d'une norme au carrée, illustrant son emploi :

```
function r = psnorm2(a,b)      (a,b sont des vecteurs colonnes)
 if (nargin==1)
   r=a'*a;
 elseif (nargin==2)
   r=a'*b;
 end
```

psnorm2(u,v) renvoie le produit scalaire et *psnorm2(u)* renvoie la norme au carrée de *u*. Remarquer que *psnorm2(u,u)* renvoie également la norme au carré de *u*.

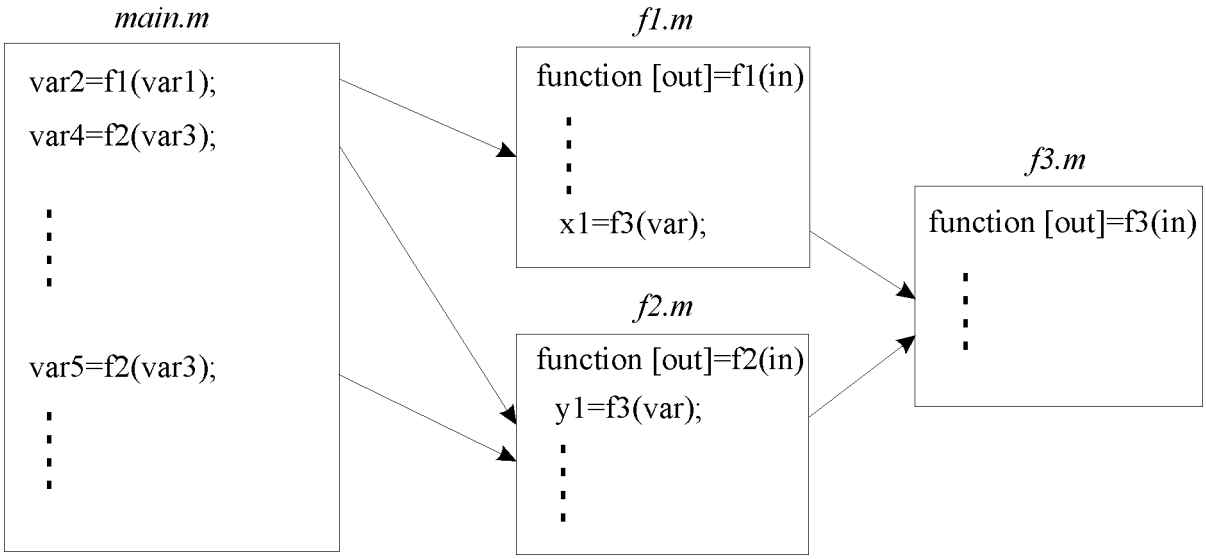
Bien qu'il soit parfois dangereux d'utiliser cette possibilité, il est important de la connaître car de nombreuses fonctions natives de Matlab en font usage.

♦ Organisation d'un programme

Toute bonne programmation repose sur l'écriture d'un *script principal* qui fait appel à des fonctions autonomes. Cela permet :

- d'améliorer la lisibilité
- de tester indépendamment des parties de programmation
- d'augmenter le degré de généralité (utilisation d'une même fonction à divers endroits du programme, voire réutilisation des fonctions dans d'autres applications)

Rappelons que vous pouvez écrire ces *scripts* à l'aide de n'importe quel éditeur de texte (emacs,…) et donc les sauvegarder !



Comme dans l'exemple ci-dessus, le nom du *script* attaché à une fonction doit obligatoirement porter le nom de la fonction avec l'extension *.m*.

Par défaut, une variable n'est connue que dans le *script* dans lequel elle a été définie. En particulier, les variables du *script principal* ne sont pas connues dans les autres *scripts*. On doit donc transmettre en arguments d'une fonction toutes les variables dont on a besoin pour son exécution. Dans certains cas, on peut également utiliser la notion de variable globale qui permet de rendre visible des variables d'un *script* à l'autre. On la déclare comme globale dans le *script principal* ainsi que dans les *scripts* dans lesquels on désire l'utiliser à l'aide de la commande :

global *varg*

♦ Fonction comme argument d'une fonction

Pour clore cette partie sur les fonctions, mentionnons l'existence de commandes permettant d'évaluer une chaîne de caractères comme une commande Matlab. La commande **eval** s'utilise ainsi :

```
comd='s1=sin(pi/2)'

comd =
s1=sin(pi/2))

eval(comd)

s1 =
     1
```

La commande **feval** permet quant à elle de transférer, en temps que chaîne de caractères, le nom d'une fonction que l'on veut évaluer :

```
fon='sin'

fon =
sin

s1=feval(fon,pi/2)     (derrière le nom de la fonction à évaluer, apparaissent les arguments)

s1 =
     1
```

7. Lecture et écriture

Par défaut, toute commande exécutée produit un résultat qui apparaît dans la fenêtre d'exécution à la suite de la commande. On peut empêcher l'affichage du résultat en terminant la commande par '**;**'. Ainsi, on a :

```
H=[1 2 3]                (Résultat affiché)

H =
     1     2     3

H=[1 2 3];               (Aucun résultat affiché)
```

En mode programmation, l'affichage d'un résultat est exceptionnel. Il sert, essentiellement, à détecter des erreurs et à afficher les résultats finaux. On prendra donc garde à ne pas oublier le '**;**' à la fin de chaque ligne de commande.

♦ Impression à l'écran

La commande standard d'écriture dans la fenêtre d'exécution est **fprintf** qui a la structure générale suivante :

fprintf(*format***,***var1***,***var2***,...)**

où *format* est une chaîne de caractères décrivant le format d'écriture des variables *var1*,*var2*,... que l'on souhaite afficher. Les principaux types de formats d'écriture sont :

%d	entier	%5d : entier de taille 5, par exemple 34562
%f	réel	%5.2f : réel de taille 5 avec 2 chiffres après la virgule, par exemple 32.42
%e	exponentiel	%10.2e : nombre de la forme -21.01e+05
%g	réel double précision	mode automatique de détection entre %e et %f
%s	chaîne de caractères	

Par souci de simplicité, on peut se contenter d'utiliser les formats %d, %f, %e sans spécifier de taille précise. Par ailleurs on dispose de certains opérateurs de mise en forme, par exemple **\n** pour passer à la ligne. Ainsi, on écrira, par exemple :

fprintf('**\n** Convergence en **%d** iterations '**,***it***)**

où *it* désigne une variable contenant un entier.

Pour plus de détails, faire **help fprintf**.

♦ Impression dans un fichier

Il est également possible d'écrire les résultats dans un fichier (et souhaitable lorsqu'il y en a beaucoup). Pour ce faire, on utilise encore la commande **fprintf**, mais en spécifiant un numéro *nfic* associé à un nom de fichier de résultats, nommé ici *ficres*. On effectue les opérations suivantes :

```
fid=fopen(ficres,'rw');                    (ouvre le fichier ficres en mode lecture et écriture)
fprintf(fid,'\n Convergence en %d iterations ',it);  (écrit dans le fichier ficres)
status=fclose(fid);                        (ferme le fichier ficres)
```

L'opération d'ouverture de fichier par la commande **fopen** a échoué si *fid* vaut -1 ; *status* renvoie 0 si l'opération de fermeture par la commande **fclose** est réussie et -1 sinon.

♦ Lecture de données

Afin de lire des données utiles à l'exécution, on peut procéder de deux façons : soit en interrompant l'exécution du programme et en demandant à l'utilisateur de fournir les données, soit en lisant un fichier de données. Cette deuxième solution est bien souvent préférable à la première.

Pour interrompre l'exécution et demander une valeur, on utilise la commande **input,** dont voici un exemple d'utilisation :

```
data=input('Donnez votre valeur (par defaut 0) ');
```

La variable *data* contiendra la réponse envoyée, qui peut prendre n'importe quel type et même la valeur vide **[]** si on a tapé sur *enter*. C'est d'ailleurs par ce moyen que l'on gère les valeurs par défaut :

```
if (data ==[])
    data=0;
end
```

Pour lire des fichiers de données, on utilise la commande **fscanf** dont le principe de fonctionnement est voisin de la commande **fprintf**. Pour lire l'entier *data* dans le fichier *ficdon,* on utilisera la suite de commandes suivante :

```
fid=fopen(ficdon,'r');                     (ouvre le fichier ficdon en mode lecture)
data=fscanf(fid,'%d');                     (lit un entier dans le fichier ficdon)
…
status=fclose(fid);                        (ferme le fichier ficdon)
```

Exemple complet

Pour lire le fichier *data* suivant :

```
173 304 -- nbs,nbt
1 1.0000000E+00 -1.0000000E+00 4
2 1.0000000E+00 -7.9999995E-01 3
3 7.6839125E-01 -8.0331707E-01 0
....
```

```
173 -1.0000000E+00 1.0000000E+00 4
1 173 170 171 2
2 154 142 143 2
3 142 126 127 2
...
304 91 93 109 2
```

on utilisera la fonction ci-dessous en l'appelant avec la commande :

[Nbpt,Nbtri,Coorneu,Refneu,Numtri,Reftri] =**lecmail***(data);*

function *[Nbpt,Nbtri,Coorneu,Refneu,Numtri,Reftri]* =**lecmail***(nomfile)*
fid=**fopen***(nomfile,'r');*
N=**fscanf***(fid,'\%i');}*
Nbpt=N(1);
Nbtri=N(2);
line=**fgets***(fid);* *(passe à la ligne suivante)*
tmp = **fscanf***(fid,'\%f',[4,Nbpt]);*
Coorneu=tmp(2:3,:)';
Refneu=tmp(4,:)';
tmp **= fscanf***(fid,'\%i',[5,Nbtri]);*
Numtri=tmp(2:4,:)';
Reftri=tmp(5,:)';

Pour plus d'informations, faire **help fscanf**, **help fopen** et **help fclose**. Il existe d'autres méthodes de lecture et d'écriture sur fichier : faire **help iofun** pour de plus amples informations.

8. Représentations graphiques sous Matlab

Indiquons quelques fonctionnalités graphiques de Matlab. Elles se séparent naturellement en deux catégories principales : les représentations dans le plan regroupées dans Matlab sous la terminologie **graph2d** et les représentations 2D/3D regroupées sous la terminologie **graph3d**. En outre, les fonctions graphiques très spécialisées (maillage par exemple) sont regroupées dans **specgraph**. Les représentations graphiques sont réalisées dans un environnement figure que vous pouvez contrôler.

♦ L'environnement figure

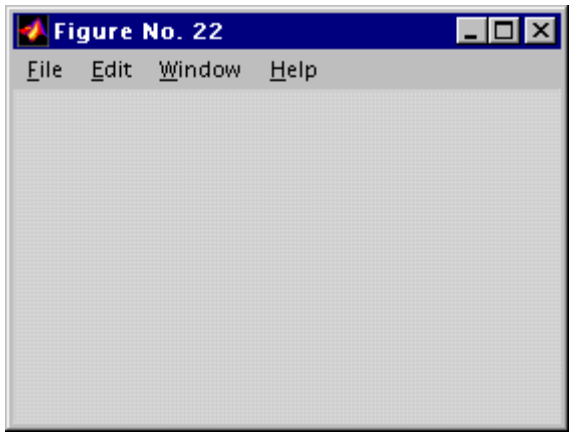
Par défaut, toute commande de représentation graphique déclenche la création d'une fenêtre **figure** nommée *figure1* contenant la représentation. Ceci implique en particulier que toute nouvelle représentation écrase la précédente.
Il est possible de conserver les différentes représentations. Il suffit de faire précéder chaque nouvelle commande de représentation graphique par la commande

figure

qui crée successivement *figure2*, *figure3*, …

On peut spécifier directement le N° de figure :

```
figure(22)
```



On peut également superposer des représentations graphiques sur une même figure en utilisant la commande

hold on

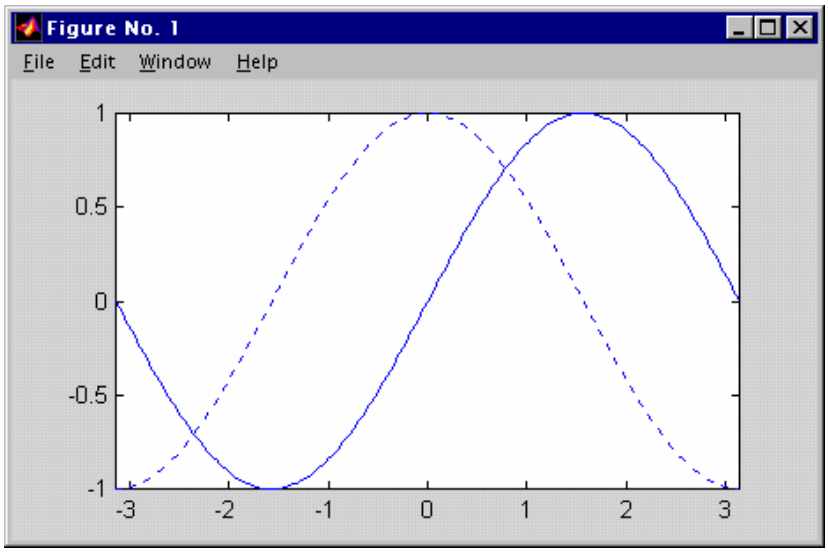
Pour désactiver le mode superposé faire

hold off

♦ Tracé de courbes

Supposons que l'on veuille représenter graphiquement en mode superposé les courbes *sin(x)* et *cos(x)* sur l'intervalle [-π,π] avec 201 points. On exécute alors les commandes suivantes :

```
x=-pi:pi/100:pi;
y=sin(x);z=cos(x);
figure
plot(x,y)
hold on                          (mode superposé)
plot(x,z,':');                   (l'option ':' spécifie un tracé tireté)
axis([-pi pi –1 1]);             (dimensionne les axes)
hold off                         (fin du mode superposé)
```



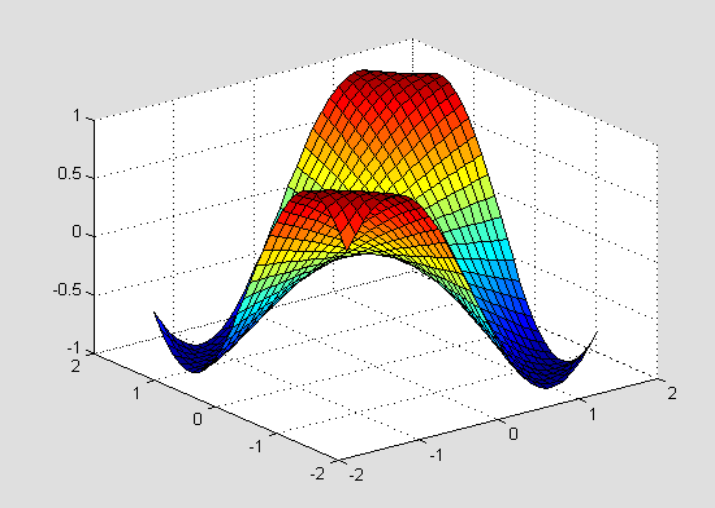
Il existe de nombreuses options pour contrôler l'affichage des courbes (couleur, axe, légende, commentaire, échelle logarithmique...), faire **help plot** et **help graph2d**.

La commande **subplot** permet d'afficher plusieurs représentations graphiques sur une même figure. Voir un exemple d'utilisation au prochain paragraphe.

♦ Représentation graphique 3D

Supposons, par exemple que l'on veuille représenter la surface définie par la fonction *z=sin(xy)* sur le carré suivant [-π/2,π/2]x**[**-π/2,π/2] à l'aide d'une grille de points 31x31. On utilise la séquence de commandes :

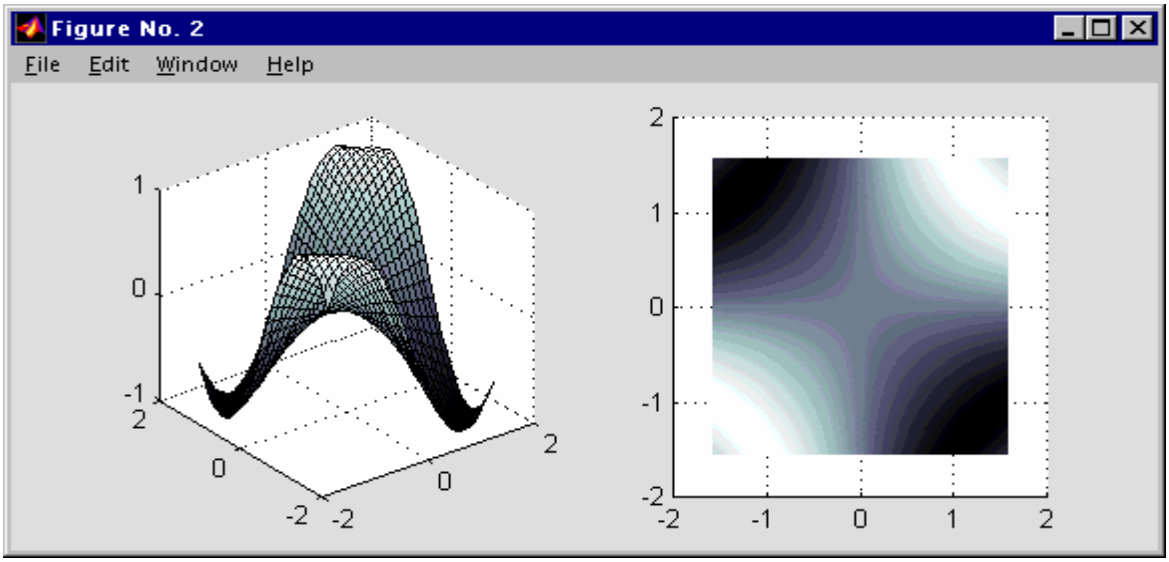
```
[xi,yi]=meshgrid(-pi/2:pi/30:pi/2);     (génération d'une grille régulière)
zi=sin(xi.*yi);
surf(xi,yi,zi,zi);                      (affichage de la surface)
```



On peut modifier l'angle de vue via la commande **view** en spécifiant soit un point d'observation ou deux angles d'élévation. Noter que la commande **view(2)** déclenche directement une vue de dessus. On peut également choisir l'angle de vue à la souris en activant l'option **rotate3d on** et on annule ce mode par la commande **rotate3d off.** De même la commande **zoom in** permet d'effectuer des zooms à la souris, seulement en vue plane!

Il existe par ailleurs de nombreuses commandes permettant de contrôler les palettes de couleurs utilisées pour la représentation graphique, en particulier la commande **colormap**. Signalons une option intéressante permettant de lisser les couleurs : **shading interp**. Pour une description exhaustive de toutes ces possibilités faire **help graph3d**. Illustrons quelques une de ces possibilités au travers de l'exemple précédent :

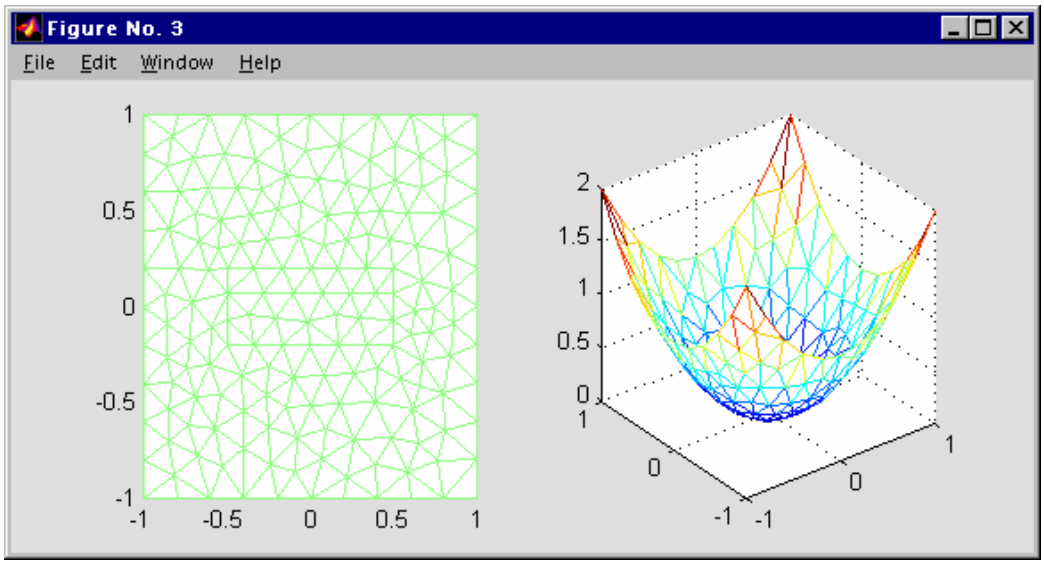
```
figure(2)
subplot(1,2,1);                 (affichage dans la première sous-figure)
surf(xi,yi,zi,zi);
subplot(1,2,2);                 (affichage dans la seconde sous-figure)
surf(xi,yi,zi,zi);
view(2);                        (vue de dessus)
shading interp;                 (lissage des couleurs)
colormap(bone);                 (changement de palette de couleurs)
```



♦ Représentation graphique sur des maillages

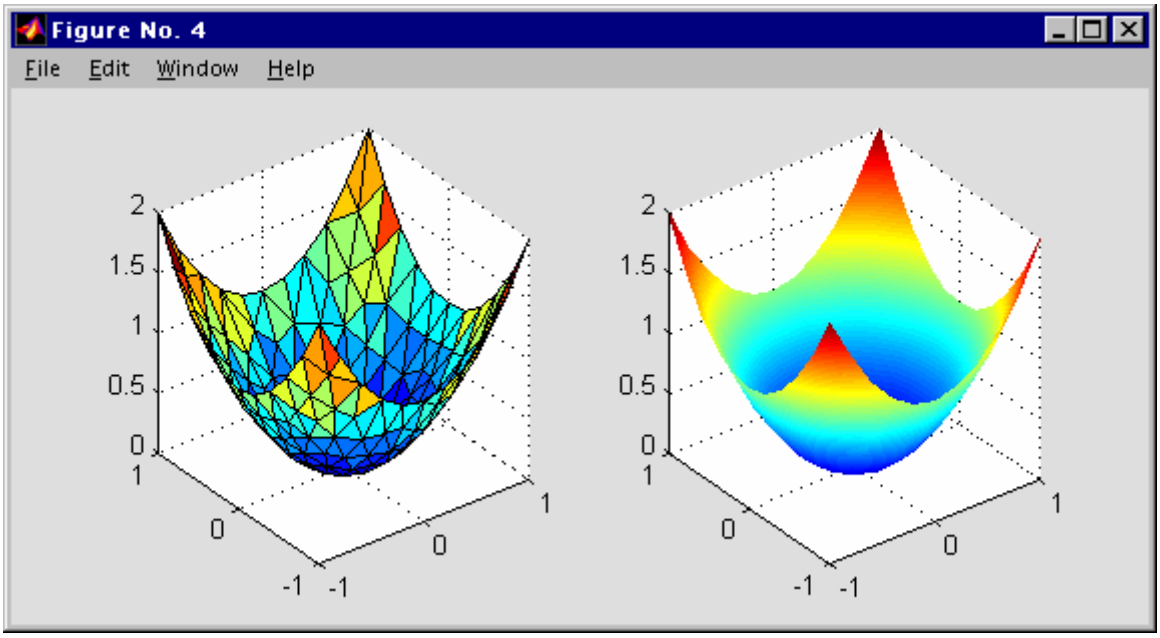
Pour un certains nombres d'applications (calcul par éléments finis et C.A.O. …) , les nappes sont constituées d'un ensemble de triangles. Ces nappes sont décrites dans Matlab par les trois vecteurs *X*, *Y*, et *Z* contenant les coordonnées des sommets de tous les triangles (chaque sommet étant compté une fois et une seule), ainsi que d'un tableau *Numtri* contenant, pour chaque triangle, les numéros de ses sommets. Précisément, les vecteurs colonnes de coordonnées sont de dimension *Nbpt*, et le tableau de correspondance est de dimension *Nbtri* par *3*. En particulier, l'ensemble formé de *Numtri* et des deux vecteurs de coordonnées *X* et *Y* forme ce que l'on appelle un maillage bidimensionnel. Pour représenter un maillage bidimensionnel, on utilise la commande **trimesh** :

```
[Nbpt,Nbtri,Coorneu,Refneu,Numtri,Reftri]=lecmail(data);      (lecture du maillage)
X=Coorneu(:,1); Y=Coorneu(:,2);
figure;subplot(1,2,1);
trimesh(Numtri,X,Y,zeros(Nbpt,1));                           (affichage du maillage)
view(2);
subplot(1,2,2);
Z=X.*X+Y.*Y;                                                 (nappe parabolique)
trimesh(Numtri,X,Y,Z);
```



La commande **trisurf** permet, en plus, de colorier chacun des triangles :

```
figure
subplot(1,2,1);
trisurf(Numtri,X,Y,Z);
subplot(1,2,2);
trisurf(Numtri,X,Y,Z);
shading interp;
```



Il existe de nombreuses autres possibilités, telles que les courbes isovaleurs (commande **contour**), les vecteurs (commande **quiver**), … . On peut également réaliser simplement des animations à l'aide de la commande **movie**. Pour de plus amples détails, faire help **specgraph**.

9. Mise au point d'une application

♦ Debugging

Lors de la mise au point d'une application, on peut avantageusement tirer parti du mode exécution de Matlab pour tester individuellement chaque fonction. En outre, on a accès, dans la fenêtre d'exécution, aux variables du programme principal ; ceci permet de vérifier le contenu des variables.

Malheureusement, on n'a pas accès aux variables locales aux fonctions. Pour contourner cette difficulté, il existe plusieurs méthodes :

- Supprimer les **';'** à la fin de chaque commande pour afficher les résultats intermédiaires (on peut également écrire ces résultats intermédiaires dans des fichiers).

- Exécuter tout ou partie d'une fonction dans la fenêtre d'exécution, en prenant soin de vérifier que toutes les variables nécessaires à cette fonction ont été définies.
- Utiliser la fonction **debug** de Matlab (faire **help debug**).

♦ Quelques erreurs fréquentes

Incohérence entre les dimensions d'objet :

```
A=ones(4,5);
b=1:1:5;
A*b

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

En effet, la commande

```
size(b)

ans =
     1     5
```

montre que le vecteur b est un vecteur ligne, et non un vecteur colonne !

Chaînes de caractères : il ne faut pas oublier de spécifier la chaîne de caractères entre **'…..',** et particulièrement lorsqu'elle est transmise comme argument d'une fonction :

```
upper('erreur')

ans =
ERREUR

upper(erreur)

??? Undefined function or variable 'erreur'.
```

Les nombres complexes : Matlab bascule automatiquement en complexe lorsqu'il rencontre des fonctions dont le résultat n'est pas nécessairement réel. Par conséquent, aucune erreur ne survient lorsqu'on évalue des quantités que l'on pense être réelles *a priori* :

```
x =2;
acos(x)

ans =
        0 + 1.3170i
```

L'arithmétique de Matlab : contrairement à un certain nombre de langages de programmation, Matlab ne stoppe pas lorsqu'il produit un résultat qui n'est pas un nombre (*Inf* = l'infini, ou *NaN* = not a number). Par exemple :

```
x= 1/0
```

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

y=0/0

Warning: Divide by zero.
y =
   NaN

x*y

ans =
   NaN
```

Lorsque cela se produit, il faut remonter à l'origine du problème ! L'instruction **dbstop if naninf** de **debug** provoque un arrêt lors de la première apparition d'un *Nan* ou d'un *Inf*.

Memento

Résolution de systèmes linéaires

- inversion : /, \ et **inv**
- factorisation : **chol**, **lu**, **qr**
- gradient conjugué : **pcg** , **gmres**

Valeurs propres

- calcul de valeurs propres : **eig**
- calcul de valeurs singulières : **svd**
- conditionnement d'une matrice : **cond**, **condeig**

Matrices creuses

- type creux/plein : **sparse**, **full**
- visualisation : **spy**

Minimisation, zeros de fonctions

- minimisation sans contrainte : **mins**
- zéros de fonction : **fzeros**

Equations différentielles, intégration numérique

- Résolution : **ode23**, **ode45**
- Représentation graphique : **odeplot**
- Calcul d'intégrale : **quad**

Eléments finis

- génération de maillage : **trimesh**, **delaunay**

Polynômes

- interpolation : **interp**, **spline**
- racines : **roots**

Transformées de Fourier

- Transformée de Fourier rapide : **fft**
- Convolution : **conv**

Graphiques

- Courbes : **plot, fplot, polar, loglog, bar, pie, contour**
- Courbe gauche : **plot3**
- Surfaces : **surf, mesh, trisurf**