

Matlab pas à pas

M. Bonnivard^{1*}; M. Ersoy^{1†}

¹ Université de Savoie, Laboratoire de Mathématiques, 73376 Le Bourget-du-Lac, France.

12 octobre 2008

Table des matières

1	Introduction	2
2	Prise en main	2
3	Syntaxe du langage, vecteurs et matrices	4
3.1	Vecteurs	6
3.2	Matrices	8
4	Concevoir un code numérique	11
4.1	Structure du programme	11
4.2	Déclaration de fonctions du type <i>M-files functions</i> et <i>Inline functions</i>	12
4.3	Tests et boucles	15
4.3.1	Tests	15
4.3.2	If, for et while	16
5	Représentation de nos résultats	18
5.1	Representation graphique	18
5.2	Lecture/écriture dans un fichier	23
5.2.1	écriture	23
5.2.2	Lecture	24
6	Debugger un programme	24
7	Exemples de programmes	31
7.1	Système prédateur proie	31
7.2	Klein bottle	34
7.3	Résolution d'un problème de valeurs propres par une méthode d'éléments finis.	36

**Matthieu.Bonnivard@etu.univ-savoie.fr*

†*Mehmet.Ersoy@univ-savoie.fr*

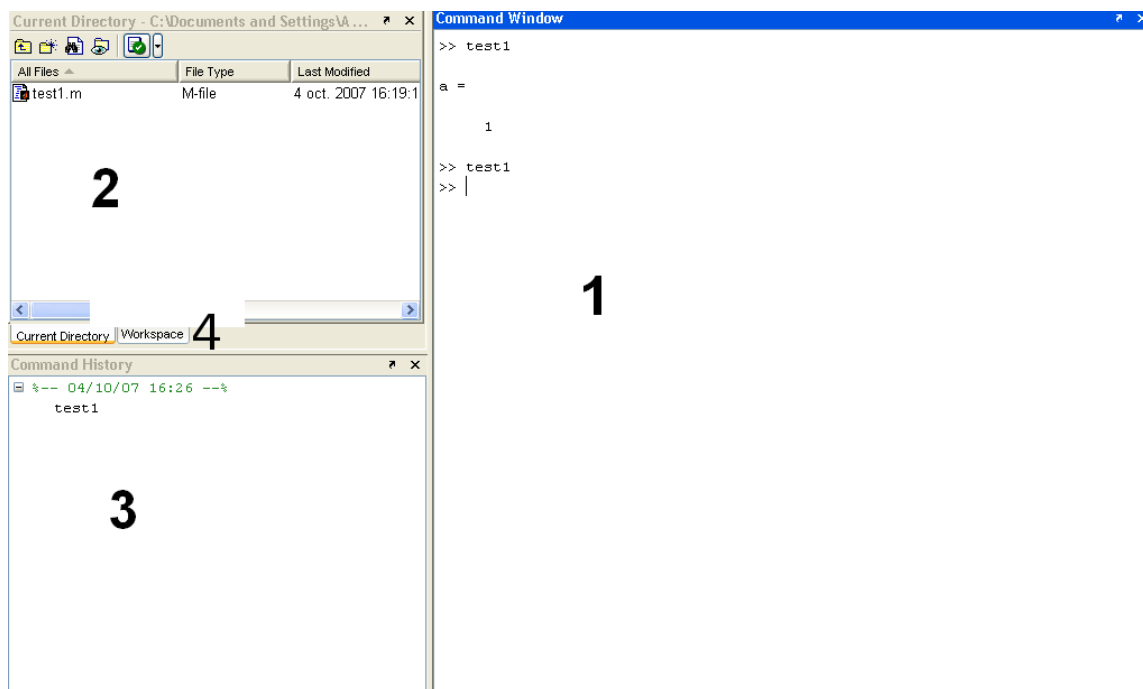
1 Introduction

Matlab est un logiciel commercial destiné au calcul numérique. Il existe deux versions gratuites de Matlab : Scilab et Octave. Octave est un clone dans le sens où les commandes Matlab et Octave sont similaires à quelques nuances près contrairement à Scilab. Il existe différentes versions du logiciel Octave dont QToctave et Octave-Workshop qui sont munies d'une interface graphique que l'on trouve facilement sur internet. Les captures d'écran sont toutes tirées de Matlab 7 et les commandes Octave utilisées nécessitent une version d'Octave supérieure à la version 3.0.

2 Prise en main

N.B. Les commandes utilisées dans ce tutoriel ne sont pas toutes compatibles avec le logiciel Octave. Cependant, on trouve facilement les commandes équivalentes *via* Google (par exemple http://enacit1.epfl.ch/cours_matlab/). Avant de commencer, il est utile de préciser que le meilleur ami d'un utilisateur de Matlab est la commande *help* (à utiliser dans la fenêtre de commande) ou la touche de raccourci F1 vers l'aide en ligne. Il est important de noter que Matlab prend en compte la casse. Par exemple, les variables Casse, casse ou CASSE sont toutes *différentes*. De plus, il est interdit d'utiliser les noms de fonctions ou les mots clefs de Matlab comme nom de variable.

Matlab est muni de l'interface suivante :



L'encadré 1 (COMMAND WINDOW) correspond à la fenêtre de commande. Toute instruction est précédée du symbole >> appelé *prompt*. C'est ici que nous effectuerons nos exécutions. A titre d'exemple, ouvrez une feuille vierge (file → New → M-file ou encore tapez *edit* dans la fenêtre de commande), puis tapez `a = 1` et sauvegardez (racc. ctrl+s) sous le nom de test1.m (par exemple).

Ensuite, tapez `test1` dans la fenêtre de commande et observez. Vous venez d'exécuter votre première ligne de commande. A l'écran apparaît alors

```
>> test1
```

```
a =
```

```
1
```

```
>>
```

Notez que l'instruction précédente se termine par le nombre 1. On peut décider d'afficher ou de ne pas afficher le résultat à l'écran en ajoutant un point virgule en fin d'instruction, comme suit `a=1 ;`. Exécutez à nouveau `test1` et observez.

Ainsi, toute exécution de programme est effectuée à partir de la fenêtre de commande.

L'encadré 2 (CURRENT DIRECTORY) correspond au répertoire courant (ou répertoire de travail) ; c'est l'emplacement des fichiers qui seront exécutés. Ainsi l'exécution de `test1` n'est possible que si ce dernier se trouve dans le répertoire courant. C'est à vous de bien définir votre répertoire de travail en créant un dossier.

Enfin, l'encadré 3 représente l'historique des commandes exécutées. Comme on peut l'observer dans la figure précédente, nous avons exécuté `test1`. Placez-vous maintenant dans la fenêtre de commande et tapez `t` puis appuyez sur la touche tabulation. Magique non ? L'historique permet ainsi d'éviter de réécrire le nom du programme à exécuter ; c'est l'*écriture intuitive*. A présent, tapez `h` puis la touche tab : vous constatez que vous avez accès à toutes les fonctions pré-programmées de Matlab, et aussi aux fichiers que vous avez codés. Essayez `t` puis tab puis `e` puis `s`. Pratique, non ?

L'encadré 4 (WORKSPACE) affiche toutes les variables en cours d'utilisation. Dans la section suivante, nous verrons comment effacer une ou toutes les variables en mémoire.

Le plus important est de savoir utiliser l'aide. Sous Matlab, on accède à l'aide grâce à la touche de raccourci F1 ou en cliquant sur l'onglet help. On trouve ensuite une barre de recherche. Si l'on souhaite obtenir des détails sur une commande dont on connaît le nom, on tape ce nom dans la barre de recherche puis entrée. Si l'on ne connaît pas le nom de la commande, on tape un mot-clé. On peut également accéder à l'aide en ligne de commande avec la commande `help commande` ou `look for MotClef`. Attention, la commande `look for` (bien pratique) ne fonctionne pas sous Octave. Par conséquent soit on utilise `help` pour faire apparaître toutes les fonctions existantes, soit on effectue une recherche sur Google. Cette petite remarque concernant Octave n'est pas valable avec les versions Qt octave ou Octave-Workshop qui bénéficient d'une aide sous forme d'onglets. Cependant, il est tout de même préférable pour un utilisateur d'Octave de connaître quelques lignes de commandes type MS/DOS (pour Windows) et Shell (pour linux : voir http://gershwin.ens.fr/vdaniel/Doc-Locale/Outils-Gnu-Linux/Linux_Doc/Unix-BournShell/). Voici quelques commandes principales :

`cd` change le répertoire de travail en la racine de l'utilisateur,

`cd rep` permet de se placer dans le répertoire rep,

cd .. permet de se placer dans le répertoire parent,

dir permet de visionner en mode console les fichiers et dossiers présent dans le répertoire courant.

3 Syntaxe du langage, vecteurs et matrices

Nous sommes désormais aptes à éditer un fichier et à l'exécuter.

Comme nous l'avons vu, le signe "=" est l'action d'affecter. Nous disposons des quatre opérations élémentaires $+$ $-$ $*$ $/$ pour effectuer des calculs vectoriels et matriciels. Ouvrez le fichier `test1.m` (ou tapez `edit test1` dans la fenêtre de commande) puis complétez le code par

```
b = a;  
c = a+b
```

et exécutez. Matlab nous donne comme réponse

```
c =
```

```
    2
```

```
>>
```

On vient donc d'affecter la valeur 2 à `c` par l'opération `a+b`. Qu'advient-il alors de `a` et `b`? Pour le savoir, tapez `a` dans la fenêtre de commande puis la touche entrée et faites de même pour `b`. Vous obtenez :

```
>> a
```

```
a =
```

```
    1
```

```
>> b
```

```
b =
```

```
    1
```

```
>>
```

On constate que cette suite d'instruction ne modifie pas la valeur affectée à `a` et `b` : Matlab garde donc en mémoire les variables affectées. Si l'on souhaite effacer la valeur de `a` ou `b`, on exécute la commande `clear a`, soit en ligne de commande (dans ce cas, la variable `a` est effacée ponctuellement : lors d'une nouvelle exécution de `test1`, la valeur de `a` sera de nouveau affectée à 1), soit dans le fichier `test1.m` (et dans ce cas la variable `a` sera effacée définitivement).

Essayons

```

>> clear a
>> a
??? Undefined function or variable 'a'.

>> test1

c =

     2

>> a

a =

     1

>>

```

Enfin, la commande `clear var1 var2 var3 ... varn` permet d'effacer toutes les variables var_i , pour $i = 1 \dots n$.

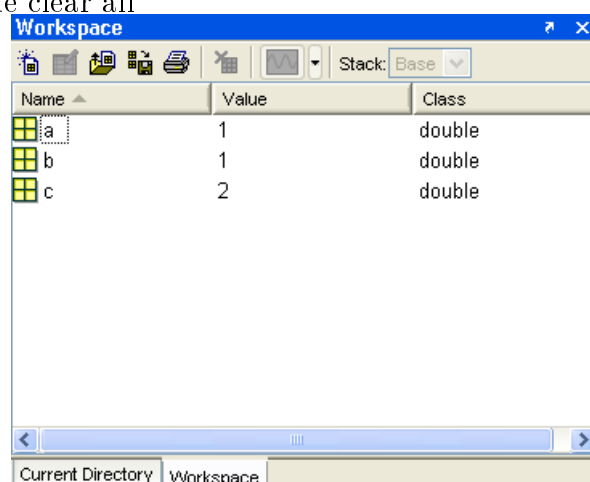
Il existe aussi une commande très pratique qui permet de supprimer toutes les variables en mémoire (voir encadré 4). Nous pouvons observer que l'utilisation de la commande `clear all` efface toutes les variables en mémoire. Si l'on exécute le fichier `test1.m` dont le contenu est

```

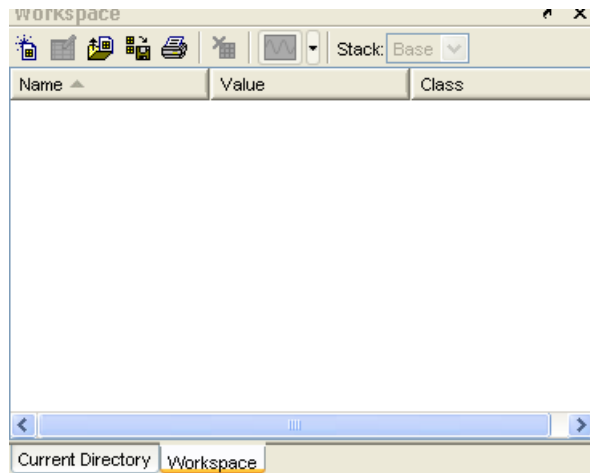
a = 1;
b = a;
c = a+b

```

La fenêtre workspace affiche
– avant l'exécution de `clear all`



– après l'exécution de `clear all`



Maintenant que notre fenêtre de travail est bien encombrée, on peut la nettoyer avec la commande `clc`. Et voilà, un nouvel environnement de travail. Attention, `clc` efface seulement l'écran et non les variables.

3.1 Vecteurs

Passons maintenant aux calculs vectoriels. On définit un vecteur de la manière suivante (ouvrez une nouvelle feuille Matlab vierge, copiez les instructions ci-dessous et sauvez-la sous le nom `test2.m`).

Attention, la numérotation des vecteurs commence toujours par 1.

```
% CECI EST UN COMMENTAIRE
clear all; % Efface toutes les variables en memoire
clc; % Nettoie l'ecran

% SYNTAXE POUR DEFINIR UN VECTEUR
v1 = [ 1, 2, 4] % Vecteur ligne
v2 = [ 1 2 4] % Encore un vecteur ligne avec des notations allégées

v3 = transpose(v2) % transpose le vecteur v qui devient
    % donc un vecteur colonne
% ATTENTION LA COMMANDE TRANSPOSE NE FONCTIONNE
% PAS SOUS OCTAVE
v4 = v1' % ' transpose le vecteur v1 (idem v2) avec une notation allégée

% EFFECTUER DES OPERATIONS AVEC DES VECTEURS
som1 = v1+v1 % Somme de vecteurs
som2 = 2*v1 % Multiplication par un scalaire

prod_scal1 = v1*v3 % Produit scalaire euclidien
prod_scal2 = v1*v2' % ou encore
```

```

prod_comp = v1.*v1 % Produit composante par composante
div_comp = v1./v1 % Division composante par composante

puiss_comp = v1.^3 % Mise a la puissance 3 composante par composante

mat = v1'*v1 % Produit vecteur colonne par vecteur ligne =matrice

%LORSQUE UNE ERREUR EST COMMISE MATLAB NE MANQUERA PAS DE VOUS
%LE SIGNALER
w = [ 1 3]
w+v1 % Operation impossible car la longueur des vecteurs est differente

% Bien entendu, nous pouvons effectuer les memes operations avec les
% vecteurs colonnes

```

Compilez et observez ...

On peut également définir un vecteur à incrément à pas constant par la commande [a :h :b] où h est le pas d'incrément, par défaut $h = 1$. Exemple,

```
>> v = [1:2:7]
```

```
v =
```

```
    1    3    5    7
```

```
>> v = [1:7]
```

```
v =
```

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

Pour extraire la $i^{\text{ème}}$ composante du vecteur v , on écrit $v(i)$. Simple, non ?

```
>> v(3)
```

```
ans =
```

```
    3
```

Du vecteur v , on peut aussi extraire une suite d'éléments

```
>> v(1:3)
```

```
ans =
```

```
    1    2    3
```

ou encore extraire les éléments suivant un incrément à pas constant (ci-dessous $h=2$)

```
>> v(1:2:7)
```

```
ans =
```

```
    1    3    5    7
```

Il peut être parfois utile de connaître la longueur du vecteur v . Pour cela, on utilise la commande `length(v)`, en l'occurrence ici la longueur de v vaut

```
>> length(v)
```

```
ans =
```

```
    7
```

P.S. N'hésitez pas à utiliser la commande `help` ou la touche F1.

3.2 Matrices

Quant aux matrices, la syntaxe est la suivante (ouvrir une nouvelle page et copiez les instructions suivantes) :

```
clear all; % Efface toutes les variables en memoire
```

```
clc; % Nettoie l'ecran
```

```
% SYNTAXE POUR DEFINIR UNE MATRICE: les lignes sont delimitees par des ;
```

```
A1 = [1 2 3 ; 4 5 6 ; 7 8 9 ]
```

```
%Ou encore
```

```
A1 = [1 2 3  
      4 5 6  
      7 8 9 ]
```

```
% ATTENTION a ne pas confondre avec ...
```

```
Attent = [1 2 3 ...
```

```
          4 5 6 ...
```

```
          7 8 9 ]
```

```
A2 = A1' % Matrice transposee
```

```
A2 = transpose(A1) % Matrice transposee
```

```
som = A1+A2
```

```
prod = A1*A2
```

```
prod_scal1 = 45*A1 % Multiplication d'une matrice par un scalaire
```

```
prod_scal2 = 45.*A1 % Ou encore
```



```
prod_comp = A1.*A2 % Produit element par element
prod_puis = A1^125 % A1*...*A1 125 fois
```

```
A3 = [ 1 2 3 ; 2 3 4 ] % Matrice 2 lignes, 3 colonnes
% A condition de respecter les dimensions des matrices, on peut effectuer des
% produits, des sommes avec des matrices non carrees
```

Compilez et observez...

Pour extraire l'élément i, j de la matrice A , on écrit $A(i, j)$. Simple, non ?

Comme pour le cas d'un vecteur, on peut définir une matrice par incrément à pas constant, extraire plusieurs éléments (en particulier des sous-matrices), ou connaître la taille d'une matrice, comme l'illustre l'exemple suivant :

```
>> A = [1:5 ; 3:2:11 ; 0:4]
```

```
A =
```

```
     1     2     3     4     5
     3     5     7     9    11
     0     1     2     3     4
```

```
>> A(1,3)
```

```
ans =
```

```
     3
```

```
>> A(1:3,1:2:3)
```

```
ans =
```

```
     1     3
     3     7
     0     2
```

```
>> size(A)
```

```
ans =
```

```
     3     5
```

Remarque 3.1 Bien qu'une matrice soit déjà définie, rien ne nous empêche d'étendre ses dimensions. Prenons par exemple la matrice A précédente de dimensions 3,5. Nous pouvons lui ajouter une ligne ou une colonne par la syntaxe suivante :

– pour une nouvelle colonne,

```
>> A(:,6)=[1:3]
```

A =

```
     1     2     3     4     5     1
     3     5     7     9    11     2
     0     1     2     3     4     3
```

La notation $A(:,6)$ désigne donc la 6^{ème} colonne de A.

– pour une nouvelle ligne,

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

A =

```
     1     2     3     4     5
     3     5     7     9    11
     0     1     2     3     4
     1     2     3     4     5
```

La notation $A(4,:)$ désigne donc la 4^{ème} ligne de A.

En bref, il est donc possible d'ajouter ou d'affecter toute une ligne ou colonne par le biais du sigle " :". Ainsi, $x = A(4,:)$ correspond au 4^{ème} vecteur ligne de A et $y = A(:,6)$ au 6^{ème} vecteur colonne de A à condition que la ligne ou la colonne à extraire existe. Dans le cas contraire un message d'erreur vous sera renvoyé.

Il existe des matrices pré-programmées telles que *eye*, *ones*, *zeros*, *magic* ...

```
>> eye(3)
```

ans =

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

```
>> ones(2,3)
```

ans =

```
     1     1     1
     1     1     1
```

```
>> zeros(4,4)
```

ans =

```
     0     0     0     0
     0     0     0     0
```

```
0 0 0 0
0 0 0 0
```

```
>> magic(3)
```

```
ans =
```

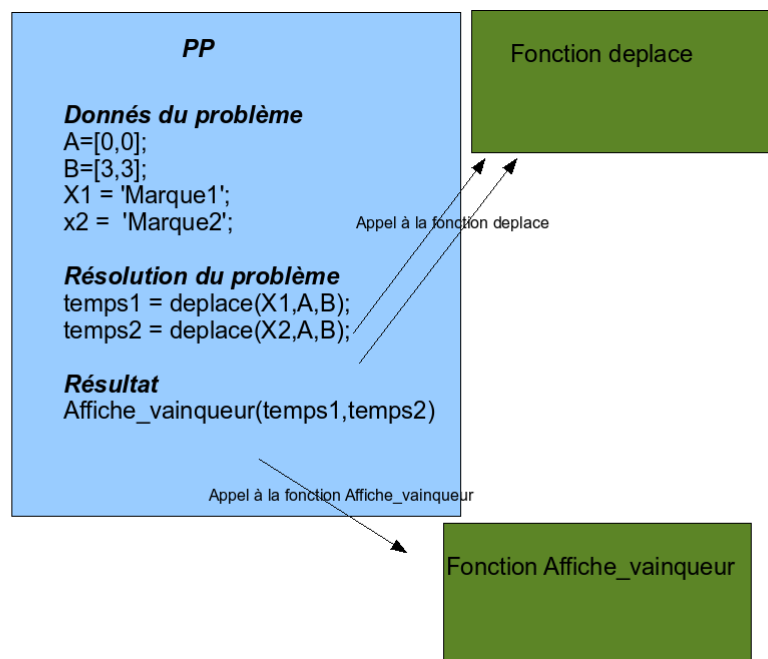
```
8 1 6
3 5 7
4 9 2
```

4 Concevoir un code numérique

4.1 Structure du programme

De manière générale, la structure d'un programme reflète la structure de notre raisonnement. Supposons que l'on souhaite calculer la durée du trajet effectué par une voiture de marque **X** pour aller d'un point **A** à un point **B**. On voudrait que ce calcul soit possible pour n'importe quelle marque de voiture et pour tous points **A** et **B**. Pour ce faire, le programme principal (**PP**) va faire appel à une fonction « **deplace** » prenant comme paramètres d'entrée la marque **X** et les points **A** et **B**, et fournissant en sortie la durée du trajet. Notre **PP** sera donc constitué des instructions suivantes : l'affectation des valeurs de **A**, **B** et **X** (les données du problème) et l'appel à la fonction **deplace** appliquée à ces données.

Le code sera donc constitué de deux « feuilles », la première contenant les instructions du programme principal et l'autre permettant de définir la fonction **deplace**.



Remarque 4.1 Dans les versions ultérieures à Matlab 6, on peut écrire un code en n'utilisant qu'une seule feuille. Cette feuille aura la structure suivante :

PP (Vu comme une fonction sans paramètres)
Donnés du problème A=[0,0]; B=[3,3]; X1 = 'Marque1'; x2 = 'Marque2';
Résolution du problème temps1 = deplace(X1,A,B); temps2 = deplace(X2,A,B);
Résultat Affiche_vainqueur(temps1,temps2)
Les fonctions (en fin de feuille) Fonction deplace Fonction Affiche_vainqueur

Dans cette figure, on montre un exemple de construction possible de programme. Nous verrons dans le paragraphe suivant comment définir une fonction.

En résumé, la structure d'un programme Matlab ou Octave est généralement la suivante :

1. données du problème,
2. résolution du problème,
3. résultats (graphiques ou numériques).

Il est fortement conseillé de commencer votre programme principal par les commandes `clear all ; clc ;` afin d'éviter tout conflit de variables entre deux exécutions.

4.2 Déclaration de fonctions du type *M-files functions* et *Inline functions*

C'est dans la partie « résolution du problème » que notre PP fait appel à d'autres codes qu'on appelle fonctions. Attention, Matlab et Octave ne possèdent pas d'environnement pour les procédures.

On définit une fonction de la manière suivante

```
function [y1,...,ym]=nom_de_la_fonction(x1,...,xn)
% CORPS DE LA FONCTION
return % Ou encore end ou endfunction
```

Remarque 4.2 Il est évident que la commande `clear all` est à éviter dans une fonction (juste après le mot clef *function*) sauf dans le cas de structure de la figure 4.1.

Les paramètres x_1, \dots, x_n sont les paramètres d'entrée (connus par le PP) tandis que y_1, \dots, y_m sont les paramètres de sorties calculés en fonction de x_1, \dots, x_n . On fait appel à la fonction `nom_de_la_fonction` par :

```
[z1,...,zm]=nom_de_la_fonction(a1,...,an)
```

On affecte le résultat de la fonction `nom_de_la_fonction` aux variables z_1, \dots, z_m grâce aux données a_1, \dots, a_n .

Nous allons procéder à la construction de notre premier programme. Créez deux nouvelles feuilles Matlab et sauvegardez-les respectivement sous les noms `pp.m` et `fun1.m` comme suit

```
[contenu de pp.m]
% ON NETTOIE LA MEMOIRE ET L'ECRAN
clear all; clc;
% DONNEES DU PROBLEMES
x0 = 3;

%CORPS DU PROGRAMME PRINCIPAL
y = fun1(x0);

%AFFICHAGE DU RESULTAT
disp(' Le resultat de fun1 est ')
y

[contenu de fun1.m]
function y = fun1(x)
y = x^3;
return
```

On exécute et Matlab renvoie

```
Le resultat de fun1 est
```

```
y =
```

```
27
```

Bravo! Nous venons de compiler notre premier programme avec succès.

Remarquez que ce PP met en oeuvre le calcul de la fonction $x \mapsto x^3$ pour $x = x_0 = 3$ où x_0 est un scalaire.

Si maintenant on souhaite non pas utiliser `fun1` avec un scalaire x mais effectuer simultanément plusieurs évaluations de `fun1`, comment procéder? Si l'on se contente de remplacer x_0 par une liste de points, c'est-à-dire à l'aide d'un vecteur $x_0 = [1, 2, 3, 4, 5]$ et d'exécuter le PP, que se passe-t-il? On voit apparaître le message d'erreur :

```
??? Error using ==> mpower Matrix must be square.
```

```
Error in ==> fun1 at 2 y = x^3;
```

```
Error in ==> pp at 9 y = fun1(x);
```

Avant de résoudre ce problème, observez le message d'erreur. On voit qu'une erreur s'est glissée dans le PP à la ligne 9 dans la fonction `fun1` à la ligne 2, à cause de la *puissance* : *la matrice doit être carrée*. Cela signifie que l'instruction `^3` est valable uniquement si la matrice est carrée. En l'occurrence le chiffre 3 est une matrice carrée de taille 1×1 , tandis que le vecteur ligne `x0` n'est pas carré ! On doit donc faire faire une évaluation de la liste point par point ; rappelez-vous que la commande `.^` permet cela. Notons au passage que les messages d'erreurs sont à lire du bas vers le haut. Procédons alors à la rectification du code de la fonction `fun1` en remplaçant `^` par `.^` et exécutons à nouveau :

```
[si x0=[1:5]]
```

```
Le resultat de fun1 est
```

```
y =
```

```
1      8     27     64    125
```

```
[si x0=3 a nouveau ?]
```

```
Le resultat de fun1 est
```

```
y =
```

```
27
```

Et ça marche ! Fort heureusement. Remarquez aussi que le type du paramètre de sortie de `fun1` (`y`) s'adapte au type du paramètre d'entrée. L'entrée étant un vecteur, la sortie est aussi un vecteur. L'adaptativité du type est une caractéristique très agréable de Matlab : elle nous évite de déclarer les types de chaque variable et constante utilisée comme en *C*, *C++*, *Fortran*, ...

Il existe un autre type de fonctions appelées *Inline functions* souvent utilisé lorsque la fonction à écrire contient une seule instruction. C'est le cas par exemple de notre fonction `fun1`. Généralement, la fonction inline est déclaré dans le PP par la syntaxe

```
nom_de_la_fonction= inline('1 instruction(x1,...,xn)', 'x1', 'x2', ..., 'xn');
```

La dernière partie signifie que la fonction dépend de `x1, ..., xn`.

L'appel à `nom_de_la_fonction` se fait de la manière suivante : étant donnés `x1, ..., xn`,

```
y = nom_de_la_fonction(x1, ..., xn).
```

Dans notre exemple, on code tout simplement

```

% COMME D'HABITUDE ON NETTOIE LA MEMOIRE ET L'ECRAN
clear all; clc;
% DONNEES DU PROBLEMES
x = [1:5];

%CORPS DU PROGRAMME PRINCIPAL
fun1=inline('x.^3','x');
y = fun1(x);

%AFFICHAGE DU RESULTAT
disp(' Le resultat de fun1 est ')
y

```

Nous venons de voir une structure de programme très simple, faisant appel à une fonction possédant une instruction. Evidemment, il est possible d'appeler plusieurs fonctions dans le PP dont les fonctions possèdent plusieurs instructions.

Nous allons corser le tout avec des conditions de test **boolean**, des structures en *if* et des boucles *while* et *for*.

Dans toute la suite de ce tutoriel, on laisse soin à l'utilisateur de se familiariser avec l'aide pour les quelques fonctions présentées. En effet, l'aide est un outil primordial en Matlab ou Octave, dont l'utilisation est simple et efficace.

4.3 Tests et boucles

4.3.1 Tests

Pour vérifier si un objet (matrice, vecteur, fonction, ...) satisfait une condition **cond**, la structure des fonctions implémentées dans Matlab est **iscond**. Pour avoir un aperçu de toutes les fonctions de ce type tapez **help is** dans la ligne de commande. Par exemple,

```

>> A=[1:4]
A =

     1     2     3     4

>> isempty(A)

ans =

     0

```

On demande à Matlab, si l'élément A défini précédemment est vide. Matlab nous répond alors non par une réponse 0 pour *false* et 1 pour *true*.

On peut également vérifier des conditions par le biais d'une structure *if*.

4.3.2 If, for et while

Les structures suivantes sont les articulations d'un programme. Il est donc essentiel de les maîtriser. Utilisez help ou F1 pour obtenir de la documentation précise sur if, while, for.

[Exemple pour if]

```
a = 1; % Quelque part dans notre code a prend la valeur 1 et plus loin...

% dans notre programme, quelque part dans une fonction
%... on teste une condition d'egalite sur a
if a==1
    disp('Bonjour') % Affiche Bonjour
    else choix = 101;
end

%Puis encore plus loin dans notre execution,...
% On teste si choix est strictement plus grand que 100 ET si a vaut 1
if choix>100 & a==1
    for k=1:134
        x(k) = 134-k;
    end
end

% Et encore plus loin depuis le debut d'execution,
% choix <100 et a ne doit pas etre egal a 1
if a~=1 % si a est different de 1
    m=0;
    while (m<100)
        m=m+100;
    end
else disp('Aurevoir') % Si a est egal a 1, nous devons malheureusement quitte
    return % ces parcelles de codes...
```

Voici une illustration simpliste d'une utilisation de if, for et while, certes dénuée de sens, mais qui met en évidence que le temps c'est de l'argent. PENSEZ A OPTIMISER LA MEMOIRE! Nous allons donc aborder ce genre de problème de stockage afin d'éviter les calculs inutiles. Pour ce faire, je choisis l'algorithme de Newton qui peut être écrit de deux façon, dont une économe.

L'algorithme de Newton consiste à calculer les différences divisées selon un procédé pyramidal. On note $f[x_0, \dots, x_k]$ les différences divisées associées aux points x_0, \dots, x_k avec $(f[x_i])_{i=0, \dots, n}$ connus. Il s'agit de calculer ces différentes valeurs pour $k = 1, \dots, n$.

Algorithme "naif" for k = 1: n
for i = 0: n - k


```

        d(i,k) = (d(i+1,k+1) - d(i,k-1))/(xi+k- xi)
    end
end

```

```

Algorithme "économe" for k=1:N+1;
    d(k)=fonction(x(k),alpha);
end
for k=1:N;
    for i=1:N+1-k;
        d(i)= (d(i+1) - d(i))/(x(i+k)-x(i));
    end
end

```

Constatez la différence entre ces deux algorithmes : l'un utilise un stockage matriciel et l'autre un stockage vectoriel. De toute évidence, le deuxième algorithme est plus « rapide » car il écrase les valeurs stockées inutilement.

Considérons un nouvel exemple : l'algorithme de Newton-Raphson consiste à chercher le zéro d'une fonction f en générant une suite $(x_n)_{n \geq 0}$ telle que $(f(x_n))_{n \geq 0}$ converge vers 0.

```

Algorithme "naif" x(1)=x0;
    n=1;
    while abs(f(x(n)))>= e & (n < itermax)
        x(n+1) = x(n) - f(x(n))./fp(x(n));
        n = n+1;
    end

```

```

Algorithme "économe" x=x0;
    n=1;
    while abs(f(x))>= e & (n < itermax)
        x = x - f(x)/fp(x);
        n = n+1;
    end

```

Comme l'exemple précédent, le deuxième algorithme écrase les valeurs de la suite $(x_n)_{n \geq 0}$ générés par le premier l'algorithme.

Enfin, pour terminer ce paragraphe, je vais vous montrer par un exemple simple que le stockage inutile engendre l'instabilité de votre système. On suppose que l'on veut stocker la matrice identité de taille 10000.

[STOCKAGE 1]

```

tic
A = speye(10000);
toc

```

```

tic
A = eye(10000);
toc

```

La commande `tic instruction toc` permet de chronométrer le temps d'exécution de l'instruction `instruction`. La commande `sparse(A)` stocke toutes les valeurs non nulles de la matrice `A`.

Remarque 4.3 Pour stopper une exécution en cas de problème *Ctrl+c*.

Compilez et observez.

5 Représentation de nos résultats

5.1 Représentation graphique

Nous allons à présent nous consacrer à la représentation graphique de nos résultats à travers des exemples.

Supposons que nous souhaitions tracer la fonction $x \in [0, 1] \mapsto x^3$. Nous pouvons procéder ainsi :

```
x=[0:1/100:1];  
y=x.^3;  
plot(x,y)
```

ou en utilisant la fonction `fun1` que nous avons créée,

```
x=[0:1/100:1];  
y=fun1(x);  
plot(x,y)
```

Si l'on souhaite tracer une deuxième fonction, par exemple

```
y1=-x.^3;
```

pour la représentation de ces deux graphes, plusieurs options sont possibles :

1. tracer `y` et `y1` sur le même graphe,
2. tracer `y` puis ajouter `y1` sur le même graphe,
3. tracer `y` et `y1` sur deux graphes différents,
4. afficher simultanément `y` et `y1` sur deux graphes différents.

Avant de commencer le cas par cas, il est bon de savoir que l'on peut rajouter un titre via la commande `title`, une légende via `legend`, nommer les abscisses et les ordonnées via `xlabel` et `ylabel`, placer un texte dans la fenêtre graphique via `text` et bien sûr la commande `help` pour en savoir plus.

Revenons au cas numéro 1 : on utilisera

```
plot(x,y,x,y1)
```

dans notre exemple.

Vous avez compris que la généralisation de l'utilisation de cette commande à n fonctions est la suivante `plot(x1,y1,...,xn,yn)`.

En ce qui concerne le cas 2, supposez que, quelque part au début de votre programme principal, vous tracez la fonction `y` et que plus loin, vous souhaitez ajouter le tracé de `y1`, alors la commande

`hold on`

est faite pour vous. Si vous souhaitez désactiver cette commande dans votre programme, il vous suffit alors d'utiliser `hold off`.

Pour le cas 3, on utilise la commande `figure(1)` suivi de la commande pour tracer, puis `figure(2)` ...

Pour le cas 4, vous souhaitez obtenir deux résultats graphiques sur une même fenêtre alors il vous suffit de diviser cette fenêtre par la commande

`subplot`

. L'utilisation de cette dernière est la suivante

`subplot(l,c,p)`

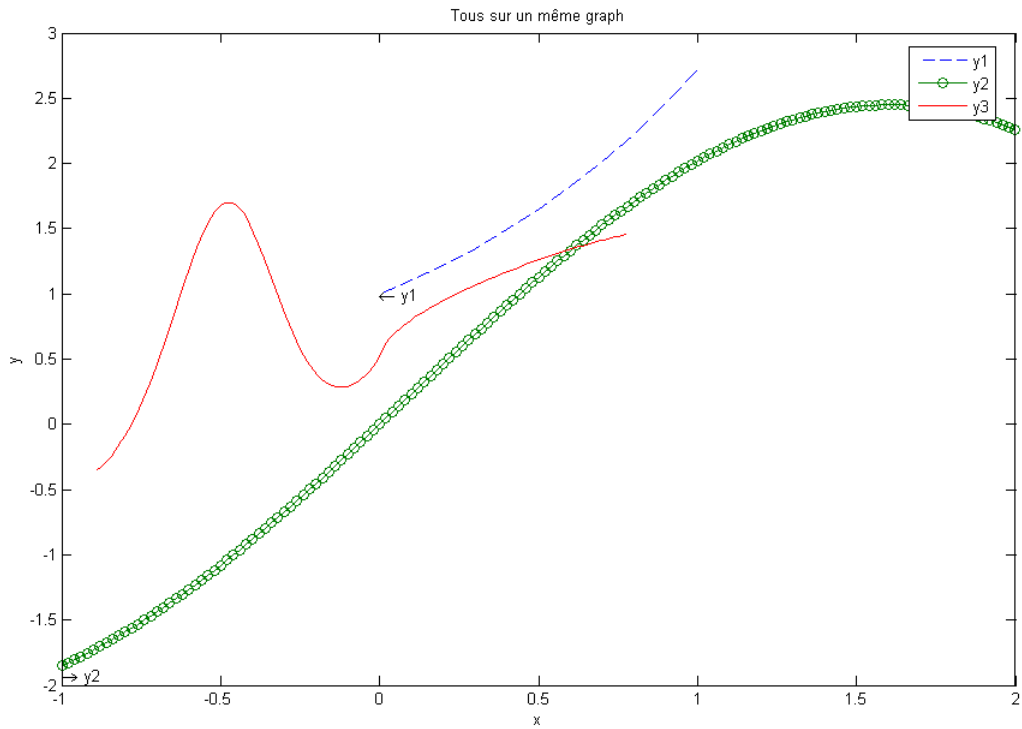
où l'indice l représente le nombre de divisions verticales, c le nombre de divisions horizontales, et p la position de votre graphe dans la fenêtre sachant que l'énumération se fait de gauche à droite et commence par 1.

Illustration :

```
[cas 1]
x1 = [0:1/10:1];
x2 = [-1:1/50:2];
x3 = [-0.89:1/89:0.78];

y1 = exp(x1);
y2 = log(x2+10).*sin(x2);
y3 = x3+ cos(x3.^x3);

plot(x1,y1,'--',x2,y2,'o-',x3,y3)
title('Toutes sur un meme
graphe') xlabel('x') ylabel('y') text(0,1,'\leftarrow y1')
text(x2(1),y2(1),'\rightarrow y2') legend('y1','y2','y3')
```



[cas 2]

```

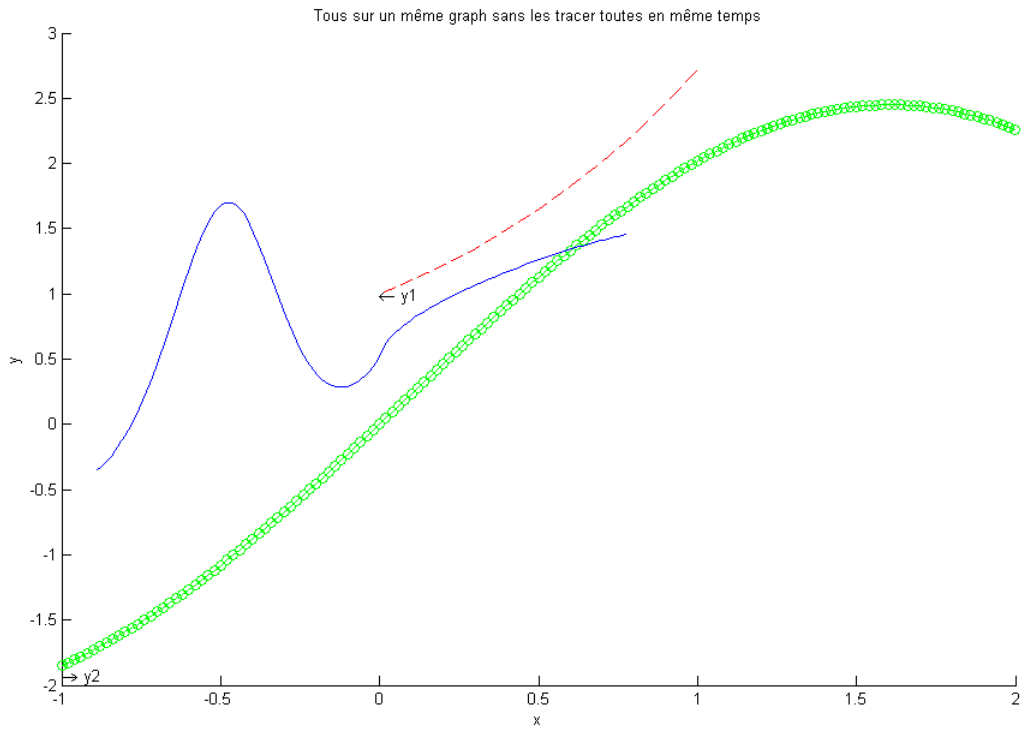
hold on plot(x1,y1,'--','color','red')
title('Toutes sur un meme
graph sans les tracer toutes en meme temps')
xlabel('x')
ylabel('y')
text(0,1,'\leftarrow y1')

plot(x2,y2,'o-','color','green')

text(x2(1),y2(1),'\rightarrow y2','VerticalAlignment','top')

plot(x3,y3)

```



```
[cas 3]
figure(1)
plot(x1,y1,'--','color','red')
```

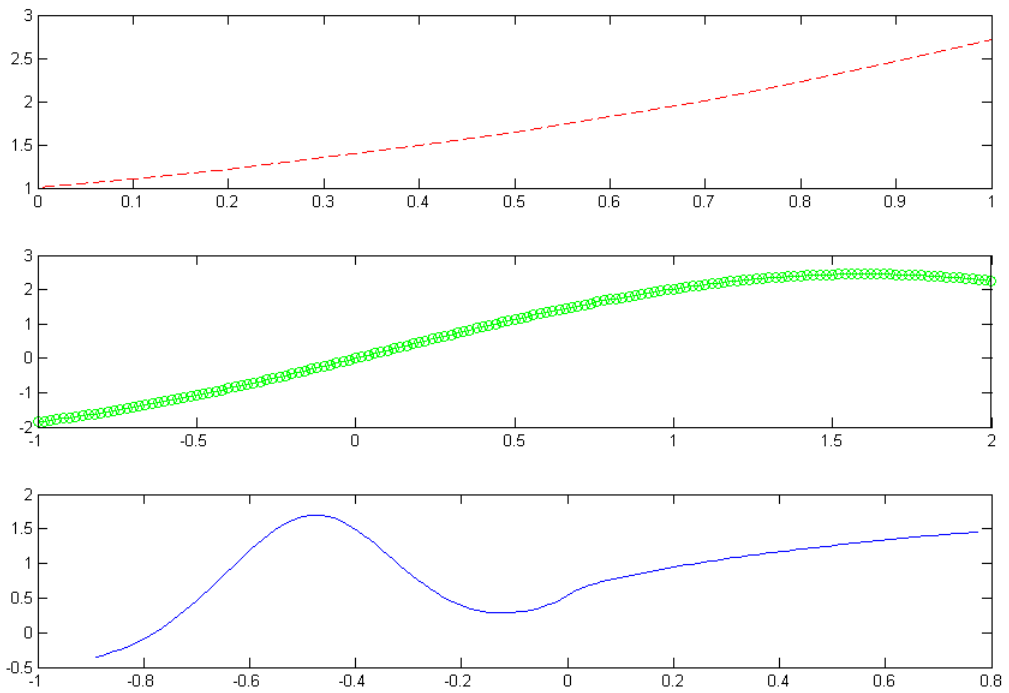
```
figure(2)
plot(x2,y2,'o-','color','green')
```

Testez *ldots*

```
[cas 4]
subplot(311)
plot(x1,y1,'--','color','red')
```

```
subplot(312)
plot(x2,y2,'o-','color','green')
```

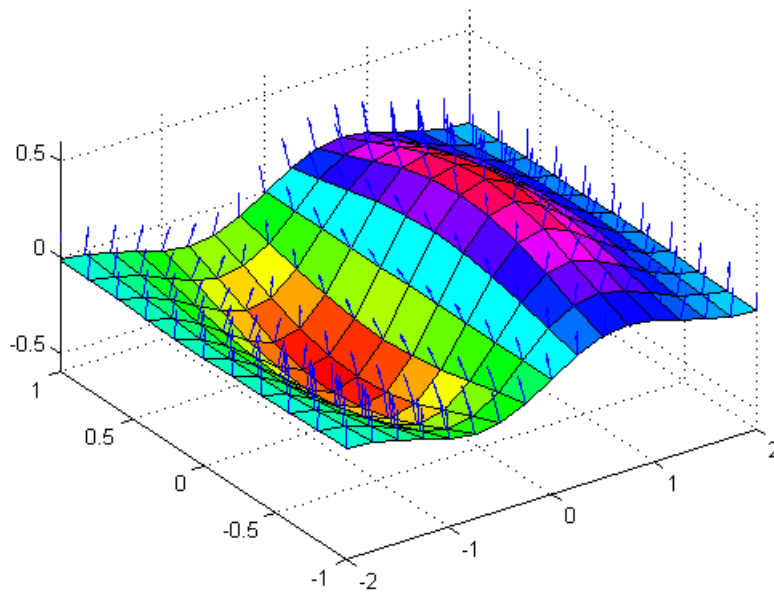
```
subplot(313)
plot(x3,y3)
```



Il existe aussi d'autres types de commande pour la représentation graphique, par exemple

`plot3`

pour le 3D ou encore `quiver`, `quiver3` pour tracer les champs de vecteurs. Par exemple,



Une fois de plus, help est votre meilleure amie ...

See Also

[axis](#), [bar](#), [grid](#), [hold](#), [legend](#), [line](#), [LineStyle](#), [loglog](#), [plot3](#), [plotyy](#), [semilogx](#), [semilogy](#), [subplot](#), [title](#), [xlabel](#), [xlim](#), [ylabel](#), [ylim](#), [zlabel](#), [zlim](#), [stem](#)

See the text [String](#) property for a list of symbols and how to display them.

See the [Plot Editor](#) for information on [plot](#) annotation tools in the figure window toolbar.

See [Basic Plots and Graphs](#) for related functions.



Je vous laisse soin de deviner à quoi cette image peut bien correspondre.

5.2 Lecture/écriture dans un fichier

5.2.1 écriture

Prenons l'exemple suivant. On suppose que l'on veut écrire (stocker) les abscisses et ordonnées d'une fonction (par exemple $x \in [0, 1] \mapsto x^3$) dans un fichier de données. Pour cela, créez un nouveau document que vous appelez `test_ecriture.txt` (ou bien `.dat`; les deux formats étant valables).

```
delete('test_ecriture.txt'); % Efface le contenu du fichier
fid=fopen('test_ecriture.txt','a') % Ouvre le fichier

% Les donnees a ecrire dans le fichier
x = [0:1/10:1]; y = x.^3;

fprintf(fid,'%g ',x); % Ecriture du vecteur x
fprintf(fid,'\n'); % Retour chariot
fprintf(fid,'%g ',y); % Ecriture du vecteur y
fprintf(fid,'\n %%Ceci est un commentaire, ... \n' );
fprintf(fid,' %%il ne sera pas lu lors de la lecture ');

fclose(fid); % Fermeture du fichier

open('test_ecriture.txt'); % Ouverture du fichier dans un editeur
```

Je vous laisse le soin de découvrir la signification des instructions

```
%g,\n,'a'
```

```
...
```

5.2.2 Lecture

Pour lire les données d'un fichier de données, on utilise la commande `open` (ou on clique tout simplement sur un fichier). Pour charger en mémoire les données du fichier on utilise la commande `load` dans le terminal. Prenons par exemple, le fichier `test_ecriture.txt` en vue d'en extraire les données.

```
X = load('test_ecriture.txt')
```

Attention, pour Octave, la syntaxe est la suivante :

```
X = load test_ecriture.txt
```

Observez le type de `X`. Matlab voit les données comme une matrice à 2 lignes et 11 colonnes vu la répartition du texte dans le fichier de données. On peut donc extraire les données et tracer la fonction énoncée dans la sous-section précédente.

```
plot(X(1,:),X(2,:))
```

De toute évidence, notre exemple a été construit de tel sorte que le résultat soit interprété comme une matrice. A vous de voir comment organiser le fichier `.txt` (ou `.dat`) selon vos exigences.

6 Debugger un programme

Dans ce paragraphe nous allons voir comment debugger un programme. Il peut arriver que rien ne marche, que Matlab vous donne des erreurs incompréhensibles et que vous commenciez à perdre patience ... Ne vous inquiétez pas et ne stressez plus car la touche F12 vous guidera vers vos erreurs. Considérez l'exemple simple suivant :


```

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: Base
debug.m x fibo.m x
1 % pp
2 - clear all;
3 - clc;
4
5
6 % Données du problème
7 - x0 = 1;
8 - x1 = 1;
9 - N = 100;
10
11 %Résolution du problème
12 - for k=1:N
13 -     y1 = fibo(x0,x1,k);
14 -     y2 = fibo(x0,y1,k);
15 -     y(k) = sin(fibo(y1,y2,k));
16 - end
17
18
19
20 % Résultats
21 - plot(1:N,y)
22

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: Base
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels  $x(n+2)=x(n+1)+x(n)$ 
3 %--- input -> x0,x1 : Initialisation de la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de la suite à l'itération N
6 %
7
8 - a1 = x0;
9 - b = x1;
10
11 - for i=0:N
12 -     xn = a + b;
13 -     a = b;
14 -     b = xn;
15 - end
16
17
18 - return
19

```

Exécutez le programme principal et observez le message d'erreur suivant :

```

Command Window

??? Undefined function or variable "a".

Error in ==> fibo at 12
    xn = a + b;

Error in ==> debug at 13
    y1 = fibo(x0,x1,k);

>> |

```

D'après le message d'erreur ci-dessus, il est clair que dans la fonction `fib`o il suffit de remplacer `a1` par `a` pour résoudre le problème mais supposons que nous ne l'ayons pas vu (ce qui pourrait vous arriver ... faute de frappe ...).

Commençons à debugger le programme. Si l'on pense que l'erreur provient de la fonction `fib`o, il suffit de placer le curseur à la ligne où la fonction `fib`o est appelée d'appuyer sur la touche F12. On voit alors apparaître

The screenshot shows the MATLAB editor interface with a menu bar (File, Edit, Text, Cell, Tools, Debug, Desktop, Window, Help) and a toolbar. Two files are open: `debug.m` and `fib.m`. The main window displays the following code:

```

1  % pp
2  - clear all;
3  - clc;
4
5
6  % Données du problème
7  - x0 = 1;
8  - x1 = 1;
9  - N = 100;
10
11 %Résolution du problème
12 - for k=1:N
13  |   y1 = fibo(x0,x1,k); breakpoint
14  -   y2 = fibo(x0,y1,k);
15  -   y(k) = sin(fibo(y1,y2,k));
16  - end
17
18
19
20 % Résultats
21 - plot(1:N,y)
22

```

Si l'on ne sait pas du tout où se trouve l'erreur, on place le « breakpoint » en tout début de programme principal et on appuie sur la touche F12, ce qui donne :

```
Command Window
??? Undefined function or variable "a".

Error in ==> fibonacci at 12
    xn = a + b;

Error in ==> debug at 13
    y1 = fibonacci(x0,x1,k);

>> debug|
```

Ensuite, on procède comme dans l'illustration qui suit :

```

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: debug
debug.m x fibo.m x
1 % pp
2 - clear all;
3 - clc;
4
5
6 % Données du problème
7 - x0 = 1;
8 - x1 = 1;
9 - N = 100;
10
11 %Résolution du problème
12 - for k=1:N
13 -> | y1 = fibo(x0,x1,k);
14 - | y2 = fibo(x0,y1,k);
15 - | y(k) = sin(fibo(y1,y2,k));
16 - end
17
18
19
20 % Résultats
21 - plot(1:N,y)
22

```



```

[Icons] Stack: fibo
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels x(n+1)+x(n)
3 %--- input -> x0,x1 : Initialisat la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de l à l'itération N
6 %
7
8 -> | a1 = x0;
9 - | b = x1;
10
11 - for i=0:N
12 - | xn = a + b;
13 - | a = b;
14 - | b = xn;
15 - end
16
17
18 - return
19

```



```
File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: fibo
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels x(n+1)+x(n)
3 %--- input -> x0,x1 : Initialisat la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de 1 à l'itération N
6 %
7
8 - a1 = x0;
9 - b = x1;
10
11 - for i=0:N
12 -     xn = a + b;
13 -     a = b;
14 -     b = xn;
15 - end
16
17
18 - return
19
```

```
File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: fibo
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels x(n+1)+x(n)
3 %--- input -> x0,x1 : Initialisat la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de 1 à l'itération N
6 %
7
8 - a1 = x0;
9 - b = x1;
10
11 - for i=0:N
12 -     xn = a + b;
13 -     a = b;
14 -     b = xn;
15 - end
16
17
18 - return
19
```

```

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: fibo
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels x(n+1)+x(n)
3 %--- input -> x0,x1 : Initialisat la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de l à l'itération N
6 %
7
8 - a1 = x0;
9 - b = x1;
10
11 - for i=0:N
12 - | xn = a + b;
13 - | a = b;
14 - | b = xn;
15 - end
16
17
18 - return
19

```



```

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: Base
debug.m x fibo.m x
1 function xn = fibo(x0,x1,N)
2 %--- description : Suite de réels x(n+2)=x(n+1)+x(n)
3 %--- input -> x0,x1 : Initialisation de la suite
4 %--- input/output -> /
5 %--- output -> / D : Résultat de la suite à l'itération N
6 %
7
8 - a1 = x0;
9 - b = x1;
10
11 - for i=0:N
12 - | xn = a + b;
13 - | a = b;
14 - | b = xn;
15 - end
16
17
18 - return
19

```

Command Window

```
??? Undefined function or variable "a".
```

```
Error in ==> fibonacci at 12
```

```
    xn = a + b;
```

```
Error in ==> debug at 13
```

```
    y1 = fibonacci(x0,x1,k);
```

```
>> |
```

Attention : cette manipulation peut être longue!!!
Terminons ce tutoriel par quelques exemples.

7 Exemples de programmes

7.1 Système prédateur proie

En mathématiques, les équations de Lotka-Volterra, que l'on désigne aussi sous le terme de « modèle prédateur-proie », sont un couple d'équations différentielles non-linéaires du premier ordre, et sont couramment utilisées pour décrire la dynamique de systèmes biologiques dans lesquels un prédateur et sa proie interagissent. Elles ont été proposées indépendamment par Alfred J. Lotka en 1925 et Vito Volterra en 1926.

```
[Programme principal]
```

```
clear all;
```

```
clc;
```

```
clf;
```

```
% SYSTEME PREDATEUR PROIE
```

```
r = 0.2;
```

```
K = 300000;
```

```
p = 1;
```

```
k = 0.25;
```

```
T = 40;
```

```
x0 = 200000/K;
```

```
Xm=0:0.1:1;
```

```
Tps = 0:1:T;
```

```
options = odeset('RelTol',1e-4,'AbsTol',1e-4);
```

```
%ABSENCE D'EXPLOITATION
```

```

[T1,Y1] = ode45(@LV0,Tps',x0,options);

subplot(221)
hold on
field(@LV0,Tps,Xm);
[t,y]=meshgrid(T1,Y1);
plot(T1,Y1,'b')
hold off
title('En l''absence d''exploitation')

%EXPLOITATION u =1000
[T2,Y2] = ode45(@LV1000,Tps',x0,options);
subplot(222)
hold on
field(@LV0,Tps,Xm);
[t,y]=meshgrid(T2,Y2);
plot(T2,Y2,'b')
hold off
title('Exploitation')

subplot(223)
plot(T1,Y1,T2,Y2)
legend('En l''absence d''exploitation','Exploitation')

%Pontryagin
[Tstar,Ystar]=ode45(@LVstar,Tps,x0,options);
[T3,Y3]=ode45(@LV20000,Tps,x0,options);
subplot(224)
plot(T1,Y1,Tstar,Ystar,T3,Y3)
axis([0,40,-0.1,1.1])
legend('u=0','u=ustar','u=Umax')
title('Taille des ressources cas optimaux')

[Fichier LV0.m]
function Y = LV0(t,X)
%--- description : SystÃme proie-prÃdateur

```



```

%--- input -> t,X : Temps, Variable ou vecteur
%--- input/output -> /
%--- output -> / D : Valeur(s)
u = 0;
r = 0.2;
K = 300000;

Y = r.*X.*(1-X)-u./K;

```

```
return
```

```

[Fichier LV1000.m]
function Y = LV1000(t,X)
%--- description : SystÃme proie-prÃdateur
%--- input -> t,X : Temps, Variable ou vecteur
%--- input/output -> /
%--- output -> / D : Valeur(s)
u = 1000;
r = 0.2;
K = 300000;

Y = r.*X.*(1-X)-u./K;

```

```
return
```

```

[Fichier LV20000.m]
function Y = LV20000(t,X)
%--- description :
%--- input -> t,X :
%--- input/output -> /
%--- output -> / D :
r = 0.2;
K = 300000;
delta = 0.01;
u = 20000;

```

```

Y = r*X*(1-X)-u/K;
return

```

```

[Fichier LVstar.m]
function Y = LVstar(t,X)
%--- description :
%--- input -> t,X :
%--- input/output -> /

```

```

%--- output -> / D :
r = 0.2;
K = 300000;
delta = 0.01;

u = (-delta+r)*K/(2*r);

```

```

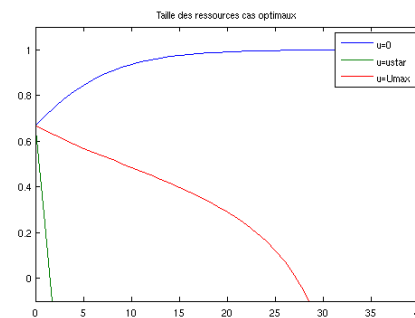
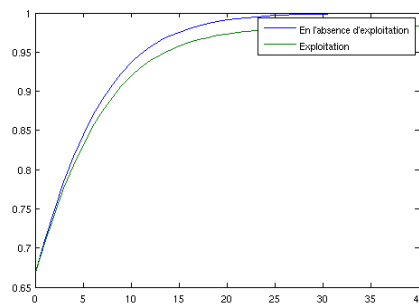
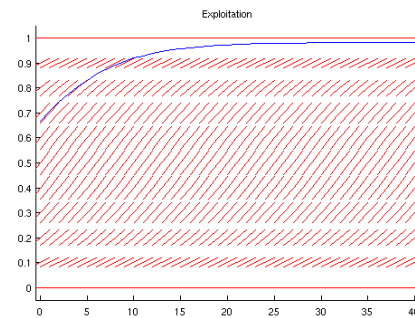
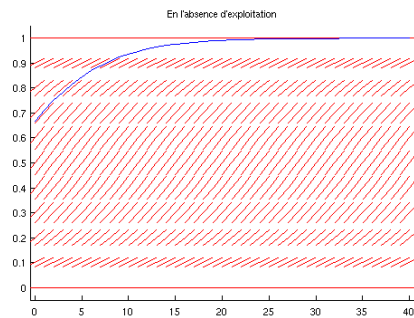
Y = r*X*(1-X)-u/K;

```

```

return

```



7.2 Klein bottle

En mathématiques, la bouteille de Klein est une surface fermée, sans bord et non orientable, c'est-à-dire une surface pour laquelle il n'est pas possible de définir un « intérieur » et un « extérieur ». La bouteille de Klein a été décrite pour la première fois en 1882 par le mathématicien allemand Felix Klein. Elle est étroitement liée au ruban de Möbius et à des plongements du plan projectif réel tels que la surface de Boy.

C'est un des exemples les plus simples de variété abstraite, car c'est une surface qui ne peut être représentée convenablement dans l'espace à trois dimensions.

```

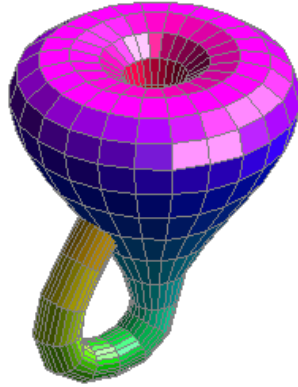
n = 12;
a = .2; % the diameter of the small tube
c = .6; % the diameter of the bulb
t1 = pi/4 : pi/n : 5*pi/4; % parameter along the tube
t2 = 5*pi/4 : pi/n : 9*pi/4; % angle around the tube
u = pi/2 : pi/n : 5*pi/2;
[X,Z1] = meshgrid(t1,u);
[Y,Z2] = meshgrid(t2,u);

% The handle
len = sqrt(sin(X).^2 + cos(2*X).^2);
x1 = c*ones(size(X)).*(cos(X).*sin(X) ...
    - 0.5*ones(size(X))+a*sin(Z1).*sin(X)./len);
y1 = a*c*cos(Z1).*ones(size(X));
z1 = ones(size(X)).*cos(X) + a*c*sin(Z1).*cos(2*X)./len;
handleHndl=surf(x1,y1,z1,X);
set(handleHndl,'EdgeColor',[.5 .5 .5]);
hold on;

% The bulb
r = sin(Y) .* cos(Y) - (a + 1/2) * ones(size(Y));
x2 = c * sin(Z2) .* r;
y2 = - c * cos(Z2) .* r;
z2 = ones(size(Y)) .* cos(Y);
bulbHndl=surf(x2,y2,z2,Y);
set(bulbHndl,'EdgeColor',[.5 .5 .5])

colormap(hsv);
axis vis3d
view(-37,30);
axis off
light('Position',[2 -4 5])
light
hold off

```



7.3 Résolution d'un problème de valeurs propres par une méthode d'éléments finis.

Considérons un domaine carré Ω sur lequel on va chercher les valeurs propres de l'opérateur laplacien avec conditions au bord de Dirichlet, et les fonctions propres associées. Mathématiquement, le problème se formule ainsi : trouver $(\lambda, u) \in \mathbb{R} \times H^1(\Omega) \setminus \{0\}$ tel que

$$\begin{cases} \Delta u = \lambda u & \text{dans } \Omega, \\ u = 0 & \text{sur } \partial\Omega. \end{cases} \quad (1)$$

Le principe de la méthode est de chercher des solutions approchées du problème (1), en remplaçant l'espace de fonctions $V := H^1(\Omega)$, de dimension infinie, par un sous-espace V_h de dimension finie. Numériquement, le problème traité est donc le suivant : trouver $(\lambda_h, u_h) \in \mathbb{R} \times V_h$ tel que

$$\begin{cases} \Delta u_h = \lambda_h u_h & \text{dans } \Omega, \\ u_h = 0 & \text{sur } \partial\Omega. \end{cases} \quad (2)$$

Pour construire les fonctions de base de l'espace V_h , on utilise un maillage du domaine Ω . Ces fonctions de base vont permettre d'écrire le problème (2) sous la forme d'un problème matriciel du type : trouver un réel λ_h et un vecteur U_h tels que $AU_h = \lambda_h BU_h$. Les matrices A et B sont générées automatiquement par Matlab, à ceci près qu'il faut ajouter à A des termes dits de « pénalisation » pour imposer la condition de Dirichlet $u_h = 0$ sur le bord.

Les étapes de la résolution du problème (2) par une méthode d'éléments finis en Matlab sont donc les suivantes :

1. **Construction du maillage** : la génération du maillage se fait grâce à la fonction `initmesh`. Cette fonction renvoie les matrices `p`, `e`, `t` qui représentent le maillage.

En particulier, la matrice \mathbf{p} est une matrice à deux colonnes : chaque ligne de \mathbf{p} contient les coordonnées des points du maillage. La fonction `initmesh` prend en entrée la matrice `d1` appelée *matrice de géométrie décomposée*, que l'on obtient en appliquant la fonction `decsg` à la matrice `gd`, appelée *matrice de description de la géométrie*. La matrice `gd` est définie par l'utilisateur ; il s'agit d'un vecteur colonne construit comme suit :

- la première case contient une valeur entière entre 1 et 4 indiquant le type de géométrie : 1 pour le cercle, 2 pour un polygone, 3 pour un rectangle, 4 pour une ellipse.
- Dans le cas du polygone et du rectangle, les cases suivantes sont remplies ainsi :
- la deuxième case contient le nombre de points sur le bord, noté `nbm` ;
- les `nbm` cases suivantes contiennent les abscisses des points sur le bord ;
- les `nbm` dernières cases contiennent les ordonnées des points sur le bord.

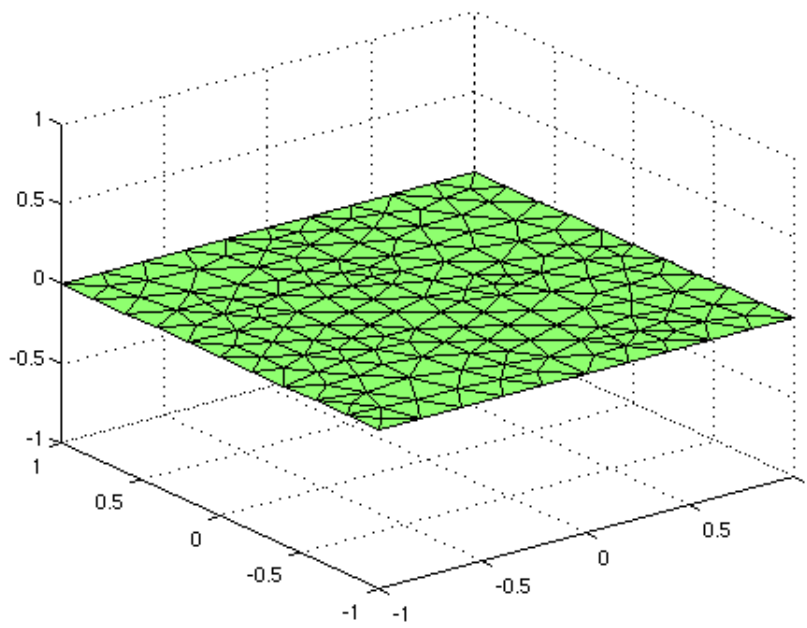
L'intérêt de définir autant de points sur le bord (alors qu'il en suffirait de quatre pour tracer le domaine) est que lors de l'appel de la fonction `initmesh`, Matlab va conserver ces points comme noeuds du maillage. Attention, on ne peut pas imposer le nombre exact de noeuds sur le bord : le nombre de points que l'on a définis dans la matrice `gd` nous donne uniquement un ordre de grandeur de la finesse du maillage.

2. **Construction des matrices :** la construction des matrices \mathbf{A} et \mathbf{B} utilise la fonction `assema`, dont les paramètres d'entrée sont les matrices `p` et `t` générées par `initmesh`, ainsi que des coefficients qui correspondent à l'équation que l'on cherche à résoudre. (Le lecteur pourra se référer à l'aide pour de plus amples informations sur le choix de ces coefficients et sur le panel d'équations qu'il est possible de traiter à l'aide des fonctions `assema` ou `asempde`.) Les conditions au bord sont traitées en ajoutant des coefficients de pénalisation $1/\varepsilon$ dans la matrice \mathbf{A} . Nous ne donnerons pas plus de détail sur la justification mathématique de cette opération (qui fonctionne, comme nous le verrons plus loin).
3. **Résolution du problème matriciel :** le calcul des valeurs propres et des vecteurs propres associés se fait grâce à la fonction `eigs`, qui nécessite d'une part de travailler avec des matrices creuses, et d'autre part de définir une matrice $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$ de sorte que notre problème s'écrive sous la forme usuelle $\mathbf{C}\mathbf{U}_h = \lambda_h\mathbf{U}_h$.
 - Les paramètres d'entrée de la fonction `eigs` sont la matrice \mathbf{C} , le nombre de valeurs propres cherchées et une valeur réelle *sigma*, valeur autour de laquelle la fonction va chercher les valeurs propres.
 - Ses paramètres de sortie sont une matrice \mathbf{V} , dont chaque colonne correspond à un vecteur propre, et une matrice diagonale \mathbf{D} , dont la diagonale est constituée des valeurs propres calculées.
4. **Affichage des résultats :** le tracé de fonctions de deux variables à valeur dans \mathbb{R} se fait grâce à la commande `trisurf`. Cette commande utilise la transposée de la matrice `tri`, qui est formée des trois premières lignes de la matrice `t` du maillage. Ainsi chaque colonne de `tri` contient les numéros des sommets de chaque triangle du maillage. Pour de plus amples informations, consulter l'aide bien sûr.

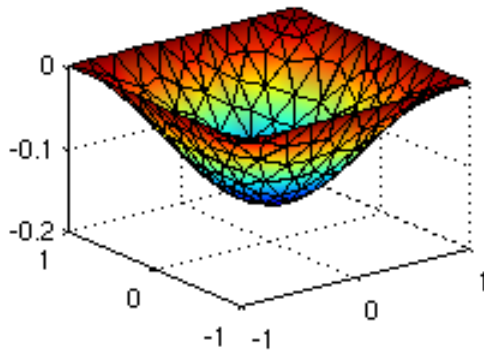
```

File Edit Text Cell Tools Debug Desktop Window Help
[Icons] Stack: Base /
1 %calcul des valeurs propres et des fonctions propres du laplacien...
2 %sur un domaine carré avec conditions au bord de Dirichlet, ...
3 %par une méthode d'éléments finis
4
5
6 - clc;
7 - clear all;
8
9 - epsilon=1/10000000;%coefficient de pénalisation 10 puissance -7
10
11 %construction du maillage
12 - nbm=10; %nombre de segments définis sur chaque côté
13 - tm=2/nbm;%distance entre deux points du bord
14 - nbp=nbm*4%nombre de points sur le bord du carré
15
16 %construction du vecteur gd donnant les coordonnées des points du bord
17 - gd(1,1)=3; %la valeur 3 indique un domaine rectangulaire
18 - gd(2,1)=nbp;
19
20 %on parcourt le bord dans le sens trigonométrique...
21 %en partant du coin inférieur gauche
22 - for i=1:nbm
23 -     gd(2+i,1)=-1+tm*(i-1);
24 -     gd(2+i+nbp,1)=-1;
25 -     gd(2+nbp+i,1)=1;
26 -     gd(2+nbp+i+nbp,1)=-1+tm*(i-1);
27 -     gd(2+nbp*2+i,1)=1-tm*(i-1);
28 -     gd(2+nbp*2+i+nbp,1)=1;
29 -     gd(2+nbp*3+i,1)=-1;
30 -     gd(2+nbp*3+i+nbp,1)=1-tm*(i-1);
31 - end
32
33 %construction des matrices p,e,t représentant le maillage
34 - d1=decsd(gd);
35 - [p,e,t]=initmesh(d1);
36
37 %construction de la matrice de rigidité A et de la matrice de masse B
38 - [A,B,F]=assema(p,t,1,1,0);
39
40 %traitement des conditions au bord de Dirichlet par pénalisation
41 - taille_A=length(p(1,:));%taille de la matrice A, qui correspond au nombre...
42 %de noeuds du maillage
43
44 %implémentation de la condition de Dirichlet par pénalisation des...
45 %coefficients de la matrice A correspondant aux points du bord
46 - for j=1:taille_A
47 -     if p(2,j)==-1 %bord inférieur
48 -         A(j,j)= A(j,j)+1/epsilon;
49 -     elseif p(1,j)==1 %bord droit
50 -         A(j,j)= A(j,j)+1/epsilon;
51 -     elseif p(2,j)==1 %bord supérieur
52 -         A(j,j)= A(j,j)+1/epsilon;
53 -     elseif p(1,j)==-1 %bord gauche
54 -         A(j,j)= A(j,j)+1/epsilon;
55 -     end
56 - end
57
58 %on résout AU=1BU
59 %pour cela il faut transformer A et B en matrices creuses
60 - A=sparse(A);
61 - B=sparse(B);
62 - C=inv(B)*A;%on se ramène au problème de valeurs propres standard CU=1U
63 - sigma=1;%valeur autour de laquelle on cherche les valeurs propres
64 - [V D]=eigs(C,4,sigma);%D est une matrice diagonale...
65 %contenant les valeurs propres, V est la matrice des fonctions propres, ...
66 %4 est le nombre de valeurs propres cherchées
67
68 %affichage du maillage
69 - tri=t(1:3,:);
70 - for i=1:taille_A
71 -     zero(i)=0;
72 - end
73 - trisurf(tri',p(1,:)',p(2,:)',zero');
74 - pause;
75
76 %représentation des fonctions propres
77 - for i=1:4
78 -     subplot(2,2,i);
79 -     trisurf(tri',p(1,:)',p(2,:)',V(:,i),'facecolor','interp');
80 -     title(['fonction propre pour la valeur propre : ',num2str(D(i,i))]);
81 - end

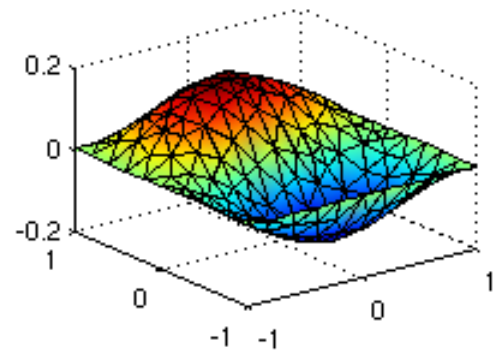
```



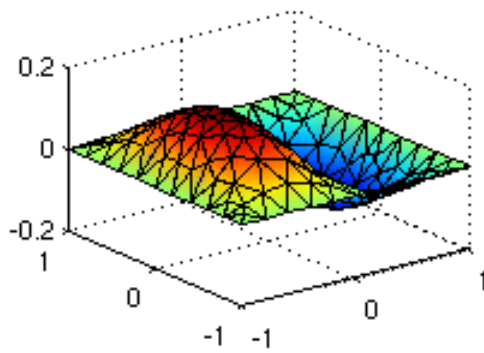
fonction propre pour la valeur propre :4.9846



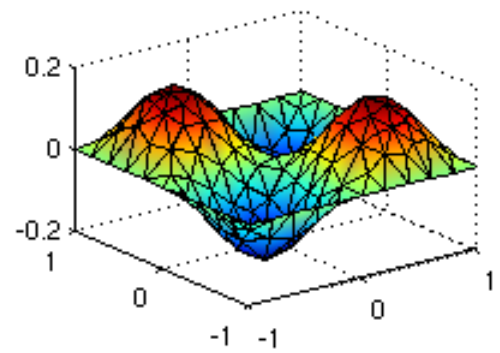
fonction propre pour la valeur propre :12.6333



fonction propre pour la valeur propre :12.6368



fonction propre pour la valeur propre :20.5017



Bon courage à tous et bienvenue dans le monde de Matlab et Octave. Bonne programmation!!!

Références

- [1] Vous, la touche F1
- [2] S. Balac, [http ://perso.univ-rennes1.fr/stephane.balac/matlab/matlab.html](http://perso.univ-rennes1.fr/stephane.balac/matlab/matlab.html)
- [3] Et bien d'autre encore sur Google