Matlab pas à pas

M. Bonnivard[1]*, M. Ersoy[1]†

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

16 septembre 2008

Table des matières

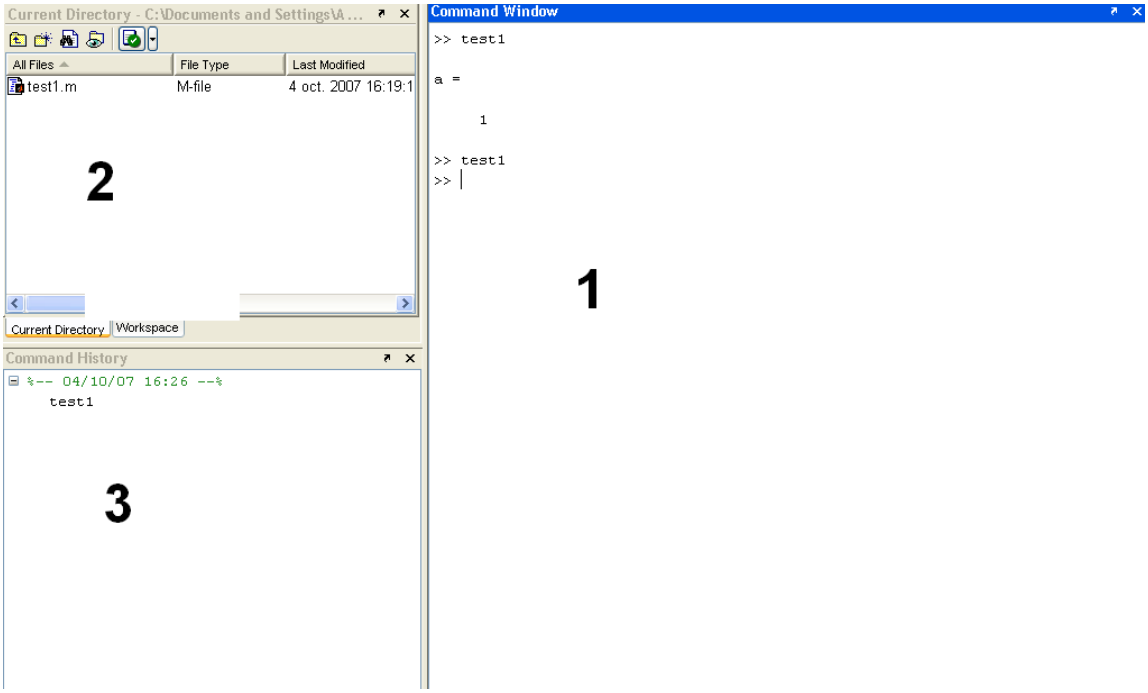
* *Matthieu.Bonnivard@etu.univ-savoie.fr*
† *Mehmet.Ersoy@univ-savoie.fr*

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.

2 Prise en main

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.
Matlab est muni de l'interface suivante :



L'encadré 1 (COMMAND WINDOW) correspond à la fenêtre de commande. Toute instruction est précédée du sigle » 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 (par exemple) test1.m
Ensuite, tapez `test1` dans la fenêtre de commande et observez. Vous venez d'exécuter votre première ligne de commande. A l'écran apparait alors

```
>> test1

a =

     1

>>
```

Notez que l'instruction précédente se termine par le nombre 1. On peut dcider 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 ;`. Executer 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. Donc il est inutile de vous dire que 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 fleches haut. 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 touche tab et voyez que vous avez accès à toutes les fonctions préprogrammées de Matlab, et aussi aux fichiers que vous aurez 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.

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) et compléter le code par

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

et exécuter. Matlab nous donne comme réponse

```
c =

     2

>>
```

On vient donc d'affecter la valeur 2 à c par l'opération $a + b$ sans a et b. Qu'advient-il alors de a et b ? Pour le savoir, tapez `a` dans la fenêtre de commande puis la touche entrée et pareillement pour b comme suit

```
>> a

a =
```

```
     1

>> b

b =

     1

>>
```

On voit alors 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 ligne de commande `clear a` , soit en ligne de commande (dans ce cas, la variable a est effacée ponctuellement ; c'est à dire, 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

>>
```

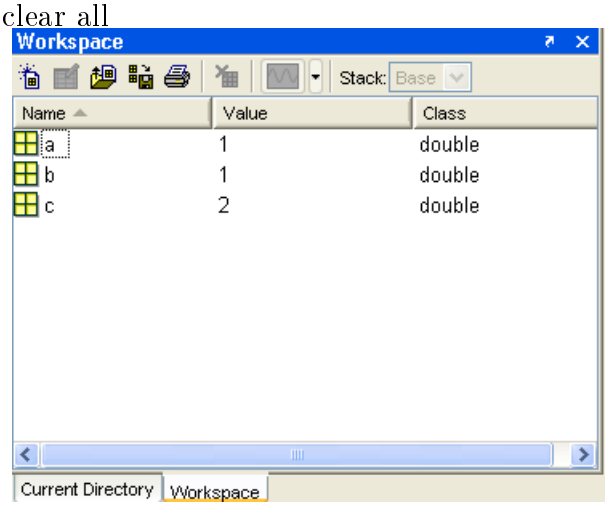
Enfin, la commande `clear` $var1\, var2\, var3 \ldots varn$ permet d'effacer toutes les variables $vari$, pour $i = 1 \ldots n$.

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 on exécute le fichier test1.m dont le contenu est

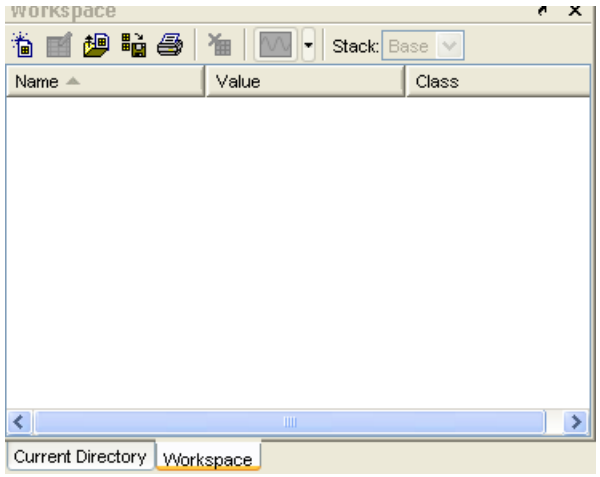
```
a = 1;
b = a;
c = a+b
```

La fenêtre workspace affiche

– avant l'exécution de clear all



– après l'execution de clear all



Maintenant que notre fenêtre de travail est bien encombrée, on peut la nettoyer avec la commande `clc`. Et voila, 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`)

```
% 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 allegees

v3 = transpose(v2) % transpose le vecteur v qui devient
   % donc un vecteur colonne
v4 = v1' % ' transpose le vecteur v avec une notation allegee
```

```
% 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 differentes


% 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'incrémentation. Exemple,

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

v =

     1     3     5     7
```

Pour définir un vecteur à incrément à pas constant de pas 1, on utilise soit `v = [a :1 :b]` soit `v = [a :b]`.

```
>> 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'élément

```
>> 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 connaêtre 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


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, connaêtre la taille,... Ceci est illustré dans 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
```

Remark 3.1 Bien qu'une matrice soit déjà définie, rien 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 $6^{\text{ème}}$ 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 $4^{\text{ème}}$ 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 $4^{\text{ème}}$ vecteur ligne de A et $y = A(:, 6)$ au $6^{\text{ème}}$ 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 tels que *eye, ones, zeros, magic,...*

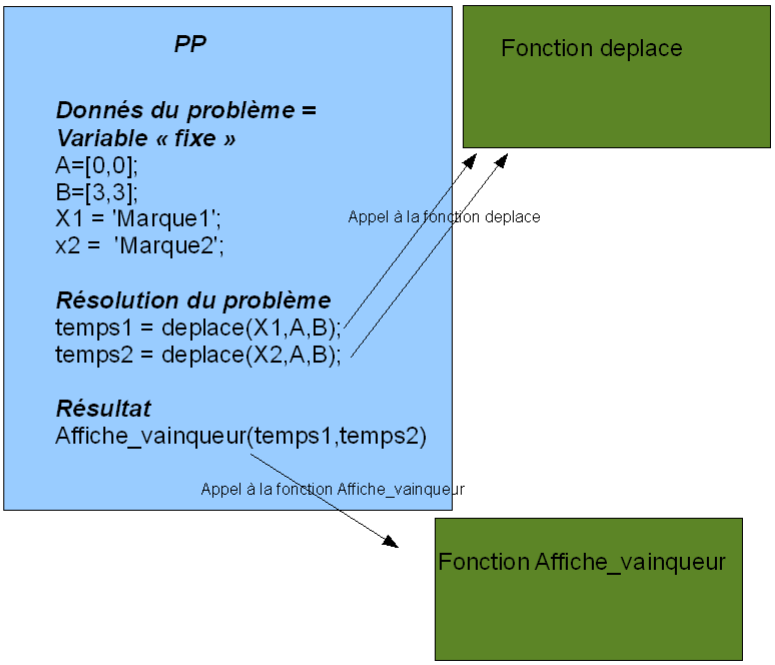
```
>> eye(3)

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 Programmation

4.1 Structure du programme

De manière général, un programme constitue le squelette de notre raisonnement. Par exemple pour programmer le temps de trajet effectué par une voiture de marque X qui se déplace d'un point A à un point B, le programme principal (pp) doit permettre le calcul du parcours de temps avec toute marque de voiture et de se déplacer d'un point à un autre connu. Pour ce faire, le pp fera appel à la fonction "déplace" avec comme données les points A et B ainsi que la marque de la voiture X.
pp sera constitué des instructions : le point A, B, X (les données du problème) et de la fonction déplace(X,A,B) qui nous renvoie le temps de parcours selon le modèle X et la distance séparant A de B.



Dans cette figure, on montre un exemple de construction possible de programme. Nous verrons dans le paragraphe suivant comment définir une fonction.
En bref, un programme est généralement construit sous la forme

1. Les 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 le conflit de variables entre deux exécutions.

4.2 Déclaration de fonctions types *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 ne possède pas d'environnements pour les procédures.
On définit une fonction de la manière suivante

```
function [y1,...,ym]=nom_de_la_function(x1,...,xn)
% CORPS DE LA FONCTION
return
```

Les paramètres $x1, \ldots, xn$ sont les paramètres d'entrées (connu par le pp) tandis que $y1, \ldots, ym$ sont les paramètres de sorties calculés en fonction de $x1, \ldots, xn$. On fait appel à la fonction `nom_de_la_function` par :

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

On affecte le résultat de la fonction `nom_de_la_function` aux variables `z1,...,zm` grâce aux données `a1,...,an`.
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 $x = x0 = 3$ où $x0$ 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 on se contente de changer $x0 = 3$ par une liste de points $x0 = 1, 2, 3, 4, 5$ et d'executer le pp, que se passe-t-il ? On voit apparaitre 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 introduite 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. Qu'est ce que ce charabia ? Cela signifie tout simplement que x^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 $x0$ défini par la suite n'est pas carré ! On doit donc faire faire une évaluation de la liste points par points ; je vous rappelle que .^ permet ceci. 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 ca marche ! Fort heureusement. Remarquez aussi que le paramêtre de sortie de fun1 (y) s'adapte au type du paramêtre d'entrée. L'entrée étant un vecteur, la sortie est aussi un vecteur. L'adaptativité du type est un point très agréable en Matlab.

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 l'on a une fonction qui dépend de $x1, \ldots, xn$ ($='x1','x2',...,'xn'$). L'appel à `nom_de_la_fonction` se fait de la manière suivante : étant donnée $x1, \ldots, xn$,

```
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 avec 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*.

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 apercu 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 $f[x_0, \ldots, x_k]$ les différences divisées associées aux points $x_0, \ldots, x_k$ avec $(f[x_i])_{i=0,\ldots,n}$ connus . Il s'agit de calculer ces différentes valeurs pour $k = 1, \ldots, n$.

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 $(x_n)_{n\geq 0}$ tels 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 stoker la matrice identité de taille 10000.

```
[STOCKAGE 1 ]

tic
A = sparse(eye(10000));
toc

tic
A = eye(10000);
toc
```

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

Remark 4.1 Pour stoppez une exécution en cas de problème *Ctrl+c*

Compilez et observez.

5 Représentation de nos résultats

5.1 Representation 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 crée,

```
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. 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 sÃ»r 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, vous souhaitez obtenir deux résultats graphiques sur une même fenêtre alors il vous suffit de diviser cette fenêtre par la commande

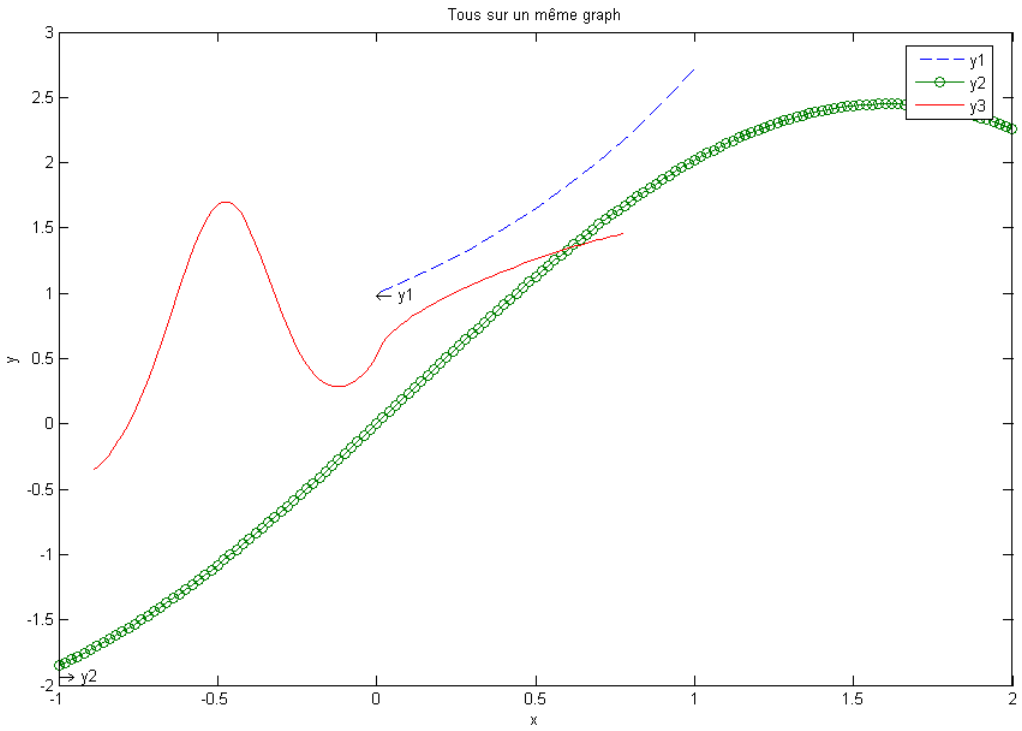
```
subplot
```

. L'utilisation de cette dernière est la suivante

```
subplot(l,c,p)
```

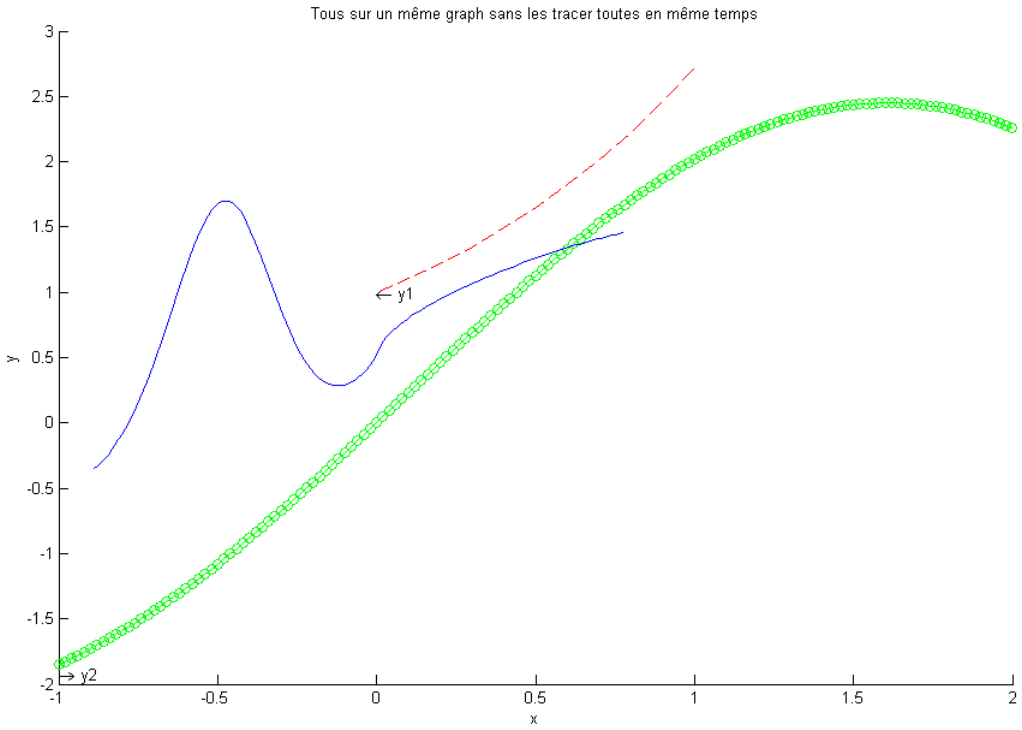
où l'indice l représente le nombre de divisions verticales, c le nombre de divisions horizontales, et p la position de votre graphe dans la fenêtre sachant que l'énumération se fait de gauche à droite et commence par 1.
Illustration :

```
[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 m\^eme
graphe') xlabel('x') ylabel('y') text(0,1,'\leftarrow y1')
text(x2(1),y2(1),'\rightarrow y2') legend('y1','y2','y3')
```

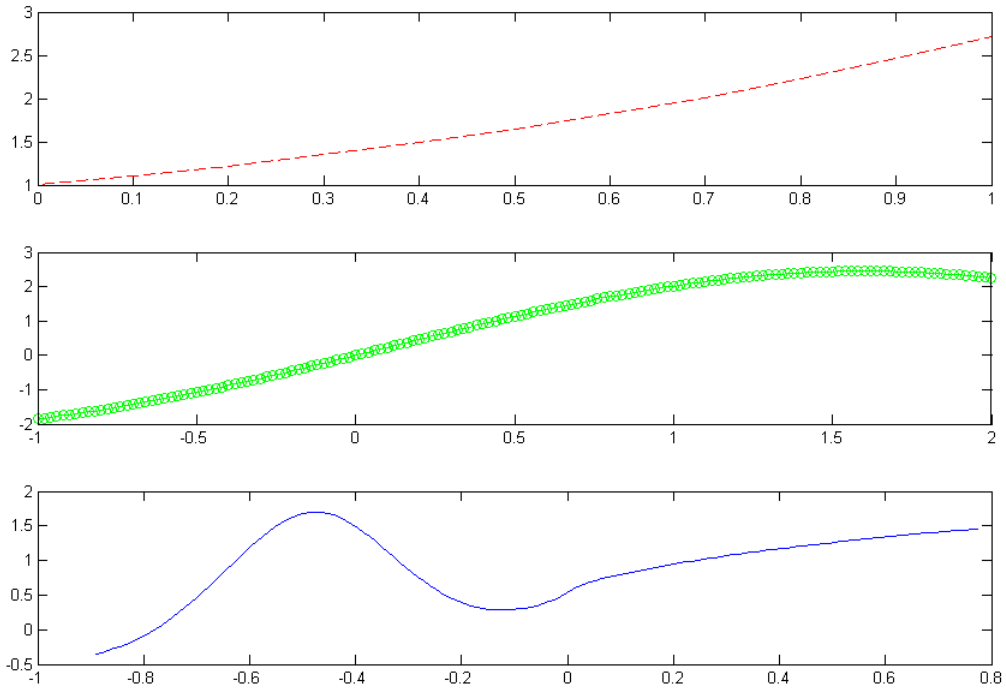



```
[cas 2]

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)
```

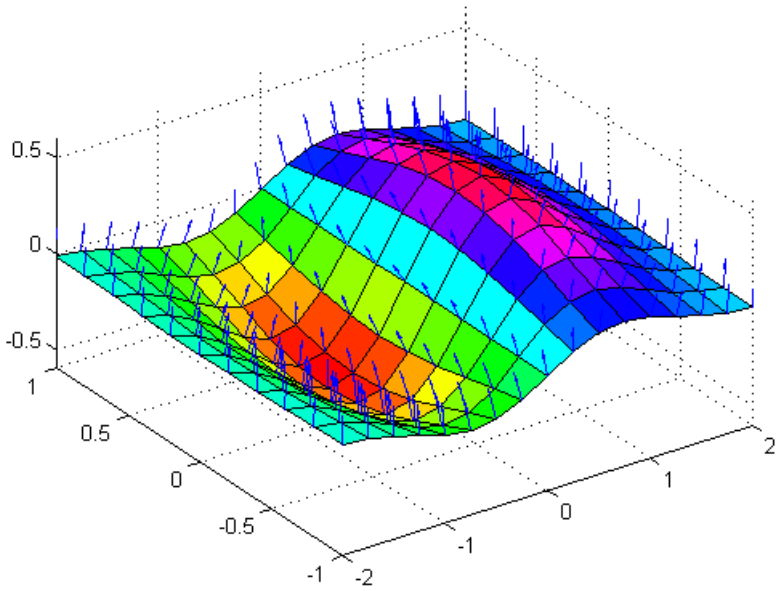



```
[cas 3]
subplot(311)
plot(x1,y1,'--','color','red')

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

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

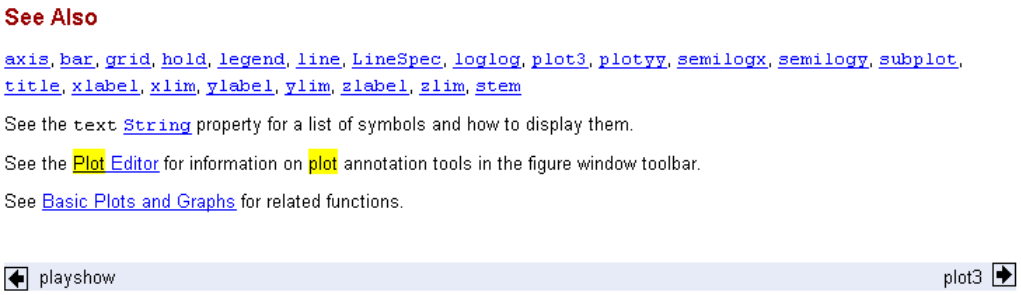


Il existe aussi d'autre types de commande pour la représentation graphique, par exemple `plot3`

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



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



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 format étant valable).

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

% 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 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*. Pour charger en mémoire les données du fichier on utilise la commande *load*. Prenons par exemple, le fichier test_ecriture.txt en vue d'en extraire les données.

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

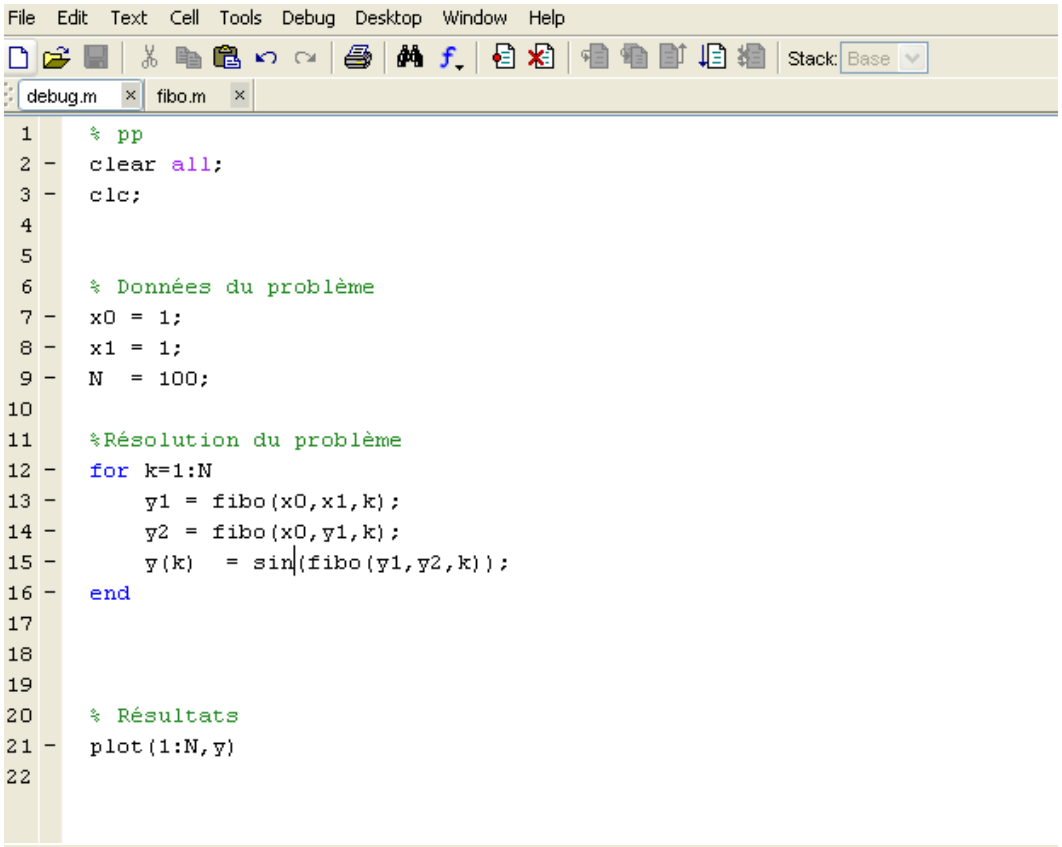
Observez le type de X. Matlab voit les données comme une matrice à 2 lignes et 11 colonnes vu la répartition de ce dernier. On peut donc extraire les données et tracer la fonction énoncée dans la sous-section précédente.

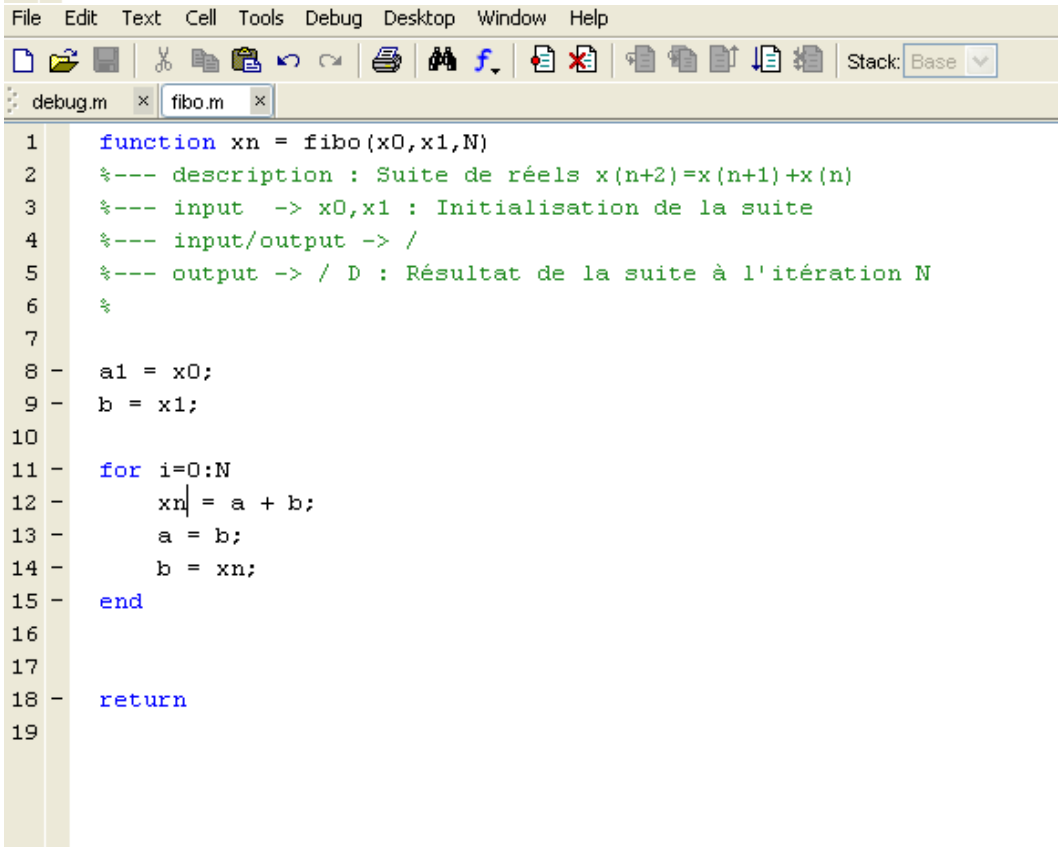
```
plot(X(1,:),X(2,:))
```

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 incomprehensibles 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 :

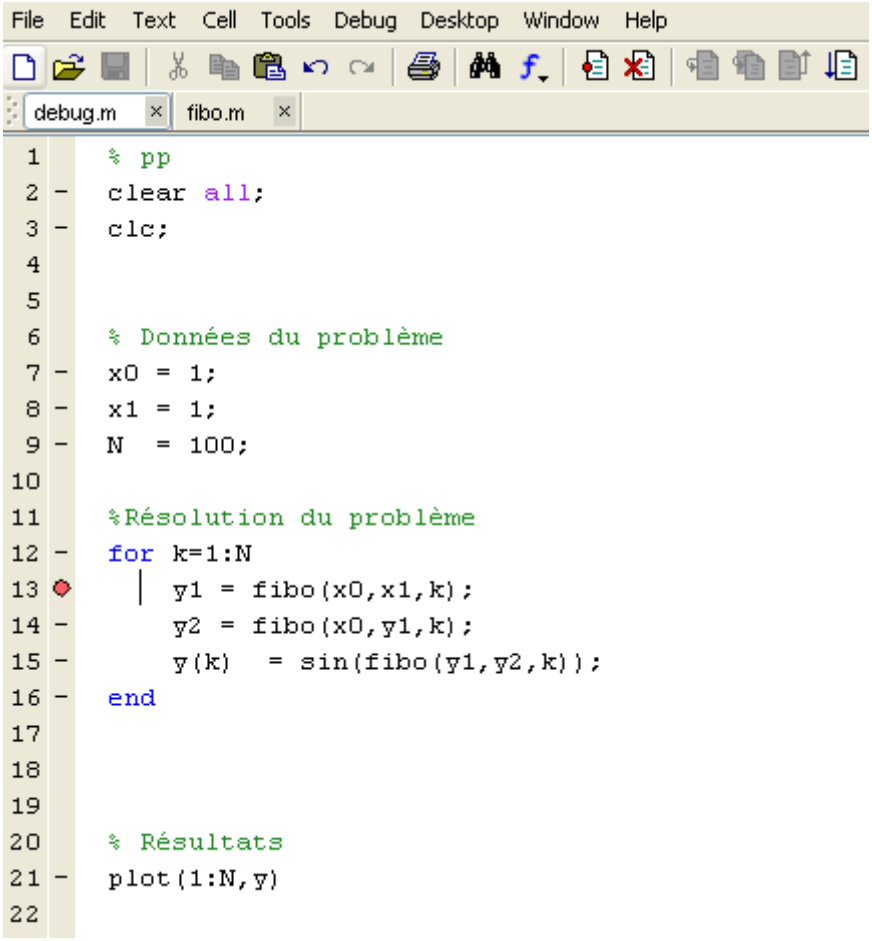




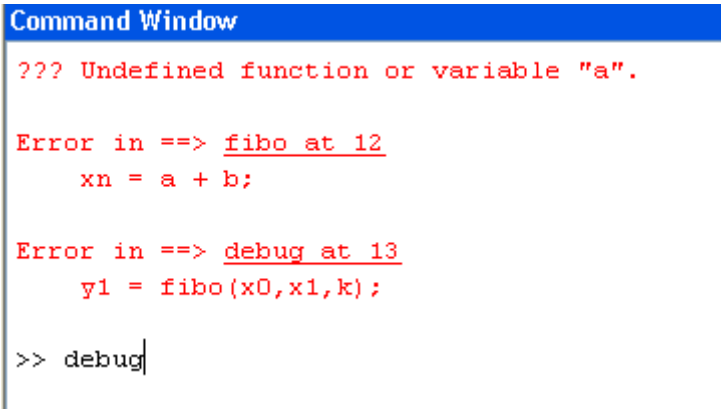
Exécutez le programme principal et observez le message d'erreur suivant :

```
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);

>>
```

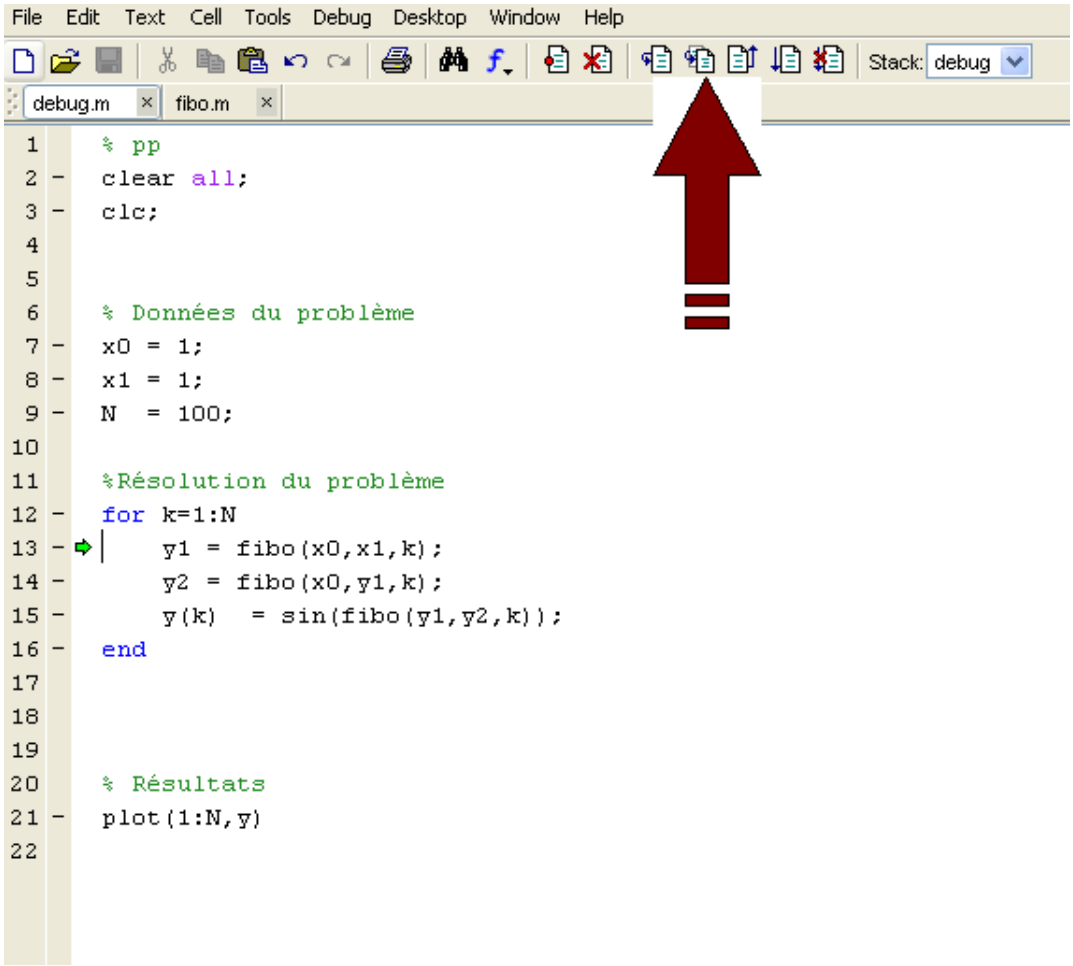
D'après le message d'erreur ci-dessus, il est clair que dans la fonction `fibo` il suffit de remplacer `a1` par `a` pour résoudre le problème mais supposons que nous ne l'ayons pas vu (ce qui pourrait vous arriver...).
Commençons à debugger le programme. Si l'on pense que l'erreur provient de la fonction `fibo`, il suffit de placer le curseur à la ligne où la fonction fibo est appelée d'appuyer sur la touche F12. On voit alors apparaitre

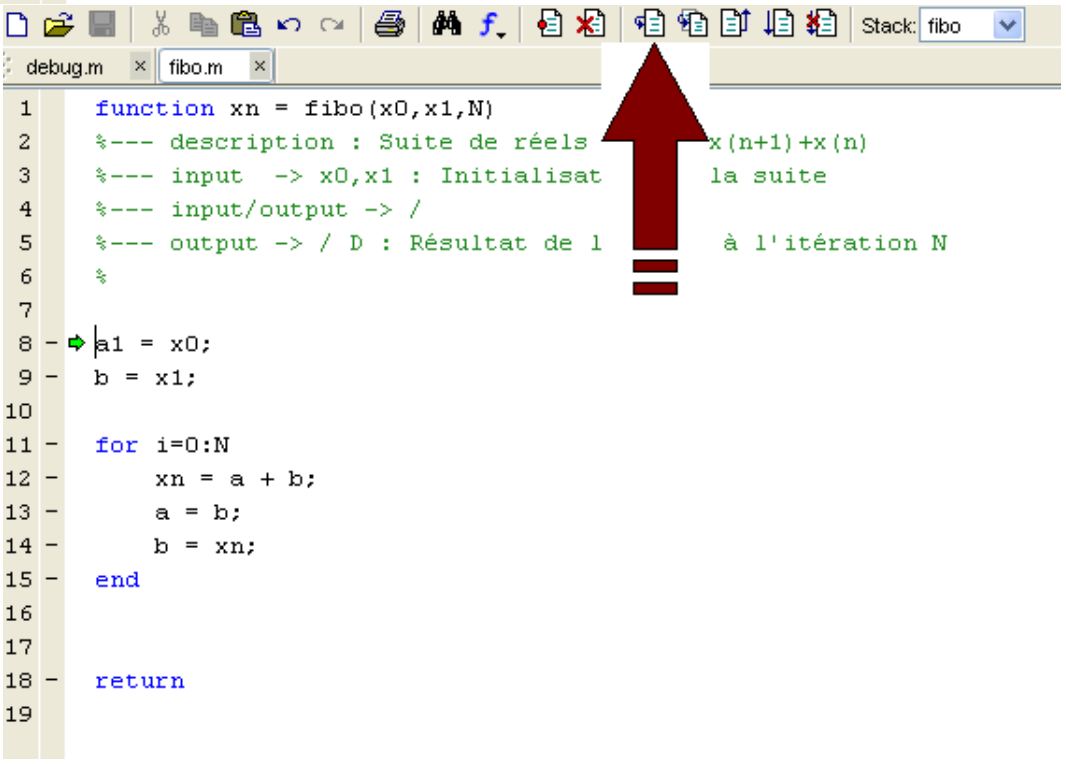


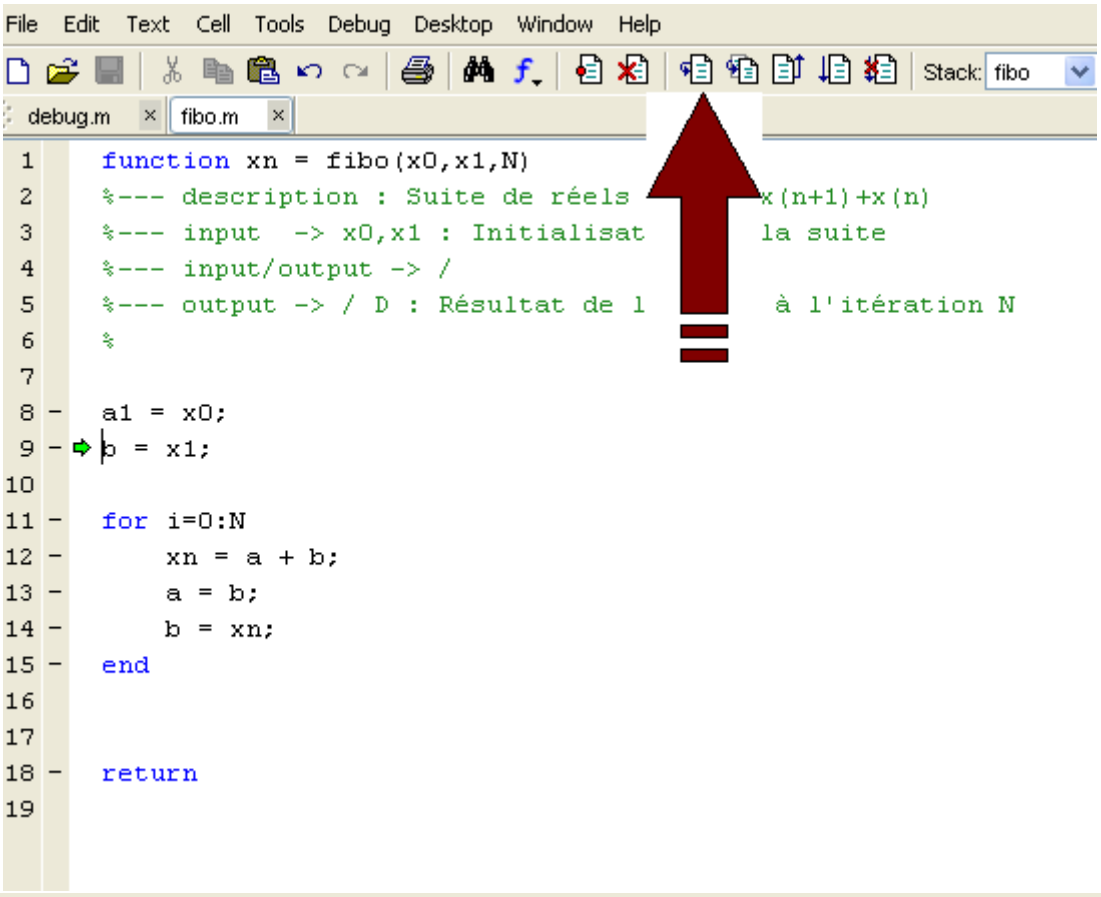
Si l'on ne sait pas du tout ou se trouve l'erreur, on place le curseur en tout début de programme principal et on appuie sur la touche F12, ce qui donne :

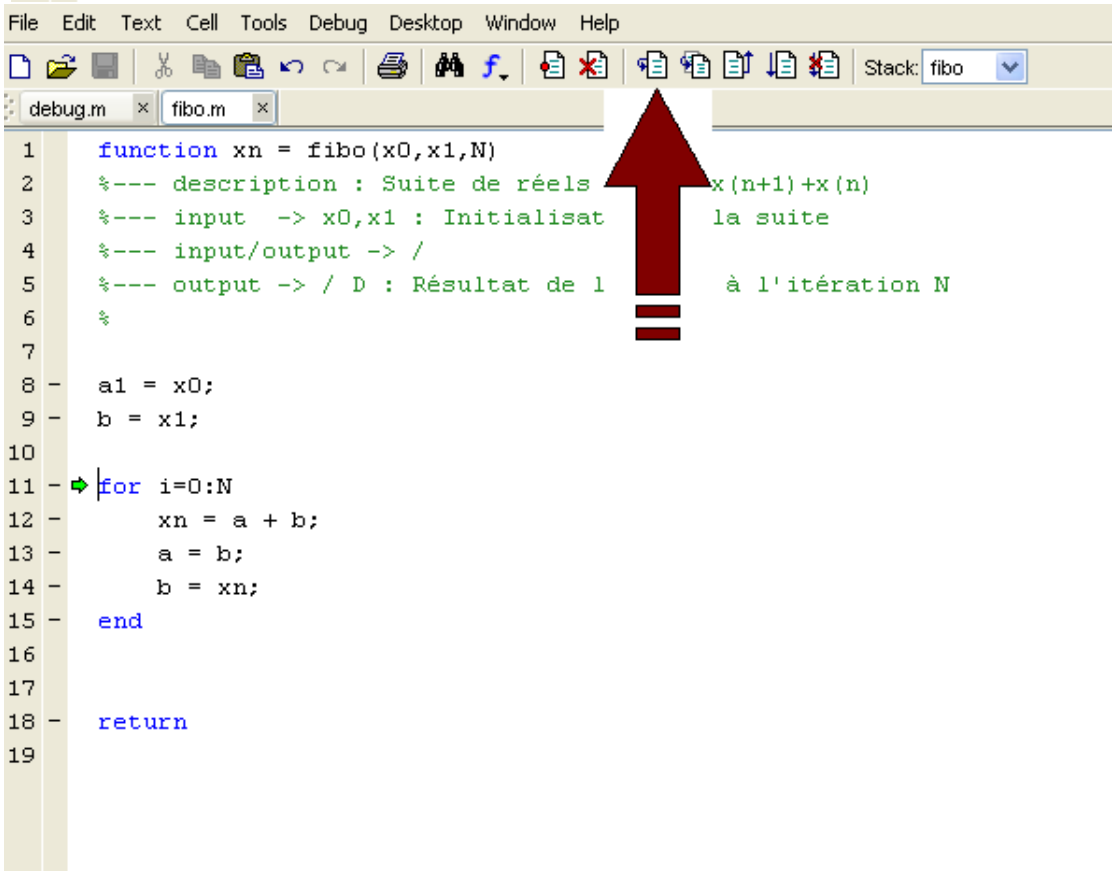


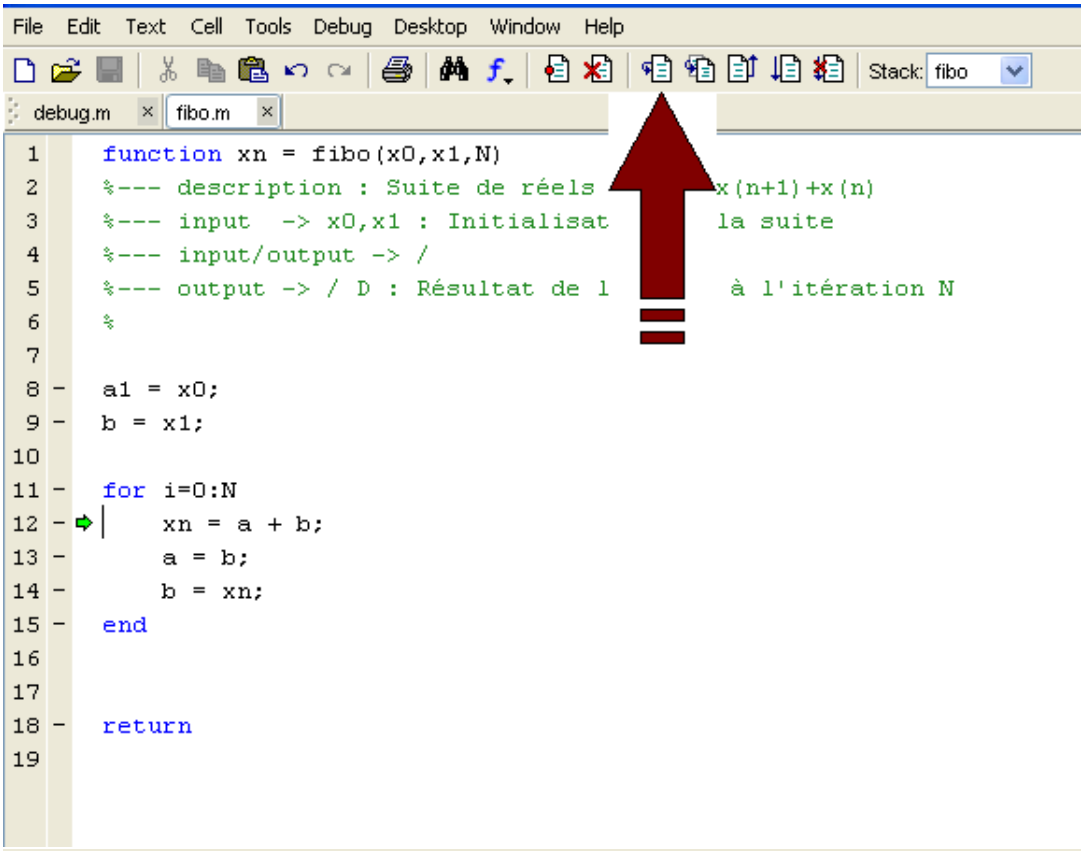
Ensuite, on procède comme dans l'illustration qui suit :

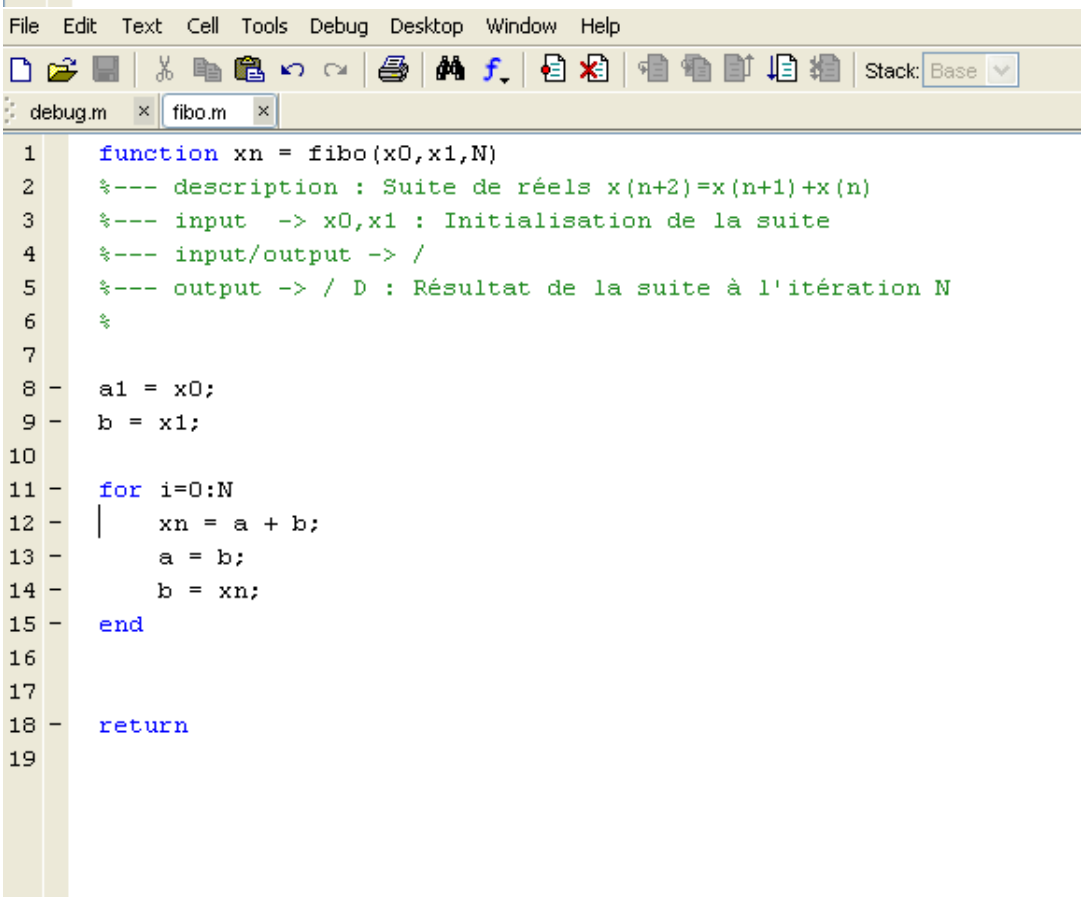

File Edit Text Cell Tools Debug Desktop Window Help
Stack: debug
debug.m
fibo.m
% pp
clear all;
clc;
% Données du problème
x0 = 1;
x1 = 1;
N = 100;
%Résolution du problème
for k=1:N
y1 = fibo(x0,x1,k);
y2 = fibo(x0,y1,k);
y(k) = sin(fibo(y1,y2,k));
end
% Résultats
plot(1:N,y)


Stack: fibo
debug.m
fibo.m
function xn = fibo(x0,x1,N)
%--- description : Suite de réels x(n+1)+x(n)
%--- input -> x0,x1 : Initialisat la suite
%--- input/output -> /
%--- output -> / D : Résultat de l à l'itération N
%
a1 = x0;
b = x1;
for i=0:N
xn = a + b;
a = b;
b = xn;
end
return


File Edit Text Cell Tools Debug Desktop Window Help
Stack: fibo
debug.m
fibo.m
function xn = fibo(x0,x1,N)
%--- description : Suite de réels x(n+1)+x(n)
%--- input -> x0,x1 : Initialisat la suite
%--- input/output -> /
%--- output -> / D : Résultat de l à l'itération N
%
a1 = x0;
b = x1;
for i=0:N
xn = a + b;
a = b;
b = xn;
end
return


File Edit Text Cell Tools Debug Desktop Window Help
Stack: fibo
debug.m
fibo.m
function xn = fibo(x0,x1,N)
%--- description : Suite de réels x(n+1)+x(n)
%--- input -> x0,x1 : Initialisat la suite
%--- input/output -> /
%--- output -> / D : Résultat de l à l'itération N
%
a1 = x0;
b = x1;
for i=0:N
xn = a + b;
a = b;
b = xn;
end
return


File Edit Text Cell Tools Debug Desktop Window Help
Stack: fibo
debug.m
fibo.m
function xn = fibo(x0,x1,N)
%--- description : Suite de réels x(n+2)=x(n+1)+x(n)
%--- input -> x0,x1 : Initialisat la suite
%--- input/output -> /
%--- output -> / D : Résultat de l à l'itération N
%
a1 = x0;
b = x1;
for i=0:N
xn = a + b;
a = b;
b = xn;
end
return


File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
debug.m
fibo.m
function xn = fibo(x0,x1,N)
%--- description : Suite de réels x(n+2)=x(n+1)+x(n)
%--- input -> x0,x1 : Initialisation de la suite
%--- input/output -> /
%--- output -> / D : Résultat de la suite à l'itération N
%
a1 = x0;
b = x1;
for i=0:N
xn = a + b;
a = b;
b = xn;
end
return

```
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);

>>
```

Attention : cette manipulation peut être longue!!!
Bon courage à tous et bienvenue dans le monde de Matlab. Bonne programmation!!!

7 Un exemple de programme : 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

$$\begin{cases} \triangle u &= \lambda u & \text{dans } \Omega, \\ u &= 0 & \text{sur } \partial\Omega. \end{cases} \tag{1}$$

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

$$\begin{cases} \triangle u_h &= \lambda_h u_h & \text{dans } \Omega, \\ u_h &= 0 & \text{sur } \partial\Omega. \end{cases} \tag{2}$$

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 `p` est une matrice à deux colonnes : chaque ligne de `p` contient les coordonnées des points du maillage. La fonction `initmesh` prend en entrée la matrice `dl` 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 :

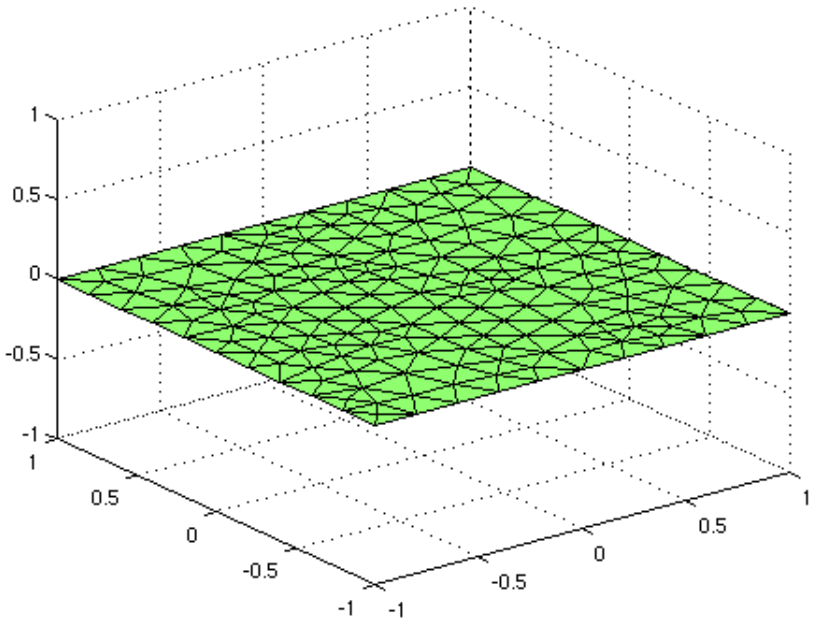
- la première case contient une valeur entière entre 1 et 4 indiquant le type de géométrie : 1 pour le cercle, 2 pour un polygône, 3 pour un rectangle, 4 pour une ellipsoÃ¯de. Dans le cas du polygône et du rectangle, les cases suivantes sont remplies ainsi :
- la deuxième case contient le nombre de points 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.

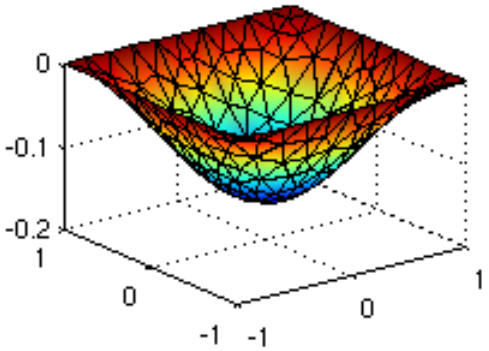
2. **Construction des matrices :** la construction des matrices `A` et `B` utilise la fonction `assema`, dont les paramètres d'entrée sont les matrices `p` et `t` générées par `initmesh`, ainsi que des coefficients qui correspondent à l'équation que l'on cherche à résoudre. (Le lecteur pourra se référer 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 `assempde`.) Les conditions au bord sont traitées en ajoutant des coefficients de pénalisation $1/\varepsilon$ dans la matrice `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 $C = B^{-1}A$ de sorte que notre problème s'écrive sous la forme usuelle $CU_h = \lambda_h U_h$.
   - Les paramètres d'entrée de la fonction `eigs` sont la matrice 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 V, dont chaque colonne correspond à un vecteur propre, et une matrice diagonale 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 sÃ»r.

```
File  Edit  Text  Cell  Tools  Debug  Desktop  Window  Help
```

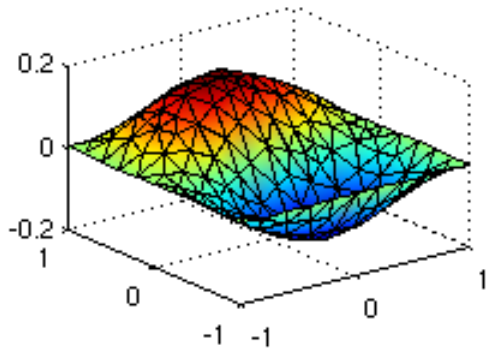
```
%calcul des valeurs propres et des fonctions propres du laplacien...
%sur un domaine carré avec conditions au bord de Dirichlet, ...
%par une méthode d'éléments finis


clc;
clear all;

epsilon=1/10000000;%coefficient de pénalisation 10 puissance -7

%construction du maillage
nbm=10; %nombre de segments définis sur chaque côté
tm=2/nbm;%distance entre deux points du bord
nbp=nbm*4%nombre de points sur le bord du carré

%construction du vecteur gd donnant les coordonnées des points du bord
gd(1,1)=3; %la valeur 3 indique un domaine rectangulaire
gd(2,1)=nbp;

%on parcourt le bord dans le sens trigonométrique...
%en partant du coin inférieur gauche
for i=1:nbm
    gd(2+i,1)=-1+tm*(i-1);
    gd(2+i+nbp,1)=-1;
    gd(2+nbm+i,1)=1;
    gd(2+nbm+i+nbp,1)=-1+tm*(i-1);
    gd(2+nbm*2+i,1)=1-tm*(i-1);
    gd(2+nbm*2+i+nbp,1)=1
    gd(2+nbm*3+i,1)=-1;
    gd(2+nbm*3+i+nbp,1)=1-tm*(i-1);
end

%construction des matrices p,e,t représentant le maillage
dl=decsg(gd);
[p,e,t]=initmesh(dl);

%construction de la matrice de rigidité A et de la matrice de masse B
[A,B,F]=assema(p,t,1,1,0);

%traitement des conditions au bord de Dirichlet par pénalisation
taille_A=length(p(1,:));%taille de la matrice A, qui correspond au nombre...
%de noeuds du maillage

%implémentation de la condition de Dirichlet par pénalisation des...
%coefficients de la matrice A correspondant aux points du bord
for j=1:taille_A
     if p(2,j)==-1 %bord inférieur
         A(j,j)= A(j,j)+1/epsilon;
     elseif p(1,j)==1 %bord droit
            A(j,j)= A(j,j)+1/epsilon;
     elseif p(2,j)==1 %bord supérieur
            A(j,j)= A(j,j)+1/epsilon;
     elseif p(1,j)==-1 %bord gauche
            A(j,j)= A(j,j)+1/epsilon;
     end
end

%on résout AU=lBU
%pour cela il faut transformer A et B en matrices creuses
A=sparse(A);
B=sparse(B);
C=inv(B)*A;%on se ramène au problème de valeurs propres standard CU=lU
sigma=1;%valeur autour de laquelle on cherche les valeurs propres
[V D]=eigs(C,4,sigma);%D est une matrice diagonale...
%contenant les valeurs propres, V est la matrice des fonctions propres, ...
%4 est le nombre de valeurs propres cherchées

%affichage du maillage
tri=t(1:3,:);
for i=1:taille_A
    zero(i)=0;
end
trisurf(tri',p(1,:)',p(2,:)',zero');
pause;

%representation des fonctions propres
for i=1:4
    subplot(2,2,i);
    trisurf(tri',p(1,:)',p(2,:)',V(:,i),'facecolor','interp');
    title(['fonction propre pour la valeur propre :',num2str(D(i,i))]);
end
```



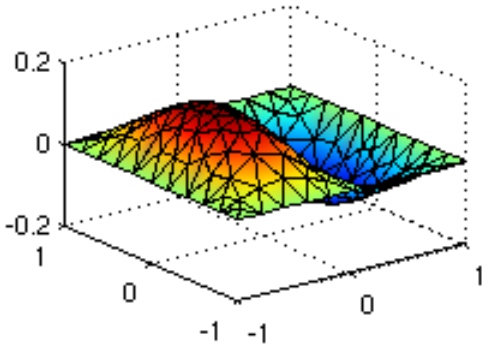
fonction propre pour la valeur propre :4.9846



fonction propre pour la valeur propre :12.6333



fonction propre pour la valeur propre :12.6368



fonction propre pour la valeur propre :20.5017