Chapitre 1

Programmer avec Matlab

Dans ce chapitre, nous allons montrer comment utiliser l'environnement de programmation Matlab pour développer des programmes.

1.1. Introduction

On distingue deux types de fichiers dans Matlab, également appelés m-files : les programmes scripts (programme principal) et les fonctions (sous-programmes).

Un m-file est créé, soit depuis le menu en cliquant sur Fichier (`New > M-File`), soit depuis l'invite commande en tapant :

```
>> edit monfichier.m
```

Pour exécuter un programme, Matlab doit le localiser. Pour cela il existe la variable d'environnement `path`. C'est l'ensemble de tous les emplacements des dossiers susceptibles de contenir des fichiers « `.m` ». Par défaut, Matlab cherche si le programme à exécuter est situé dans le répertoire courant (current directory). S'il ne le trouve pas, il cherche dans tous les dossiers spécifiés dans la variable `path`. S'il ne le trouve pas, il affiche un message d'erreur : file `not found`. Pour ajouter un répertoire dans la liste du `path` on utilise la commande `addpath().`

N.B :

Il est à noter que Matlab permet de réaliser de véritables applications très élaborées. Il utilise un langage de programmation interprété, c'est-à-dire qu'il n'y a aucune phase de compilation et les instructions du code sont directement exécutées à leur lecture.

1.2. Les programmes script ou fichiers d'instructions

Bien que Matlab offre son propre fichier éditeur (`Editor`), qui est un fichier texte avec une extension `.m`. Vous pouvez utiliser votre éditeur de texte préféré pour écrire vos programmes (bien sûr sans oublier de modifier l'extension).

Le programme script est une suite d'instructions Matlab enregistrée et exécutable. Chaque instruction est exécutée ligne par ligne comme si elle était tapée dans la fenêtre de commande. Toutes les variables utilisées par le programme sont stockées dans l'espace de travail (`workspace`), ce dernier est partagé par tous les scripts. Lorsque Matlab détecte une erreur, le compilateur s'arrête et un message d'erreur s'affiche à la fenêtre de commande avec le numéro de la ligne où l'erreur est détectée.

Illustration 1.1

Enregistrez dans le répertoire courant les lignes suivantes sous le nom `triangle.m` :

Editons notre script `monfichier.m`.
Enregistrez l'éditeur sous le nom `triangle.m`, puis saisir les instructions suivantes :

```
>> A = [0 -0.5 0.5 ; -0.5 0.5 0]
B = [-0.5 0 0.5 ; 0 0.5 -0.5]
plot(A,B)

A =

         0   -0.5000    0.5000
   -0.5000    0.5000         0
```

```
B =

   -0.5000         0    0.5000
         0    0.5000   -0.5000

>>
```

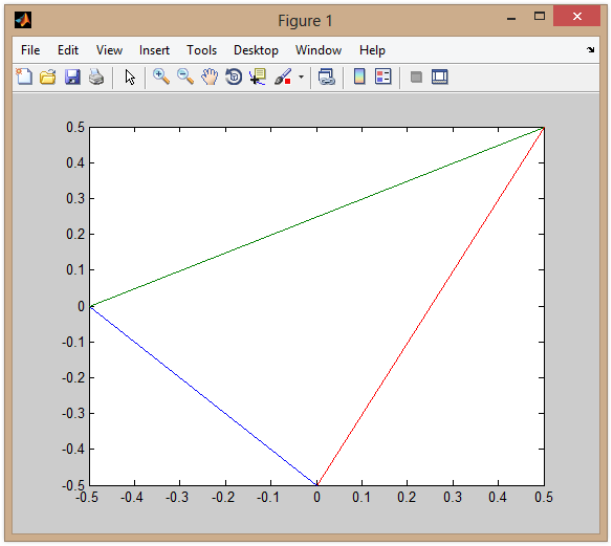


Figure 1.1. Affichage de tracé de l'illustration 1.1.

1.3. Exécution des programmes script

L'exécution complète des programmes script se fait soit :

- le script complet en cliquant sur le bouton « `run` » de l'éditeur (icône avec un triangle vert).
- ou en frappant f5.
- ou encore en tapant son nom (sans l'extension) à l'invite de commande.

Une autre alternative, c'est qu'on peut exécuter le script par section :

- en sélectionnant la section qu'on veut exécuter puis en tape f9.
- ou en clique par le bouton droit et choisissant évaluer la selection (`evaluate selection`).

La commande `triangle` affiche A, puis B, puis trace un triangle. Si ce fichier est placé dans un répertoire accessible, la commande `triangle` devient une commande Matlab comme toutes les autres.

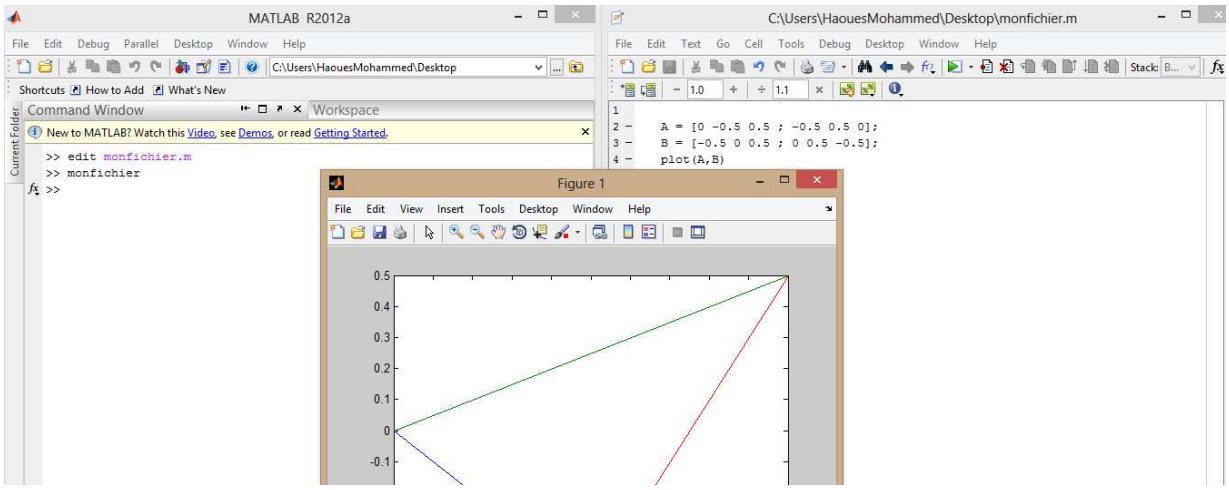


Figure 1.2. La fenêtre de commande, l'éditeur et le tracé de l'illustration 1.1.

Pour comprendre la structure et le détail du programme, des commentaires peuvent être insérés. Chaque ligne de commentaires doit être précédée par le caractère %. Les mots ou les instructions suivant ce symbole ne seront pas interprétés par le compilateur de Matlab.

Illustration 1.2

Calcul du déterminant et extraire la diagonale d'une matrice. Pour les faire, vous devez suivre les étapes suivantes :

1. Ouvrir l'éditeur de Matlab
2. Ecrire le programme suivant

```
% Cal_det_diag_matrice.
% programme calculant le déterminant
% et extraire la diagonale d'une matrice Mat
% Input :
% * Mat : matrice dont on veut connaître le deter, et la diago.
% Output:
% * deter, diago : déterminant et diagonale de Mat
deter = det(Mat);
diago = diag(Mat);
```

3. Enregistrer le programme sous le nom que vous voulez, ex: Cal_det_diag_matrice.m
4. Visualier l'aide de ce programme (help Cal_det_diag_matrice)
5. Exécuter le programme: Cal_det_diag_matrice après avoir défini la matrice Mat.

1.4. Les fichiers de fonctions

Ce sont des fichiers texte avec une extension `.m`. Une fonction est généralement utilisée pour accomplir un calcul court, utilisé par un autre programme (principal ou sous-programme). Leur syntaxe est la suivante :

```
function [Variables de sortie] = nomfonction([Variables d'entrée])
```

- Un fichier fonction commence toujours par la commande « `function` » et termine par « `end` ».

- La fonction accepte des variables d'entrée (paramètres à spécifier en argument) et de sortie (variables dans la fonction) lors de son appel dans la fenêtre de commande ou dans un autre fichier fonction.
- Les variables utilisées dans la fonction sont indépendantes (variable locales) des variables utilisées dans la fenêtre de commande ou dans un autre fichier fonction (variable globales).
- Le nom du fichier fonction doit être identique au nom de la fonction (par défaut Matlab le nome lors de l'enregistrement).

Illustration 1.3

Ecrivez dans l'éditeur la fonction qui calcule la surface d'un carré pour une côte donnée.

```
% Exemple de fichier fonction matlab
% surfacecarre calcul l'aire d'un carré de côte c
function [S] = surfacecarre (c)
% calcul Aire
S = c^2;
end
```

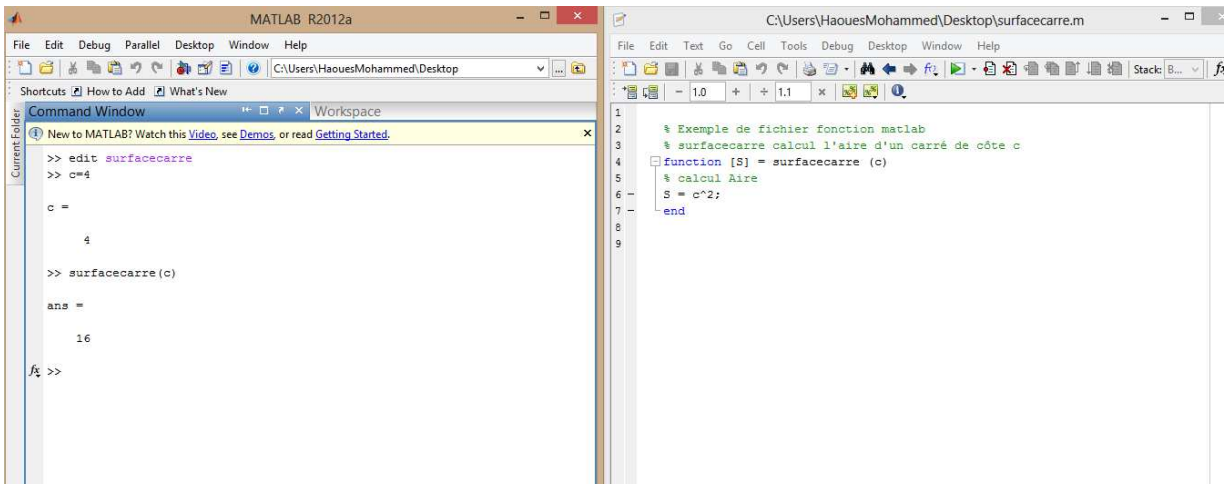


Figure 1.3. Fenêtre de l'illustration 1.3.

- Les variables locales sont des variables internes, c'est-à-dire que les variables définies dans une fonction n'existent que dans celle-ci.
- Les variables globales sont des variables du workspace. Ils ne sont pas visibles depuis une fonction.

Dans l'illustration précédente, le paramètre « S » et « c » ne sont pas connues dans le workspace.

Illustration 1.4

```
>> edit surfacecarre
>> S=surfacecarre (4)

S =

    16
```

```
>> whos
  Name      Size            Bytes  Class     Attributes

  S         1x1                 8  double
```

Illustration 1.5

Calcul de la soustraction de deux nombres.

```
function z = soustraction (x,y)
%function z = soustraction (x,y)
% calcule la soustraction de x et y
z=x-y
end
```

Pour pouvoir utiliser une variable partagée par le workspace et une (des) fonction(s), celle-ci doit être déclarée comme global à la fois dans la fenêtre de commande ou le fichier script et dans la (les) fonction(s).

Illustration 1.6

Ajoutez à la fonction :

```
global r ;
r=c ;
```

Ecrire les lignes suivantes dans la fenêtre de commande ou le fichier script.

Fenêtre de commande

```
global r ;
r=23 ;
```

Fichier script

```
>> global r ;
>> r=23 ;

S = surfacecarre (5)
>> r

r =

    5
```

La variable globale `r`, a été modifiée lors de l'appel de la fonction.

Illustration 1.7

Calcul du déterminant et extraire la diagonale d'une matrice donnée.

```
function [deter, diago] = Cal_det_diag_matrice (Mat)
       deter = det(Mat);
       diago = diag(Mat);
end
```

1. Enregistrez la fonction ci-dessus (par exemple : `Cal_det_diag_func.m`)
2. Dans l'éditeur de commande, créez un vecteur Mat dont on veut calculer le déterminant et extraire la diagonale.
Exemple : `Mat = [1 2 3 ;4 5 6 ;7 8 9];`
3. Tapez dans l'éditeur de commande : `[u,v] = Cal_det_diag_func(Mat).`

Dans ce cas, la variable u prendra le résultat du déterminant calculé dans la fonction sous le nom de variable locale `deter`. La variable v prendra le résultat de diagonale extraite sous le nom de `diago`.

4. Visualisez la valeur du paramètre `Mat` et des variables `deter` et `diago`.

Le paramètre `Mat` et les variables `deter` et `diago` sont locales. Elles n'existent pas pour l'éditeur de commande. Seules existent les variables u et v.

N.B :

Il est à noter que l'utilisation de variables globales est fortement déconseillée, car souvent source d'erreurs d'exécution, et doit donc être minimisée.

Le tableau suivant résume quelques instructions permettent de contrôler les arguments d'entrées et de sorties d'une fonction :

`nargin`	Retourne le nombre d'arguments d'entrée.
`nargout`	Retourne le nombre d'arguments de sortie.
`nargchk`	Vérifie le nombre d'arguments d'entrée.
`inputname`	Retourne le nom d'un argument d'entrée.

Tableau 1.1. Instructions de contrôle des arguments d'entrées et de sorties d'une fonction.

1.5. Les commandes structurées

1.5.1. Les boucles

Dans Matlab, il existe 2 types de boucles : la boucle for et la boucle while.

La boucle itérative for … end

Elle permet de repéter un ensemble d'instructions un nombre de fois prédéfini. Sa syntaxe est la suivante

```
for compteur = debut:increment:fin ou compteur = fin:décrement:debut
    instructions …
end
```

Le `compteur` qui est une variable est initialisé à la valeur `debut (fin)` et évolue jusqu'à la valeur `fin (début)` par pas de `increment (décrement)`. A chaque itération, le bloc `instructions` est exécuté. Généralement, le `compteur` est un scalaire, et souvent un entier.

```
for indice = 1:10
    tâche 1
    …
    tâche i
    …
    tâche 10
end
```

Dans l'exemple précédent, les tâches peuvent faire appel ou pas à l'indice (`indice`) qui vaudra 1 lors du premier passage de la boucle, `i` lors de ième passage et 10 lors du dernier passage.

Illustration 1.8

Pour calculer la somme des éléments d'un vecteur V vous pouvez calculer :

```
somme = 0; % initialisation de la variable contenant le résultat
for ind = 1:length(V) % length retourne la taille du vecteur
somme = somme+V(ind);
end
```

Illustration 1.9

Cette illustration construit élément par élément un vecteur v de dimension 5.

```
n = 5 ;
for d = 1:n
    v(d)= 1/d;
end
```

Exercice à domicile

Vous réfléchissez à un problème nécessitant une (des) boucle(s) for, mais rien ne vous empêche de chercher des problèmes avec leurs réponses dans la littérature. Tentez d'implémenter vous-même ce problème.

La boucle conditionnelle while … end

Elle permet de répéter une série d'instructions tant qu'une condition logique est vraie. Sa syntaxe est :

```
while condition logique
      instructions …
end
```

Si la condition logique est vraie, le bloc `instructions` est exécuté. Puis, la condition logique est de nouveau testée. L'exécution du bloc est répétée tant que le test est vrai.
Dans l'exemple suivant, les tâches seront effectuées tant que i est inférieur à 10 et j est supérieur à 10.

```
while i<10 & j>10
     tâche 1
     …
     tâche m
     …
     tâche 10
end
```

Illustration 1.10

Ce programme affiche 10 fois la chaîne de caractère '`toujours dans la boucle`'.

```
compteur = 0;
while compteur < 10
       disp('toujours dans la boucle') ;
       compteur = compteur + 1;
end
```

Illustration 1.11

Pour calculer la somme des éléments du vecteur V comme dans l'illustration 2.8.

```
somme = 0; % initialisation de la variable contenant le résultat
fin = 0;   % variable logique qui nous servira à arrêter la boucle
While not(fin)
somme = somme+V(ind);
       If ind < length(x)
           ind = ind+1;
       else
       fin = 1;   % pour arrêter la boucle
       end
end
```

Les instructions de rupture de séquence

`break`	Termine l'exécution de la boucle la plus proche.
`return`	Termine l'exécution d'un fichier script sans aller jusqu'à sa fin.
`error ('message')`	Affiche la chaîne de caractères message et interrompt l'exécution du programme en cours.

Tableau 1.2. Instructions de rupture de séquence.

1.5.2. Les instructions conditionnelles

Instruction de branchement conditionnel if … elseif … else

Cette structure permet d'exécuter un bloc d'instructions en fonction de la valeur logique d'une condition logique. Sa syntaxe est :

```
if condition(s)logique(s)
   instructions …
end
```

Le bloc `instructions` est exécuté seulement si la (les) `condition(s) logique(s)` est (sont) vraie(s). Plusieurs tests exclusifs peuvent être combinés.

```
if condition logique 1
       instructions1 …
elseif condition logique 2
       instructions2 …
else
       instructions3 …
end
```

Plusieurs `elseif` peuvent être concaténés. Leur bloc est exécuté si la `condition logique` correspondante est vraie et bien sûr si toutes les conditions logiques précédentes n'ont pas été satisfaites. Le bloc `instruction3` associé au `else` est quant à lui exécuté si aucune des conditions précédentes n'a été réalisées.

Illustration 1.12

```
x=0 ;  % la variable x doit être définie auparavant, i.e. calculée avant ce
       % branchement.
if x > 0
   disp('x est positif');
elseif x == 0
       disp('x est nul');
else
    x = 1;
end
% la fonction d'affichage disp permet d'afficher une chaîne de caractère spécifiée
entre apostrophes. Si x n'est ni positif ni nul, il reçoit la valeur 1.
```

Instruction de branchement multiple switch … case

Dans cette structure, une expression numérique est comparée successivement à différentes valeurs. Dès qu'il y a identité, le bloc d'instructions correspondant est exécuté. Sa syntaxe est la suivante :

```
switch expression
       case valeur 1,
       instructions 1 …
       …
       case valeur i,
       instructions i …
       …
       case valeur n,
       instructions n …
       …
       otherwise
       instructions …
end
```

L'`expression` testée, doit être un scalaire ou une chaîne de caractère. Une fois qu'un bloc d'`instructions i` est exécuté, le flux d'exécution sort de la structure et reprend après la fin de branchement `end`. Si aucune `case` vérifie l'égalité, le bloc qui suit `otherwise` est exécuté.

Illustration 1.13

```
switch x
       case 0,
            resultat = a + b;
       case 1,
            resultat = a * b;
       case 2,
            resultat = a/b;

       case 3,
            resultat = a^b;
       otherwise
            resultat = 0;
end
```

En fonction de la valeur de x une opération particulière est effectuée. Par défaut, resultat prend la valeur 0.

Chapitre 2

Utiliser le solveur fminbnd de l'`optimtool`

Dans ce chapitre, nous allons montrer à travers des illustrations comment utiliser le solveur `fminbnd` pour résoudre les fonctions unidimentionnelle dans des intervalles bornés.

2.1. Outil d'optimisation avec le solveur `fminbnd`

Cette illustration montre comment utiliser l'outil d'optimisation avec le solveur fminbnd pour minimiser une fonction à variable unique sur un intervalle fixe.

Synthaxe

```
[x,fval,exitflag,output] = fminbnd(fun,x1,x2,options)
```

`fminbnd` est un solveur unidimensionnel qui minimise un problème spécifié par :

$\min_x f^{\mathrm{T}} x$ sachant que `x1<x<x2`.

Elle retourne une valeur `x` qui est un minimum local (scalaire) de la fonction à valeur scalaire (`x1` et `x2`) décrite dans `fun` dans l'intervalle `x1<x<x2`.

Illustration 2.1

Trouvez le point où la fonction `sin (x)` prend son minimum dans l'intervalle `0 <x <2π`.

```
fun = @cos;
x1 = 0;
x2 = 2*pi;
x = fminbnd(fun,x1,x2)

x =
    3.1416
```

Pour afficher la précision, c'est la même chose que la valeur correcte `x = π`.

```
>> pi
ans =
      3.1416
```

Illustration 2.2

Trouver le minimum local et la valeur de la fonction

```
fun = @cos;
[x,fval] = fminbnd(fun,1,2*pi)

x =

    3.1416

fval =

     -1.0000
```

Illustration 2.3

Minimiser une fonction spécifiée par un fichier.m

Minimiser une fonction spécifiée par un fichier de fonction séparé. Une fonction accepte un point x comme argument d'entré et retourne un scalaire réel représentant la valeur de la fonction en x.

Écrivez la fonction suivante sous forme de fichier et enregistrez le fichier sous le nom scalarfun.m sur votre chemin MATLAB.

```
function fon = scalarfun(x)
      fon = 0;
      for i = -12:2:12
          fon = fon + (i)^4*sin(i*x)*exp(-i^2/2);
      end
end
```

Trouvez le x qui minimise la fonction scalarfun sur l'intervalle $1 \leq x \leq 10$.

```
x = fminbnd(@scalarfun,1,10)
x =

    1.2213
```

Illustration 2.4

Cette illustration à pour objectif de minimiser une fonction lorsqu'il existe un paramètre supplémentaire. La fonction `cos (x - b)` a un minimum qui dépend de la valeur du paramètre `b`.

Créez une fonction anonyme de `x` qui inclut la valeur du paramètre `b`. Minimisez cette fonction sur l'intervalle $0 < x < 2\pi$.

```
b = 1;
fun = @(x)cos(x+b);
x = fminbnd(fun,1,2*pi)

x =
    2.1416
```

Cette réponse est correcte; la valeur théorique est

```
pi - 1

ans =
      2.1416
```

Pour plus de détails sur l'inclusion des paramètres supplémentaires, vous pouvez consulter l'aide Matlab.

Illustration 2.5

Le but de cette illustration est de surveiller les itérations de fminbnd pour minimiser la fonction cos (x) pour $0 < x < 2\pi$.

```
fun = @cos;
x1 = 0;
x2 = 2*pi;
options = optimset('Display','iter');
x = fminbnd(fun,x1,x2,options)
```

```
  Func-count     x          f(x)          Procedure
    1        2.39996    -0.737369        initial
    2        3.88322    -0.737369        golden
    3        4.79993    0.0874257        golden
    4        3.14159           -1        parabolic
    5        3.14163           -1        parabolic
    6        3.14156           -1        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04

x =

    3.1416
```

Illustration 2.6

Affichez toutes les informations sur le processus de résolution `fminbnd` en demandant toutes les sorties. Surveillez également le processus à l'aide d'une fonction de tracé.

```
fun = @cos;
x1 = 0;
x2 = 2*pi;
options = optimset('PlotFcns',@optimplotfval);
[x,fval,exitflag,output] = fminbnd(fun,x1,x2,options)
```

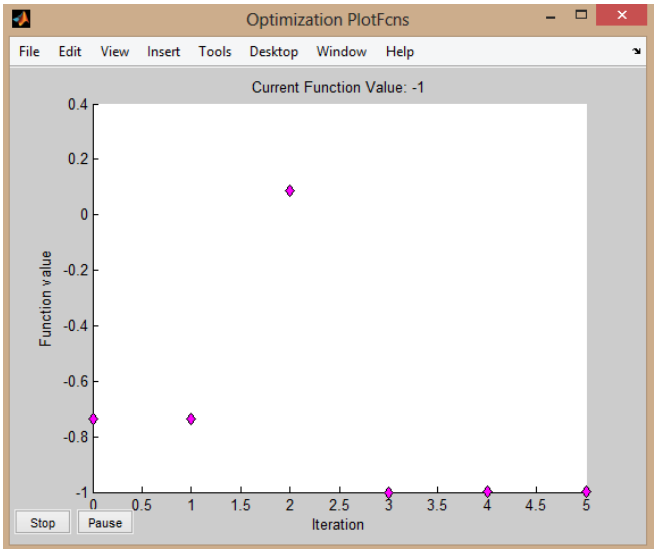


Figure 2.1. Surveillance du processus de résolution du solveur `fminbnd`.

```
x =

    3.1416
fval =

    -1
```

```
exitflag =

     1

output =
    iterations: 5
     funcCount: 6
     algorithm: 'golden section search, parabolic interpolation'
       message: [1x111 char]
```

Arguments d'entrée		
`fun`	La fonction à minimiser. Types de donnés : char | fichier_fonction | string	
`x1`	Borne inférieure. Types de donnés : double	
`X2`	Borne supérieure. Types de donnés : double	
`options`	Options de l'optimisation : une structure retournée par `optimset`.Types de donnés : `struct`	
	`Display`	Niveau d'affichage: `'notify'` (par défaut) affiche la sortie uniquement si la fonction ne converge pas. `'off'` or `'none'` n'affiche aucune sortie. `'iter'` affiche la sortie à chaque itération. `'final'` affiche uniquement la sortie finale.
	`FunValCheck`	Vérifiez si les valeurs de la fonction objectif sont valides. ○ La valeur par défaut «off» permet à fminbnd de continuer lorsque la fonction objectif reourne une valeur complexe ou NaN. ○ Le paramètre «on» génère une erreur lorsque la fonction objectif retourne une valeur complexe ou NaN.
	`MaxFunEvals`	Nombre maximum d'évaluations de fonctions autorisé, un entier positif. La valeur par défaut est 500.
	`MaxIter`	Nombre maximal d'itérations autorisé, un entier positif. La valeur par défaut est 500.
	`OutputFcn`	Spécifiez une ou plusieurs fonctions définies par l'utilisateur qu'une fonction d'optimisation appelle à chaque itération, en tant que descripteur de fonction ou en tant que tableau de cellules de descripteurs de fonction. La valeur par défaut est none ([]).
	`PlotFcns`	Trace diverses mesures de progrès pendant l'exécution de l'algorithme, sélectionnez des tracés prédéfinis ou écrivez le vôtre. Passez un descripteur de fonction ou un tableau de cellules de descripteurs de fonction. La valeur par défaut est none ([]). ○ `@optimplotx` trace le point actuel. ○ `@optimplotfunccount` trace le nombre de fonctions. ○ `@optimplotfval` trace la valeur de la fonction.
	`TolX`	Tolérance de la condition d'arrêt sur `x`, un scalaire positif. La valeur par défaut est 1e-4.

Tableau 2.1. Arguments d'entrée – `fminbnd` –

Arguments de sortie		
x	Solution, retournée comme un réel scalaire. En règle générale, x est une solution locale au problème lorsque exitflag est positif.	
fval	Valeur de la fonction objectif à la solution, retournée sous forme de nombre réel.	
exitflag	Raison pour laquelle fminbnd s'est arrêté, retournée sous forme d'entier.	
	1	La fonction a convergé vers une solution x.
	0	Nombre d'itérations dépassé options.MaxIter ou nombre d'évaluations de fonctions dépassé options.MaxFunEvals.
	-1	Arrêté par une fonction de sortie ou une fonction de tracé.
	-2	Les limites sont incohérentes, ce qui signifie x1> x2.
output	Informations sur le processus d'optimisation, retournées sous forme de structure avec des champs:	
	iterations	Nombre d'itérations atteint.
	funcCount	Nombre d'évaluations de fonctions.
	algorithm	'golden section search, parabolic interpolation'
	message	Message de sortie.

Tableau 2.2. Arguments de sortie – fminbnd –

2.2. Limites du solveur fminbnd

- La fonction à minimiser doit être continue.
- fminbnd pourrait ne donner que des solutions locales.
- fminbnd peut présenter une convergence lente lorsque la solution est sur une frontière de l'intervalle.

2.3. Les algorithmes

fminbnd est un fichier de fonction. L'algorithme est basé sur la recherche de la section d'or et l'interpolation parabolique ('golden section search, parabolic interpolation'). À moins que le point d'extrémité gauche x1 soit très proche du point d'extrémité droit x2, fminbnd n'évalue jamais les fonctions aux points d'extrémité, donc la fonction n'a besoin d'être défini que pour x dans l'intervalle x1 <x <x2.
Si le minimum se produit réellement à x1 ou x2, fminbnd renvoie un point x à l'intérieur de l'intervalle (x1, x2) qui est proche du minimum. Dans ce cas, la distance de x par rapport au minimum n'est pas supérieure à 2*(TolX + 3*abs(x)*sqrt(eps)).

Illustration 2.6

Cette illustration montre comment utiliser l'interface graphique de l'outil d'optimisation avec le solveur fminbnd pour minimiser une fonction continue a une seule variable avec des bornes.

Considérez le problème d'optimisation de l'illustration 7.1

```
fun = @cos;
x1 = 0;
x2 = 2*pi;
x = fminbnd(fun,x1,x2)
```

Configurez et exécutez le problème à l'aide de l'outil d'optimisation.

- Saisir optimtool dans « **Command Window** » (la fenêtre de commande) pour ouvrir l'outil d'optimisation (**Optimization Tool**).

- Sélectionnez « **fminbnd** » dans la liste des solveurs. Le champ Algorithm n'existe pas, parcequ'un seul algorithme est implimenté `'golden section search, parabolic interpolation'`, c'est un algorithme de résolution des fonctions sans dérivation.

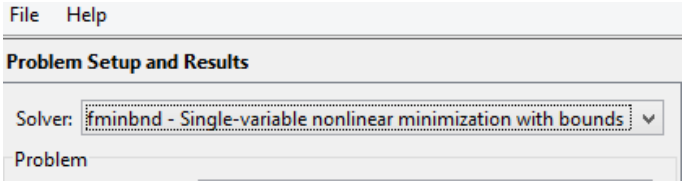


Figure 2.2. Choix du solveur `fminbnd`.

Définissez les contraintes.

- Définissez les limites de la variable `x`, faire entrez `0` dans le champ x1 pour la limite Inférieure et `2*pi` dans le champ x2 pour la borne supérieure.

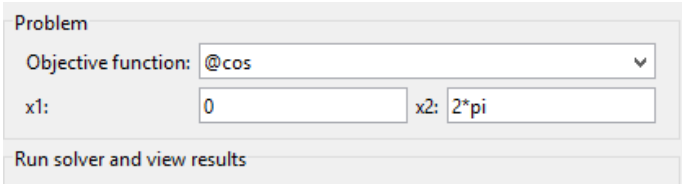


Figure 2.3. Saisie des données du solveur `fminbnd`.

- Dans le volet Options, développez l'option « **Display to command window** » (Afficher dans la fenêtre de commande) si nécessaire, et sélectionnez **Iterative** (itératif) pour afficher les informations d'algorithme à « **Command Window** » (la Fenêtre de commande) pour chaque itération.

Figure 2.4. Choix du niveau d'affiche du solveur `fminbnd`.

- Vous pouvez aussi spécifiez des conditions d'arrêts dans l'option « **Stopping criteria** ». Comme vous pouvez les laissez par défaut.
-

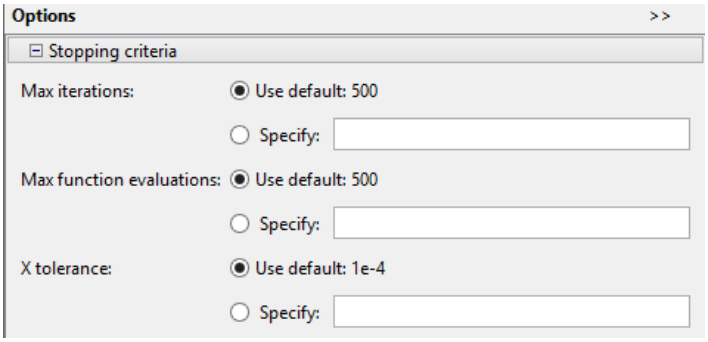


Figure 2.5. Critères d'arrêt du solveur `fminbnd`.

- Cliquez sur le bouton « **Start** » (Démarrer), comme illustré dans la figure 7.6.

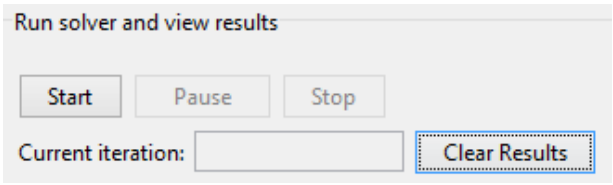


Figure 2.6. Lancement du solveur `fminbnd`.

- Lorsque l'algorithme se termine, sous « **Run solver and view results** », les informations suivantes s'affichent:

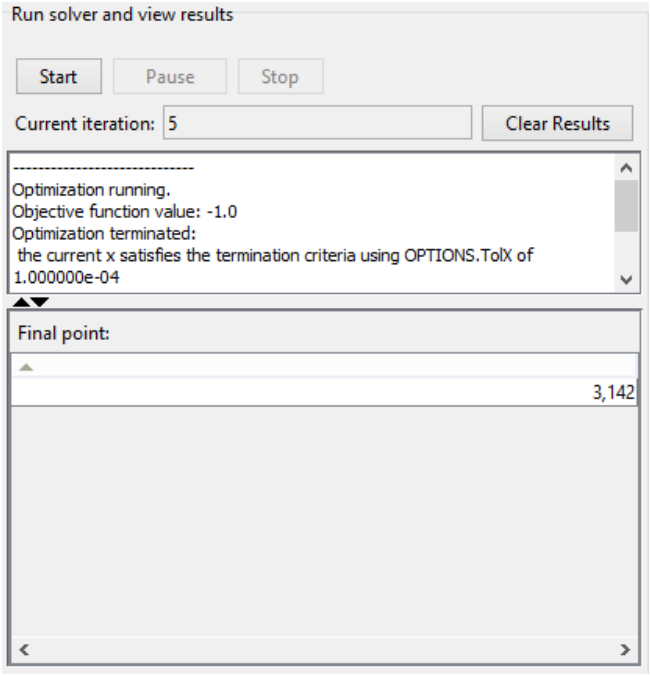
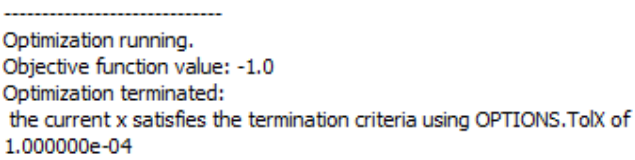


Figure 2.7. Afichage des résultats du solveur `fminbnd`.

- La valeur « **Current iteration** » d'itération actuelle à la fin de l'algorithme, qui est 5 pour cette illustration.
- La valeur finale de la fonction objectif à la fin de l'algorithme est :
  Valeur de la fonction objectif: -1.0
- Le message de fin d'algorithme est :



- Le point final, qui est pour cette illustration :

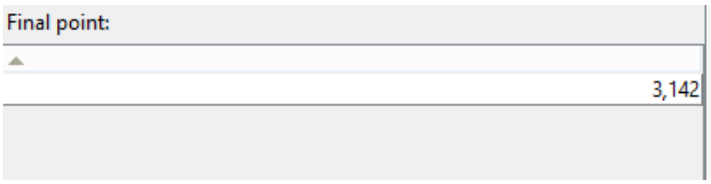


Figure 2.8. Point final du solveur `fminbnd`.

- Dans « **Command Window** » (la fenêtre de commande), les informations de l'algorithme sont affichées pour chaque itération :

```
 Func-count     x          f(x)          Procedure
    1        2.39996    -0.737369        initial
    2        3.88322    -0.737369        golden
    3        4.79993    0.0874257        golden
    4        3.14159           -1        parabolic
    5        3.14163           -1        parabolic
    6        3.14156           -1        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04
```

Pour surveiller également le processus à l'aide d'une fonction de tracé. Vous cochez une option :

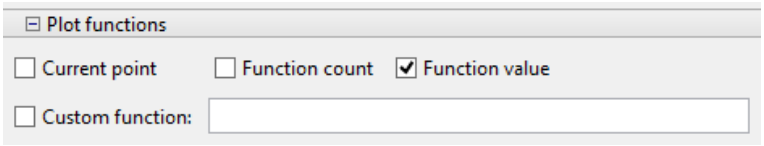


Figure 2.9. Choix de la fonction de tracé du solveur `fminbnd`.

La fenêtre suivante s'affiche :

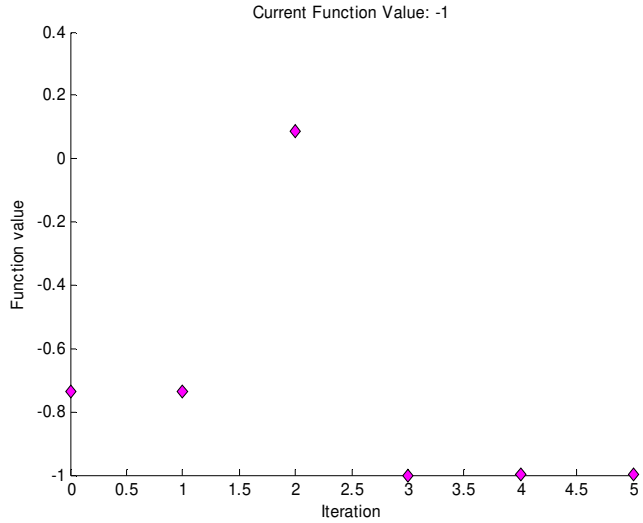


Figure 2.10. Surveillance du processus de résolution du solveur `fminbnd`.

Chapitre 3

Introduction au Logiciel GAMS (General Algebraic Modeling System)

Dans ce chapitre, nous allons montrer à travers des illustrations comment utiliser le logiciel GAMS pour résoudre des mdèles mathématiques.

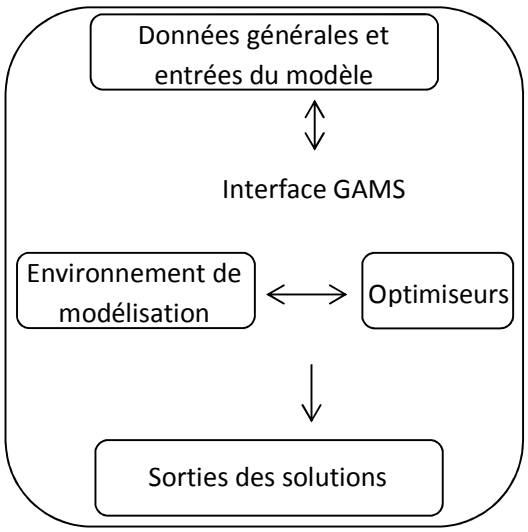
L'objectif de ce cours est de donner un rapide aperçu du logiciel GAMS. Face au développement des capacités de calcul et des algorithmes d'optimisation, ce logiciel a été développé afin de répondre à un certain nombre de besoins :

- Fournir un langage structuré pour la description synthétique de modèles complexes de grande taille.
- Permettre la définition du modèle de façon indépendante des algorithmes de résolution utilisés (logiciel modélisateur).
- Permettre la modification du modèle de façon simple et sécurisée.
- Permettre la définition univoque de relations algébriques.

Deux grands principes ont guidé les concepteurs de ce logiciel :

- Tous les algorithmes doivent être utilisables sans aucune modification du modèle.
- La formulation du modèle doit être indépendante des données utilisées par ce dernier (en particulier, l'augmentation de la taille du modèle, n'affecte pas la complexité de la représentation de ce dernier).

Comme montre la figure 1. L'environnement de modélisation et la bibliothèque d'algorithmes d'optimisation ont été conçus indépendamment.



- Le logiciel GAMS (General Algebraic Modeling System) a été à l'origine développé par un groupe d'économistes de la Banque Mondiale dans le but de faciliter la résolution des modèles complexes.

- GAMS permet de résoudre des modèles non-linéaires avec ou sans fonction objectif à optimiser.

- Il est caractérisé par la simplicité dans l'implémentation (logiciel modélisateur et pas solveur comme par exemple Matlab).

- Un des grands avantages du logiciel GAMS, la facilité des mises à jour grâce à de nouveaux algorithmes régulièrement.

- GAMS est un exécutable. L'écriture du modèle se fait dans un fichier d'entrée avec l'extension «.gms » et l'enregistrement des résultats dans un fichier sortie avec l'extension «.lst ». Le modèle porte le même nom en fichier d'entrée et de sortie.

Définition du problème (structure d'un modèle GAMS)

On spécifie un modèle GAMS en donnant à la fois sa structure et des données proprement dites dans un fichier séquentiel. Un modèle GAMS peut se décomposer en trois modules, chaque module se décompose d'un ensemble de sections :

Module d'entrée des données (étape 1) : il comporte :

- déclaration et définition des ensembles : **SET**
- déclaration, définition et assignation des données : **PARAMETERS, SCALARS, TABLES**
- présentation des données intermédiaires, comme le calcul de certains paramètres ou variables **DISPLAY**

Module de spécification du modèle (étape 2) : il rassemble :

- La déclaration des **VARIABLES**
- La définition et la déclaration des équations (l'écriture des **EQUATIONS**)
- La définition du **MODEL** (en fonction des équations choisies)

Module de procédure de résolution (étape 3) : il regroupe :

- La commande de résolution du modèle **SOLVE**
- La présentation des résultats **DISPLAY**

Dans un programme GAMS la structure suivante doit être respectée.

1) Les ensembles : **SET**
2) Les données : **PARAMETERS, SCALARS, TABLES**
3) La déclaration des **VARIABLES**
4) L'écriture des **EQUATIONS**
5) Définition du **MODEL** (en fonction des équations choisies)
6) Commande de résolution du modèle **SOLVE**
7) Calcul de certains paramètres ou variables **DISPLAY**

Les commentaires

Les commentaires dans GAMS peuvent facilement être introduits dans le modèle :

- Soit en commençant la ligne par un astérisque * si on veut mettre en commentaire une ligne.
- Soit en utilisant la syntaxe :

$ONTEXT
Texte
$OFFTEXT

Pour mettre en commentaire une section.

- Soit au sein des énoncés de GAMS

Titre

Pour en donner un, on utilise la commande **$TITLE** suivi du nom du modèle

Ponctuation

- Chaque instruction doit être terminée par un point-virgule « **;** ».

- Les affectations doivent être suivies par un point-virgule « **;** ».
- La virgule « **,** » ou le « **retour** » séparent les éléments d'une liste.
- Le « / » indique le début ou la fin d'une liste d'éléments d'un ensemble.
- ATTENTION : GAMS ne comprend pas les accents.
- GAMS ne distingue pas les majuscules des minuscules.

Règles importantes

Le modèle doit être décrit en utilisant l'environnement (éditeur) de GAMS. Ce dernier créera un fichier ayant le suffixe .gms. Par exemple : ProductionPlan.gms

Les règles suivantes doivent être respectées :

- Un élément (indice, paramètre, équation …) ne peut être référencé (affecté ou utilisé) avant d'avoir été déclaré.
- Une ligne ou une section commentaire ne sera pas interprétée par le compilateur.
- Le nom d'un élément comporte au maximum 9 caractères alphanumériques. Le premier est toujours alphabétique.
- Il existe un certain nombre de mots réservés. Ils ne doivent pas être utilisés en dehors du contexte prévu par le GAMS. A titre d'exemple le mot sets, est un mot réservé à la déclaration des indices. Il ne doit donc pas être utilisé comme identificateur d'une variable.
- Les mots réservés sont repérés en **gras** et en **bleu** par l'éditeur de texte.
- GAMS ne distingue pas les lettres minuscules des lettres majuscules.
- GAMS permets de mettre sur la même ligne une ou plusieurs commande(s). cela peut réduire la longuer du code ou faciliter l'impression.

1) LES ENSEMBLES : SET

Cette commande permet de définir les ensembles sur lesquels sont définies les variables et les paramètres

Exemple :

Pour déclarer les secteurs de l'économie d'un état

```
SET i secteurs de l'économie d'un état
      / agr secteur agricole
        ind secteur industriel
        ser secteur des services
        tel secteur des télécommunications /
;
```

Ceci équivalant à i={agr, ind, ser, tel}

Autres mots clés importants :

1-1) Sous-ensemble :

Parfois on doit définir un ensemble dont les éléments sont ceux d'un ensemble plus grand.

Exemple:

SET r(i) secteurs de l'économie hors services /agr, ind/ ;

1-2) ALIAS :

Il permet de donner un autre nom à un ensemble qui a été défini précédemment.

Exemple :

ALIAS (i,j) ;
ALIAS (m,n);

2) LES DONEES et l'étalonnage des paramètres

La commande utilisée pour introduire les données est **PARAMETER ou PARAMETERS** (qu'ils s'agissent de paramètres, de variables endogènes ou exogènes sur le plan théorique).

On commence par déclarer les paramètres puis ensuite les variables pour l'année de référence.

Exemple :

Alpha Part de la rémunération du capital dans le produit
Beta Part de la rémunération du travail dans le produit
Y0 Revenu des ménages
;

- Les données peuvent être introduites sous formes de scalaire (lorsqu'un ensemble n'est pas associé). La commande est alors **SCALAR**

Exemple :

SCALAR
Y0 /200/ ;

- Les données peuvent être introduites sous forme de tableaux. La commande est alors **TABLE**

Exemple :

TABLE MCS (m,n)
agr ind men gov
agr 100 150 150 0
ind 100 350 125 25
men 100 400 0 50
gov 0 0 25 0
;

Où MCS désigne le nom du tableau et (m,n) son domaine avec m les lignes et n les colonnes. Si le tableau n'est pas carré, on utilise TABLE DP (*,i)

agr ind
P0 1 1
XS0 256 451
;

- Enfin, il est possible de faire une « assignation directe » : étalonnage des paramètres.

Exemple :

Pms = s/Y0 ;

Alpha =R*K0/p0*Y0 ;

3) Les VARIABLES

Tous les variables du modèle en GAMS doivent être déclarés avant d'apparaître dans une équation.

Exemple :

XS(i) offre du secteur i

4) Les EQUATIONS

- Les équations doivent d'abord être déclarées avant d'être définies.
- Le nom de l'équation doit être différent du nom de la variable qu'elle cherche à expliquer.
- La définition de l'équation suppose:

1. le nom de l'équation ;
2. l'ensemble sur lequel elle est définie ;
3. les deux points « .. » pour annoncer l'équation ;
4. expression du côté gauche ;
5. l'opérateur =e= ;
6. expression du côté droit ;
7. le point-virgule.

Exemple :

EQXS(i).. XS(i)=e=ld(i)**a(i)*k(i)**(1-a(i));

Après la définition des variables et des équations, on donne les conditions initiales au modèle.
Pour les variables endogènes, on utilise .L (pour level) et pour les variables exogènes, on utilise .FX (pour fixed)

5) Le modèle et son mode de résolution

Il faut dire à GAMS quel est le modèle (c'est à dire le groupe d'équations) qu'il doit résoudre et comment il doit le résoudre.

MODEL nom /all/ ;
SOLVE nom **USING NLP maximizing** utility ; (Résous en utilisant la programmation non linéaire –NLP pour Non Linear Programming –et en maximisant l'utilité des agents)