

TP N° 5 : Les fonctions

Durant les séances précédentes nous avons utilisé certaines fonctions comme : sum, prod, ones, zeros, etc. Dans ce TP nous allons voir comment définir des nouvelles fonctions qui ne figurent pas parmi les fonctions incorporées de MATLAB et les utiliser de la même manière (ces fonctions sont nommées les fonctions utilisateur).

Activité 1 : Implémentation et utilisation d'une fonction

Ces fonctions ne sont en réalité que des scripts (fichier texte avec l'extension .m) avec des paramètres d'entrées et/ou des paramètres de sorties et qui commence obligatoirement avec un en-tête de la forme

```
function [ output_args ] = Untitled( input_args )  
%UNTITLED Summary of this function goes here  
% Detailed explanation goes here  
  
    Ici le corps de la fonction  
  
end
```

Avec :

Untitled : le nom de la fonction,

input_args : liste des paramètres d'entrée (séparés par des virgules). Si n'y a pas de paramètres d'entrée, on peut mettre des paramètres vides ou bien on ne met pas les parenthèses.

output_args : liste des paramètres des sorties (séparés par des espaces ou bien des virgules). Si n'y a pas de paramètres de sortie ou un seul paramètre de sortie, on ne met pas les crochets.

NB :

1. Les commentaires (texte en ver) ne sont pas des simples commentaires, c'est le help de la fonction (voir après).
2. Le « end » n'est pas nécessaire. Il sera nécessaire à la présence des sous fonction (voir après).
3. Comme le cas des scripts, vous pouvez écrire votre fonction en utilisant un simple éditeur de texte comme le bloc note et des enregistrer avec l'extension .m
4. L'éditeur de Matlab (cliquer sur le bouton « New » → « Function ») peut vous assister à écrire le code et vous présenter des suggestion convenables.

Exemple 1

Démarrer l'éditeur de script de Matlab et écrive les lignes de codes suivantes :

```
function [res] = produit(a,b)  
    res = a*b;  
end
```

Pour tester cette fonction, vous allez écrire dans la ligne de commande :

```
>> e = produit(2,4);
```

Une fois cette commande exécutée, on récupère dans la variable e la valeur de sortie de la fonction, c'est à dire le produit des deux paramètres en entrée.

NB :

- Avant de tester la fonction vous devez s'assurer que le fichier produit.m est placé dans Current folder.

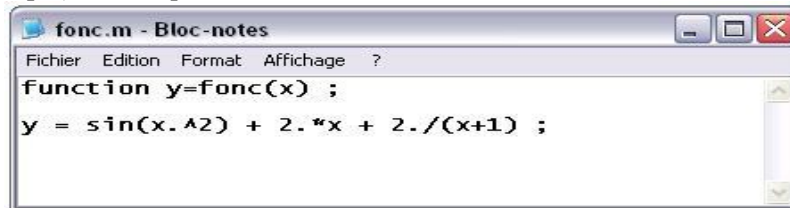
- N'oublier pas d'introduire les arguments lors de l'appel à la fonction (dans l'exemple précédant nous avons écrit « produit(2,4) », par contre « produit » seulement va générer une erreur).

Exemple 2

Soit la fonction f définie par la formule suivante :

$$f(x) = \sin(x^2) + 2x + \frac{2}{x+1}$$

L'objectif de cette exercice est d'implémenter la fonction f sur Matlab. Pour cela, créer un document texte (bloc notes par exemple) dans lequel écrire le texte ci-dessous :



Enregistrer ce document sous le nom `fonc.m` et placer le dans votre currentfolder. Utiliser la fonction `fonc` pour calculer $f(0)$, $f(1)$, $f(2)$ et $f(3)$.

Remarques

1. Généralement, il faut mettre un point-virgule devant chaque instruction de la fonction (lors de l'appel de la fonction on s'intéresse pas au détail de calcul).
2. Vous pouvez utiliser l'éditeur de script de Matlab au lieu de l'éditeur de texte Bloc Note. Pour cela, cliquer sur fichier puis nouveau script.
3. Le choix : nouveau fonction vous permet de créer un nouveau script et d'écrire automatiquement la syntaxe de la fonction (ie, le mot clé `function` ...)
4. La syntaxe d'une fonction Matlab ne vous permet d'écrire aucun caractère (même l'espace ou le retour à la ligne) avant le mot clé `function`.
5. Les commentaires situés entre la ligne `function` et la première instruction dans le corps de la fonction représente le help (voir activité 3).

Exercices d'application

1. Ecrire les fonctions : `diametre(r)`, `criconference(r)` et `surface(r)` qui reçoivent comme argument le rayon d'un cercle et renvoie respectivement le diamètre, la criconference et la surface
2. Ecrire la fonction `diametreCriconferenceSurface(r)` qui reçoit comme argument le rayon d'un cercle et renvoie à la fois le diamètre, la criconference et la surface

Solution

`diametre(r)`

```
function d = diametre(r)
    d=r*2;
end
```

`criconference(r)`

```
function c = criconference(r)
    c=r*2*pi;
end
```

`surface(r)`

```
function s = surface(r)
    s=r^2*pi;
end
```

`diametreCriconferenceSurface(r)`

```
function [d c s] = diametreCriconferenceSurface1(r)
    d=r*2;
    c=r*2*pi;
```

```
s=r^2*pi;  
end
```

ou bien :

```
function [d c s] = diametreCriconferenceSurface2(r)  
    d= diametre(r);  
    c= criconference(r);  
    s= surface(r);  
end
```

Dans cette deuxième solution, la fonction `diametreCriconferenceSurface2()` fait appel aux fonctions utilisateur `diametre()`, `criconference()` et `surface()`, ces fonctions doivent être déjà implémentées et enregistrées dans le current folder, sinon le compilateur va générer une erreur lors de l'appel de la fonction `diametreCriconferenceSurface2()`.

Activité 2 : variables locales

Pour bien comprendre cette partie essayer tout d'abord de supprimer toutes les variables de vous avez sur le workspace (en utilisant la commande `clear all`). En suite déclarer une variable `x = 4` et `y = 5`

```
>> x=4 ; y=5 ;
```

Nous allons maintenant exécuter la fonction `produit()` que nous avons implémenter dans l'activité précédente pour calculer le produit de `x` et `y` :

```
>> z=produit(x,y) ;
```

Une fois la commande `produit(x,y)` est exécutée, on récupère bien dans `z` la valeur de sortie de la fonction, c'est à dire le produit des deux paramètres en entrée `x` et `y`. On remarquera que l'on ne voit pas, dans la zone des variables de Matlab `workspace`, une variable nommée `res`. En effet, la variable `res` est interne à la fonction `produit`, et ne vit que pendant l'appel de cette fonction. Une fois la fonction terminée, la variable sera automatiquement détruite. De plus, la variable `res` n'existe que pour la fonction `produit` : elle ne peut pas être lue par une autre fonction ou par une commande directement dans Matlab.

D'autre part si vous exécuter les commandes suivantes :

```
>> a=2 ;  
>> b=4 ;  
>> z = produit();
```

Vous allez trouver que la dernière commande ne marche pas et elle va générer un message d'erreur

```
Error using produit (line 3)  
Not enough input arguments.
```

Cette erreur est dû au fait que la fonction n'a pas trouvé les arguments `a` et `b` même s'il existe des variables `a` et `b` parmi les variables de workspace la fonction ne va pas les considérer en tant que des arguments ou bien des variables internes. Par suite, il faut bien distinguer les variables globales qu'on trouve dans le workspace et qui sont visible dans tout le programme et les variables local d'une fonction qu'ont trouvent déclarer à l'intérieur d'une fonction et dans la durée de vie est le temps d'exécution de la fonction et les arguments de la fonction qu'on doit fournir entre parenthèses lors de l'appel de la fonction et qui sont le seul moyen qui permet de transmettre une ou plusieurs valeurs du workspace à la fonction.

Enfin, considérez cette fonction qui ne prend aucun paramètre de sortie :

```
function incrementer(a)  
    a=a+1 ;  
end
```

Cette fonction ajoute simplement 1 à son entrée. Testez-la dans Matlab :

```
>> x  
x =  
    5  
>> incrementer(x)  
>> x
```

```
x =  
5  
>> x=incrémenter(x)  
Error using incrémenter  
Too many output arguments.
```

La fonction s'exécute, mais la variable x n'est pas incrémentée de 1 ; en effet, Matlab a transmis le paramètre d'entrée à la fonction par suite la valeur de x qui est recopiée dans a , ensuite la fonction a incrémenté la valeur de a , mais les modifications réalisées dans a seront perdues lorsque la fonction termine son exécution parce que la variable a n'est pas déclaré comme une valeur de retour de la fonction donc c'est une simple variable locale qui sera détruite à la fin et la fonction ne va transmettre aucune valeur après son exécution.

La dernière instruction génère une erreur parce que la fonction n'a pas de valeur de retour.

Activité 3 : Documentation d'une fonction

Nous avons souvent besoin de consulter le help pour certaines fonctions afin de savoir comment les utiliser. Jusqu'à maintenant les fonctions utilisateurs que nous avons implémenter ne possède aucun help ; en effet voici le message que nous allons obtenir si on consulte le help d'une telle fonction :

```
>> help produit  
No help found for produit.m.
```

Documenter une fonction c'est-à-dire ajouter un help pour cette fonction. Pour documenter une fonction on doit ajouter des commentaires entre la première ligne (qui commence par le mot clé function) e la première instruction du corps de la fonction

Exemple :

Pour documenter la fonction `produit()`, on ajoute les deux lignes de commentaires ci-dessous puis on enregistre.

```
function [res] = produit(a,b)  
% c'est une fonction qui reçoit comme argument deux nombres réels et  
% calcul leur produit  
  
res = a*b;  
end
```

Pour afficher le help de la fonction `produit()`, on tape `help produit` dans la ligne de commande de command window.

```
>> help produit  
c'est une fonction qui reçoit comme argument deux nombres réels et  
calcul leur produit  
  
>>
```

Exercices

Exercice1

Documenter les fonctions que nous avons implémenté dans l'activité 1.

Exercice 2

Implémenter une fonction `min3` qui renvoi le minimum de trois nombres entrés en argument

Solution :

```
function m = min3(a,b,c)  
if a>b  
    if b>c  
        m=c;  
    else  
        m=b;
```

```

        end
    else
        if a>c
            m=c;
        else
            m=a;
        end
    end
end

```

NB : n'oublier d'ajouter le help et de tester la fonction

Exercice 3

Implémenter une fonction min3 qui renvoi le maximum de trois nombres entrés en argument

Solution : A faire

Exercice 4

Implémenter une fonction fact qui renvoi le factoriel d'un nombre entré en argument

Solution

```

function fac = fact(n)
fac= prod(1:n) ;
end

```

Une autre solution

```

function fac = fact(n)
if n==1
    fac=1 ;
else
    fac= n*fact(n-1) ;
end

```

NB : on suppose ici que l'utilisateur ne va entrer comme argument de la fonction que des nombre positive. Nous allons voir après comment ajouter des filtres qui vont tester l'argument avant de commencer le traitement.

Exercice 5

Implémenter la fonction Eq1d(a,b) qui renvoi la solution de l'équation : $ax+b=0$

Solution

```

function x= Eq1d(a,b)
    if a==0
        if b==0
            x='R';
        else
            x=[] ;
        end
    else
        x=-b/a;
    end
end

```

Exercice 6

Implémenter la fonction Eq2d(a,b,c) qui renvoi la solution **réelles** de l'équation : $ax^2+bx+c=0$

Solution

```

function y = Eq2d (a ,b,c)
if a==0
    y = Eq1d(b,c) ;
else
    delta = b^2 - 4*a*c ;
    if delta <0
        y=[] ; % pas de solution
    else
        if delta == 0
            y= -b/(2*a) ;
        else
            y= [-b/(2*a) ± sqrt(delta)/(2*a)] ;
        end
    end
end

```

```

        else
            x1 = (-b-sqrt(delta))/(2*a);
            x2 = (-b+sqrt(delta))/(2*a);
            y = [x1 x2] ;
        end
    end
end

```

Exercice 7

Écrire une fonction `binome(a, b, c)` qui renvoie les racines du polynôme $aX^2 + bX + c$.
(NB : N'utiliser pas la fonction `roots`).

Indications :

Suivez l'algorithme habituel. Calculez $\Delta = b^2 - 4ac$ puis en fonction du signe de Δ renvoyez les racines réelles ou complexes ;

Vous pouvez utiliser la variable `i` en Matlab pour désigner la racine carrée de -1 , et la fonction `sqrt(x)` pour obtenir la racine carrée d'un nombre positif ;

Pour renvoyer plusieurs valeurs, mettez-les dans un tableau et renvoyez le tableau.

Solution :

```

function r=binome(a,b,c)
delta=b*b-4*a*c;
if delta >= 0
    r=[(-b+sqrt(delta))/(2*a), (-b-sqrt(delta))/(2*a)];
else
    r=[(-b+i*sqrt(-delta))/(2*a), (-b-i*sqrt(-delta))/(2*a)];
end

```

Exercice 8

Écrire une fonction qui, étant donné un polynôme `P`, renvoie son degré.

Solution :

```

function r=poly_deg(P)
n=size(P,2);
r=0;
for i=1:n
    if P(i)~=0
        r=i-1;
    end
end

```

Exercice 9

Écrivez une fonction qui, étant donné un polynôme `P` et un entier `k`, renvoie le coefficient de X^k dans `P`

Solution :

```

function r=poly_coeff(P,k)
n=size(P,2);
if k>n
    r=0;
else
    r=P(k);
end

```

Exercice 10

Écrire une fonction minimum qui, étant donné un tableau en entrée, renvoie son plus petit élément, puis une fonction maximum qui renvoie le plus grand élément d'un tableau.

Solution :

```
function y = minimum(tab)
y = tab(1);
for i = 2 : size(tab,2)
if tab(i) < y
    y = tab(i);
end
end
```

1. Écrire la fonction minmax qui renvoie le plus petit et le plus grand élément d'un tableau (indication : faire appel aux deux fonctions que vous avez implémentées dans la question 1)

Exercice14

1. Implémenter la fonction suivante :

$$f(x) = \begin{cases} \sin(2x + 3) & \text{si } x < 0 \\ 0 & \text{si } x = 0 \\ \cos(2x + 3) & \text{si } x > 0 \end{cases}$$

2. Pour tester la fonction calculer f(0), f(1), f(2) et f(3).

Activité 4 : nargin, nargout, varargin et varargout

Supposons que nous voulons implémenter une fonction som() qui permet de calculer la somme d'un nombre inconnu de nombres entrés. La syntaxe :

```
function s=som(x1,x2)
    s=x1+x2 ;
end
```

vous permet de calculer la somme de deux arguments ; la fonction som (selon cette syntaxe) n'accepte pas d'être appliquée pour trois variables : som(14,15,10) génère une erreur !!!

Matlab vous propose une solution où le nombre d'entrées est dynamique :

```
function s=som(x,varargin)
    s=x;
    for i = 1:length(varargin)
        s=s+varargin{i};
    end
end
```

on peut appliquer cette fonction pour 1, 2, 3 ou plus de variables

```
>> som(4)
ans =
     4
>> som(4,2)
ans =
     6
>> som(4,2,5)
ans =
    11
```

Dans le code de la fonction `som()`, nous avons utilisé comme argument `x` et `varargin` ; `x` est un argument statique par contre `varargin` est un mot clé de Matlab qui représente une liste dynamique de variable. La variable `x` est obligatoire par contre la variable `varargin` est optionnelle parce que elle peut être une liste vide, l'absence de `x` génère une erreur.

```
>> som()
Error using som (line 2)
Not enough input arguments.
```

Le code suivant implémente une fonction `som()` qui peut être appliquée sans aucun argument et renvoie 0

```
function s=som(varargin)
    s=0;
    for i = 1:length(varargin)
        s=s+varargin{i};
    end
end
```

L'exécution de cette fonction

```
>> som()
ans =
     0
>> som(2)
ans =
     2
```

Un autre exemple

```
function [res] = produit(a,b, varargin)
    res = a*b;
    for i=1:length(varargin)
        res = res*varargin{i};
    end
end
```

Dans cet exemple, `a` et `b` sont des arguments obligatoires et `varargin` est une liste dynamique d'argument qui peut être vide

De la même façon il existe une variable nommée `varargout` qui va jouer le rôle d'une liste dynamique de variable de sortie. Afin de bien comprendre l'utilisation de cette variable nous allons traiter l'exemple suivant :

```
function [varargout] = demander()
    for i=1:nargout
        varargout{i} = input('Entrez une valeur : ');
    end
end
```

la fonction `demander()` permet de demander à l'utilisateur des valeurs pour remplir les variables de sortie. Le nombre des valeurs demandées n'est pas connu lors de l'implémentation de la fonction.

Pour l'exécution de cette fonction :

```
>>[x y]=demander()      %lors de l'exécution l'utilisateur veut demander deux valeurs
Entrez une valeur : 10
Entrez une valeur : 11
x =
    10
y =
    11
```

```
>>[x y z]=demander()    %lors de l'exécution l'utilisateur veut demander trois valeurs
Entrez une valeur : 10
Entrez une valeur : 11
Entrez une valeur : 14
x =
    10
y =
    11
z =
    14
```

La variable `nargout` dans le code de la fonction `demander()` indique le nombre de valeurs de sortie.

Un autre exemple pour `nargout`, la fonction `myIntegerDivision()` qui reçoit deux entiers `a` et `b` et calcul le quotient et le reste de la division entière de a/b :

```
function [qt, rm] = myIntegerDivision(a, b)
    qt = floor(a / b);

    if nargout == 2
        rm = rem(a, b);
    end
end
```

NB : dans ce code nous avons utiliser les fonction `floor()` et `rem()`, vous pouvez les chercher dans le help de Matlab (en tapant dans command Window les commandes : `help floor` ou bien `help rem`)

pour l'exécution :

```
>> q = myIntegerDivision(10, 7)
q = 1

>> [q, r] = myIntegerDivision(10, 7)
q = 1
r = 3
```

Nous avons également la variable `nargin` qui indiquent le nombre d'arguments d'entrée de la fonction.

Le code ci-dessous implémente une fonction `addme()` qui peut recevoir un ou deux argument. Si elle a reçu un argument `a`, elle va renvoyer $a+a$ et si elle a reçu deux argument `a` et `b`, elle va renvoyer $a+b$.

```
function c=addme(a,b)
    switch nargin
        case 2
            c=a+b;
        case 1
            c=a+a;
        otherwise
            c=0;
    end
end
```

lors de l'appel de la fonction `addme()`, `nargin` indique le nombre des variables d'entrée, ensuite `switch` décide la valeur de `c` selon la valeur de `nargin`.

Voici un exemple de l'exécution de la fonction `addme()`

```
>> addme(2) %Si la fonction reçoit un seul argument a, elle va renvoyer c=a+a
ans =
     4

>> addme(2,5) %Si elle a reçu deux argument a et b, elle va renvoyer c=a+b.
ans =
     7
```

Un autre exemple de l'utilisation de `nargin`, le code ci dessous implémente une fonction `myVector()` qui peut être appliquée à une, deux ou trois variable :

```
function res = myVector(a, b, c)
    switch nargin
        case 1
            res = 0:a ;
        case 2
            res = a:b ;
        case 3
            res = a:b:c ;
        otherwise
            error('Wrong number of params');
    end
end
```

la fonction `myVector` contrôle (lors de son exécution) le nombre d'arguments d'entrées fournies à l'aide de la variable `nargin`

l'exécution de la fonction `myVector()`:

```
>> myVector(10)
ans =
     0     1     2     3     4     5     6     7     8     9    10

>> myVector(10, 20)
ans =
    10    11    12    13    14    15    16    17    18    19    20

>> myVector(10, 2, 20)
ans =
    10    12    14    16    18    20
```

Vous pouvez également utiliser `nargin` et `nargout` comme des fonctions qui renvoient respectivement le nombre d'argument et le nombre des valeurs de retour d'une fonction

Exemple :

```
>> nargin('sin') %je cherche le nombre d'argument d'entrée de la fonction sin
ans =
     1

>> nargout('sin') %je cherche le nombre des valeurs de sortie de la fonction sin
ans =
     1

>> nargout('exp') %je cherche le nombre d'argument d'entrée de la fonction exp
```

```
ans =  
    1  
  
>> nargin('exp') %je cherche le nombre des valeurs de sortie de la fonction exp  
ans =  
    1
```

Activité 5 : Sous fonction

Nous avons vu dans la série précédente que chaque fonction doit être enregistrée dans un fichier qui porte le nom de la fonction. Cette règle à une seule exception : c'est le cas des sous fonctions.

Une sous fonction est une fonction qui a la même syntaxe qu'une fonction principale sauf que la sous fonction doit être enregistrée dans le même fichier de la fonction principale.

La sous fonction est visible seulement à partir de la fonction principale ou bien d'une autre sous fonction de la fonction principale

Exemple

Dans cet exemple nous allons implémenter la fonction `diamCricfSurface(r)` qui permet de calculer à la fois le diamètre, la circonférence et la surface d'un cercle dont le rayon est entré comme argument. Cette fonction va utiliser trois sous fonctions `diametreCercle()`, `criconferenceCercle()` et `surfaceCercle()`. Pour cela nous allons écrire dans le même fichier nommé « `diamCricfSurface.m` » le code suivant :

```
function [d c s] = diamCricfSurface(r)  
    d= diametreCercle(r);  
    c= criconferenceCercle(r);  
    s= surfaceCercle(r);  
end  
  
function d = diametreCercle(r)  
    d=r*2;  
end  
  
function c = criconferenceCercle(r)  
    c=r*2*pi;  
end  
  
function s = surfaceCercle(r)  
    s=r^2*pi;  
end
```

Pour l'exécution :

L'appel de la fonction `diamCricfSurface` se passe sans aucun problème

```
>> [x y z] = diamCricfSurface(2)  
x =  
    4  
y =  
   12.5664  
z =  
   12.5664
```

Par contre nous ne pouvons pas appeler directement les fonctions `diametreCercle()`, `criconferenceCercle()` et `surfaceCercle()` car elle ne sont visible que à l'intérieur du fichier `diamCricfSurface.m` et dont seule la fonction `diamCricfSurface()` qui peut les voir et par suite les appeler.

```
>> criconferenceCercle(4)
Undefined function 'criconferenceCercle' for input
arguments of type 'double'.

>> diametreCercle(4)
Undefined function 'diametreCercle' for input arguments
of type 'double'.

>> surfaceCercle(4)
Undefined function 'surfaceCercle' for input arguments
of type 'double'.
```