Mini cours 01 - *TPs Algo/EDP*
Matlab avancé

Table des matières

1 path

Dans Matlab/Octave, le chemin de recherche ((search path)) est une liste de dossiers parcourus pour localiser les fichiers (et donc les fonctions). Il est possible de visualiser ou modifier ce chemin à l'aide de la fonction **path** .
Voici quelques opérations courantes qu'il est possible d'effectuer :

- **path** : affiche le chemin de recherche,
- **addpath**(DIR1) : ajoute DIR1 (chaine de caractères) au chemin de recherche,
- **rmpath**(DIR1) : supprime DIR1 (chaine de caractères) du chemin de recherche,
- ...

Le chemin de recherche est crucial car il détermine l'ordre dans lequel la recherche de fonctions et de scripts s'effectue. Si plusieurs fichiers portant le même nom sont dans des dossiers différents, celui qui apparaît en premier sur le chemin de recherche sera utilisé.
Attention : suivant le logiciel utilisé (voir même les versions utilisée ?), une fonction du répertoire de travail prendra la priorité sur une fonction de même nom présente dans le chemin de recherche ... ou pas ! (exemple Matlab R2023b,R2022a pas prioritaire, Matlab R2018b prioritaire, Octave 9.2.0,9.1.0,7.3.0 prioritaire)

2 Indexage

Soit $\mathbb{A} \in \mathcal{M}_{m,n}(\mathbb{R})$ et $N = mn$.

$$\mathbb{A} = \left(\begin{array}{c|c|c} \mathbb{A}_{:,1} & \dots & \mathbb{A}_{:,n} \end{array} \right) \text{ avec } \mathbb{A}_{:,j} \in \mathbb{R}^n,\ j^{\text{ème}} \text{ colonne de } \mathbb{A}.$$

$\hookrightarrow$ stockage de tous tableaux/matrices pleines de manière contigüe en mémoire.

$$\boldsymbol{a} = \left(\begin{array}{c} \mathbb{A}_{:,1} \\ \hdashline \vdots \\ \hdashline \mathbb{A}_{:,m} \end{array} \right) \in \mathbb{R}^N$$

a. m=5;n=4;A=**rand**(m,n); a=A(:); $\hookrightarrow$ tableau 2D A vers tableau 1D a

b. m=5;n=4;b=**rand**(m*n,1); B=**reshape**(b,[m,n]); $\hookrightarrow$ tableau 1D b vers tableau 2D B

- Numérotation 2D (classique), *2D-indices*, en $(i, j) \in [\![1, m]\!] \times [\![1, n]\!]$,
- Numérotation 1D, *linear indices*, en $k \in [\![1, N]\!]$.

On note $\mathcal{M} : [\![1, m]\!] \times [\![1, n]\!] \longrightarrow [\![1, N]\!]$, $N = mn$, la bijection de la numérotation 2D à la numérotation 1D

$$K = \mathcal{M}(I, J) \quad \hookrightarrow \text{K=sub2ind([m,n],I,J)}$$

La fonction réciproque est

$$(I, J) = \mathcal{M}^{-1}(K) \quad \hookrightarrow \text{[I,J]=ind2sub([m,n],K)}$$

Ici I, J et K sont des tableaux de **même dimensions**.

```
m=4;n=3;a=1:m*n;
A=reshape(a,m,n)
A=A'
% Comment recuperer A(1,3), A(2,1) et A(3,2) ?
I=[1,2,3];J=[3,1,2]; % A(1,3)=A(I(1),J(1)), ...
K=sub2ind([n,m],I,J); % ou K=sub2ind(size(A),I,J);
b=A(K)  % => b(1)=A(1,3), b(2)=A(2,1) et b(3)=A(3,2)
A(K)=0  % =>  A(1,3)=0, A(2,1)=0 et A(3,2)=0
```

Listing 1 – `codes/indexage/exempleA.m`

```
>> m=4;n=3;a=1:m*n;
>> A=reshape(a,m,n)
A =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
>> A=A'
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
>> % Comment recuperer A(1,3), A(2,1) et A(3,2) ?
```

```
>> I=[1,2,3];J=[3,1,2]; % A(1,3)=A(I(1),J(1)), ...
>> K=sub2ind([n,m],I,J); % ou K=sub2ind(size(A),I,J);
>> b=A(K)  % => b(1)=A(1,3), b(2)=A(2,1) et b(3)=A(3,2)
b =
     3     5    10
>> A(K)=0  % =>  A(1,3)=0, A(2,1)=0 et A(3,2)=0
A =
     1     2     0     4
     0     6     7     8
     9     0    11    12
```

Listing 2 – Running`codes/indexage/exempleA.m`

Exercice 1

Soit B une matrice n-by-n existante. On veut la modifier pour avoir B(i,i)=−1 pour i impair. Comment faire ? (voir `codes/indexage/exempleB.m`)

Correction

```
>> n=4;b=1:n*n;
>> B=reshape(b,m,n)
B =
     1     5     9    13
     2     6    10    14
     3     7    11    15
     4     8    12    16
>> for i=1:2:n, B(i,i)=-1;end
>> B
B =
    -1     5     9    13
     2     6    10    14
     3     7    -1    15
     4     8    12    16
```

```
>> BB=reshape(b,m,n);
>> I=1:2:n;
>> K=sub2ind(size(BB),I,I)
K =
     1    11
>> BB(K)=-1
BB =
    -1     5     9    13
     2     6    10    14
     3     7    -1    15
     4     8    12    16
```

Listing 3 – Running`codes/indexage/exempleB.m`

◇

Remarque 1. *Pour avoir des codes performants sous Matlab/Octave, il est important de s'affranchir des "grandes" boucles, i.e. boucles avec un grand nombre d'itérations. Cette étape s'appelle la vectorisation du code. Pour celà la maitrise des indexages est primordiale.*

Soient $\mathbb{A} \in \mathcal{M}_{m,n}(\mathbb{R})$, $I \subset [\![1,m]\!]$, $J \subset [\![1,n]\!]$, $d = \#I = \#J$.

$$\boldsymbol{b} \in \mathbb{R}^d \text{ t.q. } \boldsymbol{b}(k) = \mathbb{A}(I(k),J(k)),\ \forall k \in [\![1,d]\!] \quad \hookrightarrow \text{ L=sub2ind(}\textbf{size}\text{(A),I,J); b=A(L);}$$

Soient $\mathbb{A} \in \mathcal{M}_{m,n}(\mathbb{R})$, $I \subset [\![1,m]\!]$, $d_I = \#I$, $J \subset [\![1,n]\!]$, $d_J = \#J$.
On note $\mathbb{M} \in \mathcal{M}_{d_I,d_J}(\mathbb{R})$:

$$\mathbb{M}_{i,j} = \mathbb{A}(I(i),J(j)),\quad \forall (i,j) \in [\![1,d_I]\!] \times [\![1,d_J]\!] \quad \hookrightarrow \text{ M=A(I,J);}$$

A tableau 2D m-by-n,

- A(I,J)=M; *ou* A(I,J)=A(I,J)+M; ↪ dim. of M : 1×1, $1\times d_J$, $d_I \times 1$ *or* $d_I \times d_J$
- A(I,:)=M; *ou* A(I,:)=A(I,:)+M; ↪ dim. of M : 1×1, $1\times n$, $d_I \times 1$ *or* $d_I \times n$
- A(:,J)=M; *ou* A(:,J)=A(:,J)+M; ↪ dim. of M : 1×1, $1\times d_J$, $m \times 1$ *or* $m \times d_J$.

```
>> m=4;n=3;a=1:m*n;
>> A=reshape(a,m,n)
A =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
>> I=[1,2,4];J=[2,3];
>> M=A(I,J)
M =
     5     9
     6    10
     8    12
>> A(I,J)=0
A =
     1     0     0
     2     0     0
     3     7    11
     4     0     0
>> A(I,J)=A(I,J)-M
A =
     1    -5    -9
     2    -6   -10
     3     7    11
     4    -8   -12
>> A(I,J)=A(I,J)+1
A =
     1    -4    -8
     2    -5    -9
     3     7    11
     4    -7   -11
>> A(I,J)=A(I,J)+[1,2]
A =
     1    -3    -6
     2    -4    -7
     3     7    11
     4    -6    -9
>> A(I,J)=A(I,J)+[1;2;3]
A =
     1    -2    -5
     2    -2    -5
     3     7    11
     4    -3    -6
```

Listing 4 – Running `codes/indexage/exempleC.m`

3 Cell array

Un tableau de cellules *cell array* permet de stocker différents types de données dans une seule entité.

```
>> C={10,rand(2);'test',1:3} % 2-by-2 cell array
C =
{
  [1,1] = 10
  [2,1] = test
  [1,2] =
     0.3439    0.4575
     0.4377    0.5084
  [2,2] =
     1    2    3
}
```

Pour accéder aux éléments d'un tableau de cellules, l'indexage peut s'effectuer

- avec des parenthèses (...) pour récupérer la ou les cellules spécifiées.
- avec des accolades {...} pour récupérer le contenu de la cellule ou les cellules spécifiées.

```
>> C{1,2}
ans =
    0.719763    0.804817
    0.038053    0.396806
>> C(1,2)
ans =
{
  [1,1] =
     0.719763    0.804817
```

```
    0.038053    0.396806
}
```

Voici une liste de fonctions liées aux tableaux de cellules pouvant être utiles :

- cellfun permet d'executer une fonction sur les éléments d'un tableau de cellules.

4 Fonctions

Il est possible de définir des fonctions locales à une fonction principale et celles-ci devront être définies après la fonction principale, dans le même fichier, et ne seront utilisable que dans le fichier. Dans l'exemple qui suit la fonction principale, nommée fun01 (et donc stockée dans le fichier fun01.m), utilise la fonction locale f qui elle-même utilise les deux fonctions locales **min** et g. Il est à noter que ces fonctions sont prioritaires sur toutes autres fonctions existantes de même nom. La fonction usuelle **min** prédéfinie, calculant un minimum, n'est plus accessible dans l'ensemble des fonctions du fichier fun01.m. A contrario, les fonctions locales f, g et **min** seront inaccessible en dehors du fichier fun01.m.

```
function y=fun01(x)
  x=fliplr(x);
  y=f(x); % utilise la fonction locale ci-dessous
end

function y=f(x)
  keyboard
  y=min(g(x)); % utilise min et g ci-dessous
end

function y=min(x)
  y=x+1;
end

function y=g(x)
  y=x-1;
end
```

Listing 5 – Fichier `codes/functions/fun01.m`

Voici un exemple d'utilisation :

```
>> fun01(1:10)
ans =
   10     9     8     7     6     5     4     3 ...
          2     1
```

4.1 Fonctions anonymes

Une fonction anonyme (*anonymous function*) est une fonction qui n'a pas besoin d'être stockée dans un fichier `.m` dédié, mais qui peut être définie à tout endroit dans un code et qui est du type function_handle.
Les fonction anonymes sont défines en utilisant la syntaxe :

@(argument list) expression

Toutes les variables qui ne sont pas trouvées dans la liste d'arguments doivent être connues lors de la création, et ces variables peuvent ensuite être effacées, modifiées sans que celà ne change la fonction anonyme créée.

```
>> f=@(x) x.^2
f =
@(x) x .^ 2
>> class(f)
ans = function_handle
>> f(2:3)
ans =
    4    9
>> a=randi(10,1,3)
a =
    10    10     3
```

```
>> g=@(t) a(1)+a(2)*t+a(3)*t.^2
g =
@(t) a(1)+a(2)*t+a(3)*t.^2
>> g(1)
ans = 23
>> clear a
>> g(1)
ans = 23
```

Listing 6 – Running `codes/functions/anonymous/anonymous01.m`

4.1.1 Création à partir d'une chaine de caractères

Soit $\boldsymbol{a} \in \mathbb{R}^n$, nous voulons créer une fonction anonyme permettant de calculer le polynôme P défini par

$$P(t) = \sum_{i=1}^{n} a_i t^{i-1}$$

```
>> a=randi(10,1,4)
a =
    7    6    5    1
>> I=1:length(a);
>> J=[I;I-1];J=J(:)'
J =
    1    0    2    1    3    2    4    3
>> str=sprintf('a(%d)*t.^(%d)+',J)
str = a(1)*t.^(0)+a(2)*t.^(1)+a(3)*t.^(2)+a(4)*t.^(3)+
>> str=['@(t) ',str(1:end-1)]
str = @(t) a(1)*t.^(0)+a(2)*t.^(1)+a(3)*t.^(2)+a(4)*t.^(3)
>> P=eval(str)
P =
@(t) a(1)*t.^(0)+a(2)*t.^(1)+a(3)*t.^(2)+a(4)*t.^(3)
>> clear a
>> P(2)
ans = 47
>> P(1:5)
ans =
    17    53   127   251   437
```

Listing 7 – Running `codes/functions/anonymous/anonymous02.m`

4.1.2 Multiple inputs

```
>> f=@(x,s) printf('double: %.5f, string:%s\n',x,s)
f =
@(x, s) printf ('double: %.5f, string:%s\n', x, s)
>> f(pi,'titi')
double: 3.14159, string:titi
>> g=@(x,y) sqrt(x.^2+y.^2)
g =
@(x, y) sqrt (x .^ 2 + y .^ 2)
>> g(1,2)
```

```
ans = 2.2361
```

Listing 8 – Running `codes/functions/anonymous/anonymous03.m`

4.1.3 Multiple outputs

En utilisant la fonction **deal**, on peut retourner plusieurs outputs :

```
>> g=@(x,y) deal(x.^2,y.^2,sqrt(sum(x.^2+y.^2)))
g =
@(x, y) deal (x .^ 2, y .^ 2, sqrt (sum (x .^ 2 + y .^ 2)))
>> [X,Y,N]=g(1:2,2:3)
X =
    1    4
Y =
    4    9
N = 4.2426
```

Listing 9 – Running `codes/functions/anonymous/anonymous04.m`

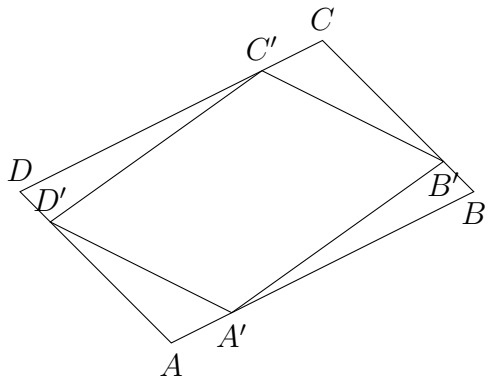
La fonction **meshgrid** retourne deux outputs, on peut alors l'invoquer directement pour créer une fonction anonyme retournant ces deux outputs :

```
>> [X1,Y1]=meshgrid(linspace(-1,1,10),linspace(-2,2,10));
>> MyMeshGrid=@(a,b,n) ...
    meshgrid(linspace(-a,a,n),linspace(-b,b,n));
>> [X2,Y2]=MyMeshGrid(1,2,10);
>> norm(X1-X2)
ans = 0
>> norm(Y1-Y2)
ans = 0
```

Listing 10 – Running `codes/functions/anonymous/anonymous05.m`

4.2 Nombre variable d'arguments en entrée

EXERCICE 2



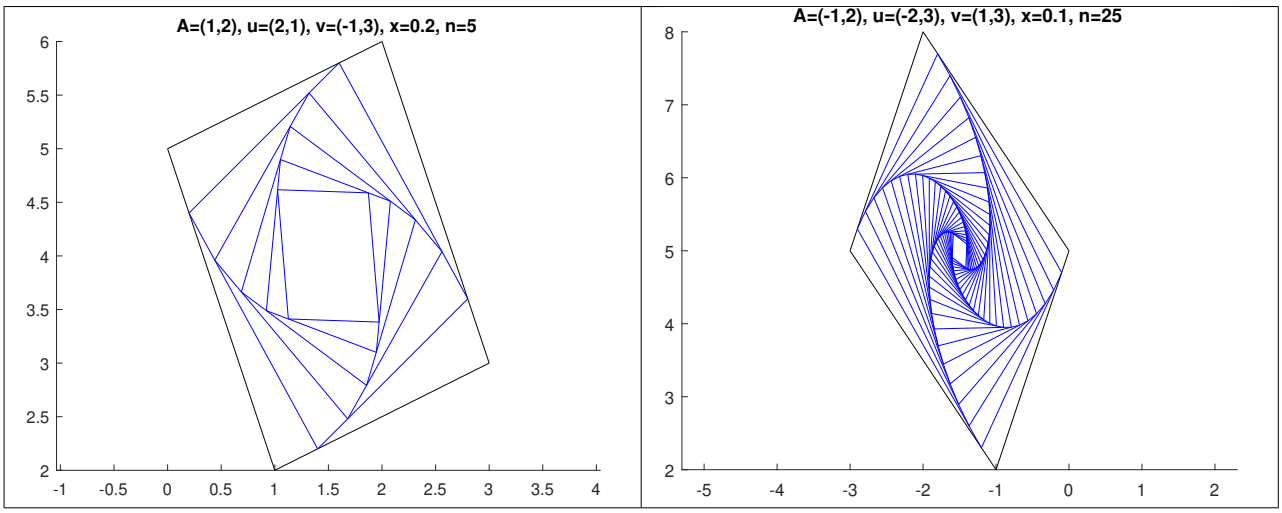
Soit un parallélogramme de sommets A, B, C et D. A partir de ce parallélogramme on peut construire un nouveau parallélogramme de sommets A', B', C' et D' vérifiant

$$\overrightarrow{AA'} = x\overrightarrow{AB}$$
$$\overrightarrow{BB'} = x\overrightarrow{BC}$$
$$\overrightarrow{CC'} = x\overrightarrow{CD}$$
$$\overrightarrow{DD'} = x\overrightarrow{DA}$$

avec x un réel compris strictement entre 0 et 1.

L'objectif est, pour un x fixé, d'itérer n fois ce processus de construction en partant à chaque itération du dernier parallélogramme construit et de représenter l'ensemble des parallélogrammes.

Q. 1 *Ecrire une fonction* `parallelogrammes` *permettant à partir du sommet A, du vecteur $\boldsymbol{u} = \overrightarrow{AB}$ et du vecteur $\boldsymbol{v} = \overrightarrow{AD}$ d'un parallélogramme initial quelconque d'aire non nulle, de représenter ce parallélogramme ainsi que les n parallélogrammes obtenus par le processus de construction décrit ci-dessus avec un x donné dans $]0,1[$. Voici deux exemples d'utilisation de cette fonction :*



□

Correction On va tout d'abord déterminer des formules qui a partir de A, $\overrightarrow{u}$, et $\boldsymbol{v}$ permettent de calculer A', $\overrightarrow{u'} = \overrightarrow{A'B'}$, et $\overrightarrow{v} = \overrightarrow{A'D'}$

- Calcul de A' :
$$A' = A + \overrightarrow{AA'} = A + x\overrightarrow{AB} = A + x\overrightarrow{u}.$$
- Calcul de B' :
$$B' = B + \overrightarrow{BB'} = B + x\overrightarrow{BC} = A + \overrightarrow{u} + x\overrightarrow{v}.$$
- Calcul de D' :
$$D' = D + \overrightarrow{DD'} = D + x\overrightarrow{DA} = A + \overrightarrow{v} - x\overrightarrow{v}.$$

On en déduit

- Calcul de $\overrightarrow{u'}$:
$$\overrightarrow{u'} = \overrightarrow{A'B'} = B' - A' = A + \overrightarrow{u} + x\overrightarrow{v} - (A + x\overrightarrow{u}) = \overrightarrow{u} + x(\overrightarrow{v} - \overrightarrow{u}).$$
- Calcul de $\overrightarrow{v'}$:
$$\overrightarrow{v'} = \overrightarrow{A'D'} = D' - A' = A + \overrightarrow{v} - x\overrightarrow{u} - (A + x\overrightarrow{u}) = \overrightarrow{v} - x(\overrightarrow{u} + \overrightarrow{v}).$$

Voici une première ébauche d'un code :

```
function parallelogrammes(A,u,v,x,n)
% exemple: parallelogrammes([-1;2],[-2;3],[1;3],0.1,25)
  clf;hold on;axis equal
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),'k')
  for i=1:n
    A=A+x*u;          % nouveau A
    utmp=u+x*(v-u);   % nouveau u
    v=v-x*(u+v);      % nouveau v
```

```
    u=utmp;
    X=[A,A+u,A+u+v,A+v,A];
    plot(X(1,:),X(2,:),'b')
  end
end
```

Listing 11 – Fonction `parallelogrammes`

◇

Les 3 premiers paramètres doivent être des vecteurs colonnes 2-by-1.
Voici deux variantes de cette fonction :

```
function parallelogrammesv1(A,u,v,x,n)
% exemple: parallelogrammesv1([-1;2],[-2;3],[1;3],0.1,25)
  clf;hold on;axis equal
  plot_para(A,u,v,'k')
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,'b')
  end
end

function plot_para(A,u,v,color)
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),color)
end

function [Ap,up,vp]=calcul(A,u,v,x)
  Ap=A+x*u;
  up=u+x*(v-u);
  vp=v-x*(u+v);
end
```

Listing 12 – Fonction `parallelogrammesv1` utilisant des fonctions locales

```
function parallelogrammesv2(A,u,v,x,n)
% exemple: parallelogrammesv2([-1;2],[-2;3],[1;3],0.1,25)
  clf;hold on;axis equal
  calcul=@(A,u,v,x) deal(A+x*u,u+x*(v-u),v-x*(u+v));
  getX=@(A,u,v) [A,A+u,A+u+v,A+v,A];
  plot_para=@(X,color) plot(X(1,:),X(2,:),color);
  X=getX(A,u,v);
  plot_para(X,'k')
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    X=getX(A,u,v);
    plot_para(X,'b')
  end
end
```

Listing 13 – Fonction `parallelogrammesv2` utilisant des fonctions anonymes

4.2.1 Utilisation de `nargin`, `varargin`

```
function parallelogrammesv3(A,u,v,x,n,color1,color2)
```

```
% Version arguments optionnels et utilisant la fonction exist
% Exemples:
%  parallelogrammesv3([-1,2],[-2,3],[1,3],0.1,25)
%  parallelogrammesv3([-1,2],[-2,3],[1,3],0.1,25,'m')
%  parallelogrammesv3([-1,2],[-2,3],[1,3],0.1,25,'m','g')
%  parallelogrammesv3([-1,2],[-2,3],[1,3],0.1,25,[],'g')
  if ~exist('color1') || isempty(color1), color1='k';end
  if ~exist('color2') || isempty(color2), color2='b';end
  clf;hold on;axis equal
  plot_para(A,u,v,color1)
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,color2)
  end
end

function plot_para(A,u,v,color)
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),color)
end

function [Ap,up,vp]=calcul(A,u,v,x)
  Ap=A+x*u;
  up=u+x*(v-u);
  vp=v-x*(u+v);
end
```

Listing 14 – Fonction `parallelogrammesv3` avec arguments optionnels, utilisant la fonction **exist**

```
function parallelogrammesv4(A,u,v,x,n,color1,color2)
% Version arguments optionnels et utilisant la fonction nargin
% Exemples:
%  parallelogrammesv4([-1,2],[-2,3],[1,3],0.1,25)
%  parallelogrammesv4([-1,2],[-2,3],[1,3],0.1,25,'m')
%  parallelogrammesv4([-1,2],[-2,3],[1,3],0.1,25,'m','g')
%  parallelogrammesv4([-1,2],[-2,3],[1,3],0.1,25,[],'g')
  assert(ismember(nargin,5+[0:2]),sprintf('Number of input ...
     parameters: 5 to 7, given %d',nargin))
  if nargin<6 || isempty(color1), color1='k';end
  if nargin<7 || isempty(color2), color2='b';end
  clf;hold on;axis equal
  plot_para(A,u,v,color1)
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,color2)
  end
end

function plot_para(A,u,v,color)
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),color)
end

```

```
function [Ap,up,vp]=calcul(A,u,v,x)
  Ap=A+x*u;
  up=u+x*(v-u);
  vp=v-x*(u+v);
end
```

Listing 15 – Fonction `parallelogrammesv4` avec arguments optionnels, utilisant la fonction **nargin**

Dans l'exemple qui suit, on utilise la variable varargin qui permet de passer des arguments optionnels à une fonction. Cette variable est de type tableau de cellules (*cell array*).

```
function parallelogrammesv5(A,u,v,x,n,varargin)
% Version avec arguments optionnels utilisant varargin et nargin
% Exemples:
%  parallelogrammesv5([-1,2],[-2,3],[1,3],0.1,25)
%  parallelogrammesv5([-1,2],[-2,3],[1,3],0.1,25,'m')
%  parallelogrammesv5([-1,2],[-2,3],[1,3],0.1,25,'m','g')
%  parallelogrammesv5([-1,2],[-2,3],[1,3],0.1,25,[],'g')
  assert(ismember(nargin,5+[0:2]),sprintf('Number of input ...
     parameters: 5 to 7, given %d',nargin))
  color1='k';color2='b';
  if nargin>=6 && ~isempty(varargin{1}),color1=varargin{1};end
  if nargin==7 && ~isempty(varargin{2}),color2=varargin{2};end
  clf;hold on;axis equal
  plot_para(A,u,v,'k')
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,'b')
  end
end

function plot_para(A,u,v,color)
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),color)
end

function [Ap,up,vp]=calcul(A,u,v,x)
  Ap=A+x*u;
  up=u+x*(v-u);
  vp=v-x*(u+v);
end
```

Listing 16 – Fonction `parallelogrammesv5` avec arguments optionnels, utilisant **nargin** et varargin

4.2.2 Utilisation de `inputParser`

```
function parallelogrammesv10(A,u,v,x,n,varargin)
% Version avec arguments optionnels utilisant inputParser
%  parallelogrammesv10([-1;2],[-1;3],[1;3],0.1,25)
%  parallelogrammesv10([-1;2],[-1;3],[1;3],0.1,25, ...
   'color1','b','color2','r')
%  parallelogrammesv10([-1;2],[-1;3],[1;3],0.1,25, 'color1','g')
%  parallelogrammesv10([-1;2],[-1;3],[1;3],0.1,25, 'color2','m')
  ip = inputParser;
```

```
  ip.PartialMatching=false;
  ip.addParameter('color1','k');
  ip.addParameter('color2','b');
  ip.parse(varargin{:});
  clf;hold on;axis equal
  plot_para(A,u,v,ip.Results.color1)
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,ip.Results.color2)
  end
end

function plot_para(A,u,v,color)
  X=[A,A+u,A+u+v,A+v,A];
  plot(X(1,:),X(2,:),color)
end

function [Ap,up,vp]=calcul(A,u,v,x)
  Ap=A+x*u;
  up=u+x*(v-u);
  vp=v-x*(u+v);
end
```

Listing 17 – Fonction `parallelogrammesv10` avec inputParser

```
function parallelogrammesv11(A,u,v,x,n,varargin)
% exemples:
%  parallelogrammesv11([-1;2],[-1;3],[1;3],0.1,25)
%  parallelogrammesv11([-1;2],[-1;3],[1;3],0.1,25, ...
    'color1','b','color2','r','linewidth',2,'linestyle',':')
  ip = inputParser;
  ip.KeepUnmatched=true;
  ip.PartialMatching=false;
  checkcolor=@(x)   ( ischar(x) && ...
     ismember(x,{'k','r','g','b','y','m','c'} ) ...
                  || ( isnumeric(x) && all(size(x)==[1,3])) );
  ip.addParameter('color1','k',checkcolor);
  ip.addParameter('color2','b',checkcolor);
  ip.parse(varargin{:});
  if ~exist('namedargs2cell'), % Matlab < R2019b, Octave < 6.1.0
    namedargs2cell = ...
    @(s) reshape ([fieldnames(s), struct2cell(s)].', 1, []);
  end
  opts=namedargs2cell(ip.Unmatched);
  clf;hold on; axis equal
  plot_para(A,u,v,'color',ip.Results.color1,opts{:})
  for i=1:n
    [A,u,v]=calcul(A,u,v,x);
    plot_para(A,u,v,'color',ip.Results.color2,opts{:})
  end
end

function plot_para(A,u,v,varargin)
  X=[A,A+u,A+u+v,A+v,A];
```

```
28    plot(X(1,:),X(2,:),varargin{:})
29  end
30
31  function [Ap,up,vp]=calcul(A,u,v,x)
32    Ap=A+x*u;
33    up=u+x*(v-u);
34    vp=v-x*(u+v);
35  end
```

Listing 18 – Fonction `parallelogrammesv11` avec inputParser

4.3 Nombre variable d'arguments en sortie

4.3.1 Utilisation de `nargout`, `varargout`

5 Structures

une structure (ou « tableau de structures ») regroupe un ensemble de données à l'aide de conteneurs appelés champs (*fields*). Chaque champ peut contenir n'importe quel type de données. Voici trois exemples permettant de créer un tableau de structure simple.

```
>> clear all
>> S.name='AMD Ryzen 5 7600X3D';
>> S.MultiThreadRating=26140;
>> S.SingleThreadRating=3633;
>> S.Cores=6;
>> S.Threads=12;
>> disp(S)
     name = AMD Ryzen 5 7600X3D
     MultiThreadRating = 26140
     SingleThreadRating = 3633
     Cores = 6
     Threads = 12
>>
>> S(2).name='Intel Core i7-13700KF';
>> S(2).MultiThreadRating=46285;
>> S(2).SingleThreadRating=4355;
>> S(2).Cores=[8,8];     % [Performance Cores, Efficient Cores]
>> S(2).Threads=[16,8]; % [Performance Cores, Efficient Cores]
>> disp(S)
  1x2 struct array containing the fields:
    name
    MultiThreadRating
    SingleThreadRating
    Cores
    Threads
```

Listing 19 – Création d'un tableau de structure simple, running `codes/structures/struct01.m`

```
>> clear all
>> S=struct('name','AMD Ryzen 5 7600X3D','MultiThreadRating',26140, ...
   'SingleThreadRating',3633, 'Cores', 6, 'Threads',12);
>> disp(S)
     name = AMD Ryzen 5 7600X3D
     MultiThreadRating = 26140
```

```
    SingleThreadRating = 3633
    Cores = 6
    Threads = 12
>>
>> S(2).name='Intel Core i7-13700KF';
>> S(2).MultiThreadRating=46285;
>> S(2).SingleThreadRating=4355;
>> S(2).Cores=[8,8];     % [Performance Cores, Efficient Cores]
>> S(2).Threads=[16,8]; % [Performance Cores, Efficient Cores]
>>
>> S(3)=struct('name','Intel Core i3-14100F','MultiThreadRating',15333, ...
   'SingleThreadRating',3770, 'Cores', 4, 'Threads',8);
>> disp(S)
  1x3 struct array containing the fields:
    name
    MultiThreadRating
    SingleThreadRating
    Cores
    Threads
```

Listing 20 – Création d'un tableau de structure simple, running `codes/structures/struct02.m`

```
>> clear all
>> setStructElem=@(name,MTR,STR,cores,threads) ...
   struct('name',name,'MultiThreadRating',MTR, ...
   'SingleThreadRating',STR, 'Cores',cores, 'Threads',threads);
>> S(1)=setStructElem('AMD Ryzen 5 7600X3D',26140,3633,6,12);
>> disp(S)
    name = AMD Ryzen 5 7600X3D
    MultiThreadRating = 26140
    SingleThreadRating = 3633
    Cores = 6
    Threads = 12
>> S(2)=setStructElem('Intel Core i7-13700KF',46285,4355,[8,8],[16,8]);
>> S(3)=setStructElem('Intel Core i3-14100F',15333,3770,4,8);
>> disp(S)
  1x3 struct array containing the fields:
    name
    MultiThreadRating
    SingleThreadRating
    Cores
    Threads
```

Listing 21 – Création d'un tableau de structure simple, running `codes/structures/struct03.m`

Pour accéder au contenu du champ field1 d'une structure existante X : X.field1

```
>> clear all
>> setStructElem=@(name,MTR,STR,cores,threads) ...
   struct('name',name,'MultiThreadRating',MTR, ...
   'SingleThreadRating',STR, 'Cores',cores, 'Threads',threads);
>> S(1)=setStructElem('AMD Ryzen 5 7600X3D',26140,3633,6,12);
>> S(2)=setStructElem('Intel Core i7-13700KF',46285,4355,[8,8],[16,8]);
>> S(3)=setStructElem('Intel Core i3-14100F',15333,3770,4,8);
```

```
>> S(3).name=1; % peut-etre d'un type different!
>> C={S.name}   % biensur [S.name] n'est pas possible!
C =
{
  [1,1] = AMD Ryzen 5 7600X3D
  [1,2] = Intel Core i7-13700KF
  [1,3] = 1
}
>> A1=[S.Cores]
A1 =
   6    8    8    4
>> A2={S.Cores}
A2 =
{
  [1,1] = 6
  [1,2] = 8    8
  [1,3] = 4
}
```

Listing 22 – Running `codes/structures/struct04.m`

Voici une liste de fonctions liées aux structures (ou tableau de structure) :

- isstruct(X) retourne vrai si X est une structure ou un tableau de structure,
- fieldnames(X) retourne un tableau de cellules de chaine de caractères avec les noms des champs de la structure ou tableau de structure X,
- isfield(X,name) retourne vrai si la structure X contient un champ nommé name (chaine de caractères).
- structfun (seulement sur une structure)
- arrayfun(fun,A) (seulement sur un tableau de structure) execute une fonction en chaque élément d'un tableau.

```
>> setStructElem=@(name,MTR,STR,cores,threads) ...
    struct('name',name,'MultiThreadRating',MTR, ...
    'SingleThreadRating',STR, 'Cores',cores, 'Threads',threads);
>> S(1)=setStructElem('AMD Ryzen 5 7600X3D',26140,3633,6,12);
>> S(2)=setStructElem('Intel Core ...
    i7-13700KF',46285,4355,[8,8],[16,8]);
>> S(3)=setStructElem('Intel Core i3-14100F',15333,3770,4,8);
>> idx=arrayfun(@(x) x.MultiThreadRating>20000,S)
idx =
  1  1  0
>> SF=S(idx);
>> {SF.name}
ans =
{
  [1,1] = AMD Ryzen 5 7600X3D
  [1,2] = Intel Core i7-13700KF
}
```

Listing 23 – Utilisation de arrayfun sur un tableau de structure simple, running `codes/structures/struct05.m`

6 *Namespaces*

Les *namespaces* sont des répertoires spéciaux, de premier caractère `'+'`, pouvant contenir

- des fichiers de fonctions,
- des fichiers scripts (programmes),
- des répertoires de *class*,
- d'autres *namespaces*.

Pour pouvoir utiliser un *namespace* il faut se positionner dans le répertoire le contenant ou ajouter ce répertoire dans le *path* avec la commande **addpath** par exemple.
Voici un exemple d'arborescence de *namespace* :

```
+toto/fun1.m
+toto/fun2.m
+toto/prg1.m
+toto/+titi/fun1.m
+toto/+titi/fun3.m
+toto/+titi/prg1.m
```

```
function y=fun1(x)
  y=x.^2;
end
```

Listing 24 – `+toto/fun1.m`

```
function y=fun2(x)
  y=toto.fun1(x)+1;
end
```

Listing 25 – `+toto/fun2.m`

```
clear all;close all
X=1:3
Y1=toto.fun2(X)
Y3=toto.titi.fun3(X)
```

Listing 26 – `+toto/prg1.m`

```
function y=fun1(x)
  y=x-1;
end
```

Listing 27 – `+toto/+titi/fun1.m`

```
function y=fun3(x)
  y=toto.fun1(x);
  y=y+toto.titi.fun1(x);
end
```

Listing 28 – `+toto/+titi/fun3.m`

```
clear all;close all
X=1:3
Y1=toto.fun1(X)
Y3=toto.titi.fun3(X)
```

Listing 29 – `+toto/+titi/prg1.m`

Voici le résultat de l'execution des deux programmes `+toto/prg1.m` et `+toto/+titi/prg1.m`

```
>> clear all;close all
>> X=1:3
X =
    1    2    3
>> Y1=toto.fun2(X)
Y1 =
     2     5    10
>> Y3=toto.titi.fun3(X)
Y3 =
     1     5    11
```

Listing 30 – output de `toto.prg1`

```
>> clear all;close all
>> X=1:3
X =
    1    2    3
>> Y1=toto.fun1(X)
Y1 =
```

```
   1     4     9
>> Y3=toto.titi.fun3(X)
Y3 =
     1      5     11
```

Listing 31 – output de `toto.titi.prg1`

Attention : avec Octave (7.3.0, 9.1.0, 9.2.0), il semble que le nom d'un namespace doive différer du nom d'une fonction déjà existente. Par exemple le namespace `+mesh` n'est pas *valide* car la fonction `mesh` est une fonction interne.

7 Débogage

- **keyboard** : stoppe l'execution d'un programme à l'endroit exacte où cette commande est écrite et entre en mode debogage.
- **dbstop** : défini des *breakpoints* (voir aide).
- **dbcont** : reprend l'execution du code (voir aide).
- **dbstep** : execute la ligne suivante du code (voir aide).
- **dbquit** : quitte immediatement le mode debogage sans executer la suite du code.
- ...

8 Objets