Chapitre 3 - Scripts et fonctions dans Matlab

Jusqu'ici, nous n'avons considéré que la fenêtre de commandes, cette approche est pratique pour démarrer, ou pour réaliser de petits calculs, mais elle n'est pas appropriée dès qu'on considère des problèmes plus complexes.

Nous allons voir dans ce qui suit, comment utiliser Matlab comme un véritable langage de programmation, en passant des fichiers de commandes (qu'on peut sauvegarder et donc réutiliser), en écrivant nos propres fonctions, script et en utilisant des structures de contrôle.

1. Qu'est ce qu'un script

Un script est une séquence d'instructions et de commandes Matlab. Les différentes instructions et commandes doivent être séparées par une virgule, un point-virgule ou saut de ligne. Les instructions s'exécutent dans leur ordre d'écriture, et seules les valeurs des expressions qui ne sont pas suivies d'un point-virgule sont affichées.

Matlab permet d'enregistrer les scripts dans des fichiers texte avec l'extension ".m" appelés m-files et de les réutiliser même après fermeture et réouverture de Matlab.

1.1. Écrire un script

Pour écrire un script, on peut utiliser l'éditeur fourni par l'environnement Matlab (ou un autre éditeur comme : gedit, notepad ... etc.). Une fois terminé, un script doit être enregistré dans un fichier avec l'extension ".m" (appelé *m-file* dans quelques références).

Pour exécuter un script, il suffit de l'appeler par son nom dans la fenêtre de commandes, sans préciser l'extension '.m'. C'est pourquoi il est conseillé de choisir des noms significatifs, permettant une identification rapide de l'utilité des scripts créés.

Exemple : Enregistrer et utiliser un script

```
% Script - discSurf
rayon = 5
surface = rayon^2 * pi
```

```
>> discSurf

rayon =

     5
```

```
6
7
8 surface =
9
10    78.5398
```

Cet exemple montre la création et l'exécution d'un script qui calcule la surface d'un disque de rayon "5".

2. Instructions dans un script

Cette section présente les différentes catégories d'instructions qu'on peut utiliser dans un script.

2.1. Les instructions d'entrée-sortie

Les fonctions `input` et `disp` (ou `display`) permettent respectivement à l'utilisateur d'introduire la valeur d'une variable ou d'afficher cette dernière sur l'écran. La syntaxe suivante est utilisée pour la fonction `input` :

```
variable = input('message indicatif');
```

L'instruction `input` a pour effet d'attendre que l'utilisateur saisisse une valeur au clavier et la valide en appuyant sur la touche `entrée` (ou `enter` dans un clavier anglais). La valeur saisie par l'utilisateur est par la suite affectée à la variable à gauche de l'opération d'affectation. Le message passé comme paramètre s'affiche à l'écran pour indiquer à l'utilisateur ce qu'il doit faire, mais il n'a pas d'influence sur le déroulement du programme (ou script).

Pour la fonction `disp` nous utilisons la syntaxe suivante :

```
disp(variable);
```

L'argument de la fonction `disp` est soit du texte brut écrit entre guillemets, soit une variable dont on veut afficher la valeur ou une expression à évaluer avant d'afficher sa valeur. Dans le cas de texte brut, ce dernier apparaît tel quel à l'écran. Dans le cas d'une expression, c'est le résultat du calcul de cette expression qui est affiché.

Exemple : Utiliser les instructions input et disp

```
1 % Script - add.m
2 a = input('introduisez la valeur de a : ');
3 b = input('introduisez la valeur de b : ');
4 somme = a + b;
5 disp(somme);
```

Le script dans cet exemple utilise la fonction input pour demander à l'utilisateur de saisir la valeur des deux variables (le message indicatif lui sera affiché) `a` et `b`, puis il calcule et affiche la somme de ces deux nombres en utilisant la fonction disp. Nous montrons ci-dessous un scénario d'exécution de ce script.

```
1 >> add
2 introduisez la valeur de a : 1
3 introduisez la valeur de b : 2
4      3
```

Attention : "disp" ou "display"

Les deux fonctions `disp` et `display` ont pour mission d'afficher un message (ou une valeur) à l'écran. La seule différence entre les deux est que la fonction `disp` affiche seulement la valeur de la variable passée comme argument, alors que la fonction `display` afficher sa valeur, ainsi que son identifiant.

Exemple : Utilisation de "disp" ou "display"

```
>> x = 3;
>> disp(x);
     3

>> display(x);

x =

     3
```

2.2. Structures de contrôle

Les structures de contrôle permettent une exécution conditionnelle ou répétitive de quelques instructions. Ces derniers forment ce que l'on appelle communément un bloc d'instructions. Les structures de contrôle forment ainsi des blocs, délimités par un mot clé spécifique (donnant le sens de la structure de contrôle) et le mot clé `end`.

Les structures de contrôle sont utiles dans le cas ou l'avancement de l'exécution d'un script peut prendre des chemins différents selon l'évolution des variables utilisées. L'exemple le plus simple d'un problème mathématique qui nécessite l'utilisation des structures de contrôle est la résolution d'une équation de second degré. Cette dernière a deux, une ou aucune solution en fonction de la valeur de `delta`. Donc un script résolvant ce problème devra adapter son comportement en fonction des valeurs prises par certaines variables.

2.2.1. Les instructions conditionnelles (if ... else ... end)

Définition : Condition

Une condition est une expression dont le résultat est une valeur logique, pour cela nous utilisons des opérateurs logiques. Les opérateurs de comparaison (<, >, =, ≤, ≥, ≠) sont les plus utilisés pour tester des conditions (par exemple : `if a > b`).

Le résultat issu d'un de ces tests peut prendre deux valeurs : `true` ou `false`.

Complément : Opérateurs logiques de base

Les trois opérateurs logiques de base (`&, |, ~`) sont utilisés pour mieux manipuler les conditions (par exemple : tester si plusieurs conditions sont satisfaites).

- L'opérateur `&` (le et logique) : `cond1 & cond2` retourne la valeur logique `true` si et seulement si la condition `cond1` et la condition `cond2` retournent toutes les deux la valeur logique `true`, sinon le résultat retourné est la valeur logique `false`.
- L'opérateur `|` (le ou logique) : `cond1 | cond2` retourne la valeur logique `true` si et seulement si au moins l'une des deux conditions `cond1` et `cond2` retournent la valeur logique `true`, sinon le résultat retourné est la valeur logique `false`.

- L'opérateur ~ (la négation logique) :`~cond` retourne la valeur logique `true` si `cond` retourne la valeur logique `false`, sinon retourne la valeur logique `false`.

Attention : Symboles des opérateurs logiques dans Matlab

Comme vous l'avez peut être remarqué, l'opérateur = est utilisé pour effectuer une affectation, donc le même opérateur ne peut pas être utilisé pour une opération de comparaison. D'un autre côté les caractères ≤, ≥ et ≠ ne sont pas présents sur le clavier. Pour simplifier l'écriture d'un code Matlab et éviter des ambiguïtés des opérations présentes dans le code, le tableau suivant montre les caractères spéciaux utilisés par Matlab.

Symboles	Signification
=	Affectation
==	Test d'égalité
~=	Test d'inégalité (équivalent de ≠)
« <= »	L'équivalent de ≤
>=	L'équivalent de ≥
|	L'équivalent du `ou` logique
&	L'équivalent du `et` logique
~	L'équivalent de la négation logique

Tableau 3 : Les symboles utilisés par Matlab et leurs significations

L'instruction if ... end

La commande if permet de tester une condition avant d'exécuter une séquence d'instruction. La syntaxe suivante est utilisée :

```
if condition
   bloc d'instructions
end
```

Le bloc d'instructions entre le test de la condition et la commande `end` n'est exécuté que si la condition testée est satisfaite, autrement les instructions ne sont pas exécutées. Il arrive souvent qu'un programme doive exécuter un bloc d'instructions si une condition est vraie, et un autre bloc si cette même condition est fausse. Plutôt que de tester une condition puis son contraire, il est possible d'utiliser la structure `if ... else ... end`, dont la syntaxe est la suivante :

```
if condition
   bloc d'instructions 1
else
   bloc d'instructions 2
```

`end`

Notez qu'un seul des deux blocs est exécuté (`bloc d'instructions 1` et `bloc d'instructions 2`). Le premier bloc est exécuté si la condition est satisfaite, et le deuxième est exécuté sinon. Cette structure fonctionne de la manière suivante :

- si la condition testée est satisfaite, alors le premier bloc d'instructions est exécuté ;
- sinon, le deuxième bloc d'instructions est exécuté ;
- les instructions après la commande `end` (s'il y en a) sont exécutées par la suite.

Exemple : Utilisation de if ... end

Dans cet exemple nous allons écrire un algorithme qui demande à l'utilisateur de saisir deux nombres : `a` et `b`, puis il calcule et affiche le résultat de la division de `a` par `b`. Évidement la division par zéro ne peut pas être effectuée, donc si l'utilisateur donne la valeur `0` à la variable `b` l'opération de division ne doit pas être exécutée et le résultat n'est pas affiché.

```
% Script - division.m
x = input(`Donnez la valeur du numerateur : ');
y = input(`Donnez la valeur du denominateur : ');
if (y ~= 0)
   res = x / y;
   disp(res);
end;
```

Le script précédent n'effectue la division par `y` lorsque ce dernier est différent de 0. Autrement la division n'est pas effectuée et rien n'est affiché par le programme. Nous allons maintenant améliorer ce script pour qu'il affiche un message d'erreur indiquant qu'il est impossible de diviser par 0.

Pour cela nous allons utiliser la commande `else`.

```
% Script - division.m
x = input(`Donnez la valeur du numerateur : ');
y = input(`Donnez la valeur du denominateur : ');
if (y == 0)
   disp('On ne peut pas diviser par zero');
else
   res = x / y;
   disp(res);
end;
```

L'exécution de la deuxième version de ce script est exposée ci-dessous

```
>> division
Donnez la valeur du numerateur : 5
Donnez la valeur du denominateur : 2
    2.5000

>> division
Donnez la valeur du numerateur : 5
Donnez la valeur du denominateur : 0
On ne peut pas diviser par zero
```

Complément : La commande elseif

Lorsqu'il y a plus de deux alternatives, la commande `elseif` est utilisée pour tester plusieurs possibilités. La syntaxe de l'instruction `elseif` est la suivante :

```
if (condition 1)
   bloc d'instructions 1
elseif (condition 2)
   bloc d'instructions 2
elseif (condition 3)
.
.
.
elseif condition n)
   bloc d'instructions n
else
   bloc d'instructions n + 1
end;
```

Chaque bloc d'instructions *i* est exécuté si et seulement si la condition *i* est vraie, et toutes les conditions *j* pour $j < i$ sont fausses. Le dernier bloc d'instructions *(n + 1)* n'est exécuté que si toutes les conditions testées sont fausses.

Exemple : Utilisation de la commande elseif

Pour illustrer l'utilisation de la commande `elseif` nous allons écrire un script Matlab permettant de lire un entier et puis faire le traitement suivant :

- Afficher le message "positif" si le nombre saisi est strictement supérieur à zéro.
- Afficher le message "négatif" si le nombre saisi est strictement inférieur à zéro.
- Afficher le message "null" si le nombre saisi vaut zéro.

```
1 % Script - signe.m
2 n = input('Veuillez S.V.P saisir un nombre : ');
3 if n > 0
4     disp('positif');
5 elseif n < 0
6     disp('négatif');
7 else
8     disp('null');
9 end
```

Un exemple d'exécution de ce script est montré ci-dessous

```
1 >> signe
2 Veuillez S.V.P saisir un nombre : 4
3 positif
4 >> signe
```

```
Veuillez S.V.P saisir un nombre : -5
négatif
>> signe
Veuillez S.V.P saisir un nombre : 0
null
```

2.2.2. Les structures itératives

Certains algorithmes nécessitent la répétition de certaines instructions plusieurs fois avant d'obtenir le résultat voulu. Cette répétition est réalisée en utilisant une structure de contrôle de type itératif, nommée boucle.

Une boucle est une structure qui permet d'exécuter un certain nombre de fois un même bloc d'instructions. Nous distinguons dans cette section deux types de boucles, la boucle `while` qui sert à répéter l'exécution d'un bloc d'instructions tant qu'une condition est satisfaite, et la boucle `for` qui sert à répéter l'exécution d'un bloc d'instructions pour un nombre fini de valeurs différentes.

Complément : La boucle while ... end

La boucle `while` permet de répéter l'exécution d'un bloc d'instructions tant qu'une condition reste vérifiée. On arrête de boucler dès que cette condition n'est plus satisfaite. Ce processus est mis en œuvre en utilisant la boucle `while`. Lorsque la condition testée n'est plus satisfaite, on passe à l'instruction qui suit immédiatement l'instruction `end` (marquant la fin du bloc à répéter). La syntaxe utilisée pour la boucle `while` est la suivante :

```
while condition
   bloc d'instructions
end;
```

Exemple : Utilisation de la boucle while ... end

Pour expliquer le principe de la boucle `while`, nous allons reprendre le script précédent qui permet de lire deux nombres (`x` et `y`) saisis par l'utilisateur, et puis calculer `x / y`. Dans l'exemple précédent nous avons effectué un test pour assurer que `y` est différent de 0 avant de faire la division. Dans cet exemple nous allons aller plus loin et demander à l'utilisateur de donner une autre valeur à `y`. Pour cela nous utilisons une boucle `while` qui permet de demander à l'utilisateur de ressaisir `y` à chaque fois qu'il donne une valeur nulle.

```
% Script - boucle.m
x = input('Veuillez donner la valeur de x : ');
y = input('Veuillez donner la valeur de y : ');
while y == 0
    y = input('Veuillez donner une valeur différente de 0 à y : ');
end;
z = x / y;
disp(z);
```

L'exécution de ce script est illustrée ci-dessous :

```
>> boucle
Veuillez donner la valeur de x : 6
Veuillez donner la valeur de y : 0
Veuillez donner une valeur différente de 0 à y : 0
Veuillez donner une valeur différente de 0 à y : 3
     2
```

Complément : La boucle for ... end

Il est possible d'utiliser une boucle `while` pour parcourir les éléments d'un vecteur (ligne ou colonne). Cette utilisation des boucles est très fréquente, c'est pour cela qu'une autre structure de boucle est mise à notre disposition. Nous parlons de la boucle `for`. La boucle `for` répète l'exécution d'un bloc d'instructions pour tous les éléments d'un tableau donné `T`. Pour faire cela, nous utilisons la syntaxe suivante :

```
for k = T
   bloc d'instructions
end;
```

La boucle précédente exécute le bloc d'instructions (entre `for` et `end`) pour `x` prenant la valeur de chaque élément du vecteur `T`.

Remarque : La relation entre la boucle for ... end et la boucle while ... end

la structure `for ... end` n'est pas indispensable. C'est à dire qu'on peut programmer toutes les situations de boucle `for` en utilisant la boucle `while`. Le seul intérêt derrière la boucle `for` est d'épargner un peu de fatigue au programmeur, en lui évitant de gérer lui-même la variable qui sert à parcourir les éléments du tableau. Autrement dit, la boucle `for` est un cas particulier de la boucle `while`.

La boucle `for` précédente peut s'écrire sous la forme d'une boucle `while` de la façon suivante :

```
i = 1
while i <= length(T)
   x = T(i)
   bloc d'instructions
i = i + 1
end;
```

Exemple : Utiliser la boucle for ... end

Cet exemple illustre deux scripts permettant d'afficher tous les éléments d'un vecteur. Les deux scripts répondent au même problème en utilisant deux structures de code différentes (boucle `for` et boucle `while`).

```
1 % Script - boucleFor.m
2 tableau = 2:6;
3 for element = tableau
4     disp(element);
5 end;
```

```
1 % Script - boucleWhile.m
2 tableau = 2:6;
3 i = 1;
4 while i <= length(tableau)
5     element = tableau(i);
6     disp(element);
7     i = i + 1;
8 end;
```

L'exécution des deux codes donne le même résultat comme nous le montrons dans le code suivant :

```
>> boucleFor
     2

     3

     4

     5

     6

>> boucleWhile
     2

     3

     4

     5

     6

```

Remarque : L'opérateur colon dans la boucle for

L'opérateur colon ":" pourrait être utilisé dans la boucle `for` pour créer le vecteur et itérer dessus sans avoir besoin de mettre le vecteur dans une autre variable. L'exemple précédent peut être ré-écrit de la façon suivante (cela ne change pas le comportement du script).

```
% Script - boucleFor.m
for element = 2:6
    disp(element);
end;
```

Conseil : Quelle boucle utiliser

Nous avons montré dans cette section que deux structures différentes peuvent être utilisées pour faire des boucles, la boucle `for` et la boucle `while`. Nous donnons ici quelques conseils pour choisir quelle boucle utiliser :

- La structure `while` doit être employée dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont on ne connaît pas d'avance la quantité, par exemple : le contrôle d'une saisie.
- La structure `for` doit être employée dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble (ou d'un tableau) dont le programmeur connaît d'avance la quantité.

2.3. Appel de fonctions

Les instructions que comporte un script peuvent aussi être des appels de fonctions. Ces fonctions sont soit prédéfinies dans l'environnement Matlab, soit définies et personnalisées par l'utilisateur. En plus des fonctions que nous avons déjà présentées dans le chapitre précédent, voici une petite liste de fonctions de calcul mathématique qui sont prédéfinies dans Matlab.

- Fonctions trigonométriques et inverses : `sin`, `cos`, `tan`, `asin`, `acos`, `atan`.
- Fonctions hyperboliques : `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.

- Racine, logarithmes et exponentielles : `sqrt`, `log`, `log10`, `exp`.

La plupart de ces fonctions acceptent comme argument des nombres réels ou des nombres complexes. Et, elles acceptent aussi des valeurs simples ou sous forme de tableaux.

Exemple : Utiliser des fonctions

On retiendra que pour appliquer une fonction à une valeur ou une variable, il faut mettre cette dernière entre parenthèses comme le montre l'exemple suivant :

```
>> sin(pi/2)

ans =

     1
```

3. Définir des fonctions

À côté des fonctions prédéfinies, MATLAB offre à l'utilisateur la possibilité de définir d'autres fonctions personnalisées. Comme pour les scripts, une fois que la fonction est définie, elle doit être enregistrée dans un fichier portant le même nom de la fonction avec l'extension '.m'. La syntaxe suivante est utilisée pour définir une fonction :

```
function sortie = nomFonction(listeEntrées)
   bloc d'instructions
```

Une fonction qui a été définie peut être utilisée à partir de la fenêtre de commande, ou bien dans un script de la même façon qu'on utilisait les fonctions prédéfinies.

Exemple : La fonction moyenne

Dans cet exemple nous allons définir une fonction qui calcule la moyenne de deux nombres passés comme arguments.

```
% Fonction - moyenne.m
function res = moyenne(a, b)
res = (a + b) / 2;
```

Nous montrons ci-dessous comment utiliser la fonction `moyenne`.

```
>> moyenne(3, 4)

ans =

    3.5000
```

Complément : Fonctions ayant plusieurs sorties

Il est possible de définir une fonction ayant plusieurs sorties en suivant la syntaxe suivante.

```
function [listeSorties] = nomFonction(listeEntrées)
bloc d'instructions
```

La gestion des variables de sortie est très souple sous Matlab. Si l'on n'est intéressé que par la première valeur retournée par la fonction, on peut se contenter de ne mettre qu'une seule variable de sortie à l'utilisation de la fonction, `x = nomFonction(listeEntrées)`. Par contre, même si l'on ne souhaite recueillir la deuxième valeur, on est obligé de récupérer la première aussi et donc de définir une variable inutile. Aussi, d'une manière générale, il est bon de ranger les variables de sortie par ordre d'importance.

Exemple : Résoudre une équation de deuxième degré

Nous allons dans cet exemple définir une fonction permettant de résoudre une équation linéaire de deuxième degré. La fonction que nous définissons retourne la valeur de `delta`, x_1 et x_2. Nous n'allons pas séparer les cas ou la valeur de `delta` est inférieure ou égale à 0.

```
% Fonction - solveEquation.m
function [delta, X1, X2] = solveEquation(a, b, c)
delta = b^2 - 4 * a * c;
X1 = (-b - sqrt(delta)) / (2 * a);
X2 = (-b + sqrt(delta)) / (2 * a);
```

Dans l'exemple précédent, si `delta` vaut zéro, nous obtenons la même solution dans `X1` et `X2`. Si `delta` est inférieur à zéro, nous obtenant des valeurs complexes pour `X1` et `X2`. Ci-dessous nous montrons comment peut-on utiliser cette fonction pour résoudre l'équation $x^2 + 6x + 5$.

```
>> delta = solveEquation(1, 6, 5) % Obtenir la valeur de delta seulement

delta =

    16

>> [delta, X1] = solveEquation(1, 6, 5) % Obtenir la valeur de delta seulement et
la première solution seulement

delta =

    16


X1 =

    -5

>> [delta, X1, X2] = solveEquation(1, 6, 5) % Obtenir toutes les sorties de la
fonction

delta =

    16


X1 =

    -5


X2 =

    -1
```

Complément : Fonctions plus dynamiques

Il est également possible d'appeler une fonction donnée avec moins d'entrées que le nombre indiqué pour sa définition (il faudra bien sur que le code de la fonction soit écrit de sorte qu'il prévoit cette éventualité). Pour cela il est possible d'utiliser la commande `nargin` qui retourne le nombre de variables d'entrée utilisées lors de l'appel et de la fonction puis écrire le code selon ce nombre.

Exemple : Fonctions dynamiques

Dans cet exemple, nous allons changer le code de la fonction précédente (permettant de résoudre une équation de deuxième degré) pour qu'il prenne en charge aussi les équations de premier degré. Dans le cas d'une équation de deuxième degré, la fonction s'attend à avoir trois arguments : `a`, `b` et `c`, alors que pour une équation linéaire de premier degré la fonction s'attend à recevoir deux arguments seulement (`a` et `b`) et l'équation est résolue en calculant la valeur de `-b / a`. Nous allons aussi supposer que la fonction ne doit pas retourner la valeur de `delta` (même si elle l'utilise comme une variable intermédiaire). Le code de cette fonction est le suivant.

```
% Fonction - solveEquation.m
function [X1, X2] = solveEquation(a, b, c)
if nargin == 3
    delta = b^2 - 4 * a * c;
    X1 = (-b - sqrt(delta)) / (2 * a);
    X2 = (-b + sqrt(delta)) / (2 * a);
elseif nargin == 2
    X1 = -b / a;
end
```

L'utilisation de la nouvelle version de la fonction `solveEquation` est illustrée ci-dessous

```
>> x = solveEquation(1, 6) % résoudre l'équation x + 6 = 0

x =

    -6

>> x = solveEquation(1, 6, 5) % résoudre l'équation x² + 6x + 5 = 0 (calculer une
solution seulement)

x =

    -5

>> [x y] = solveEquation(1, 6, 5) % résoudre l'équation x² + 6x + 5 = 0 (calculer
les deux solution)

x =

    -5


y =

    -1
```

Remarque : La commande nargout

Il est aussi possible d'utiliser la commande `nargout` pour personnaliser le comportement d'une fonction selon le nombre de paramètres de sortie demandés à l'appel de la fonction.

Complément : Fonctions anonymes

Une fonction anonyme est une fonction Matlab qui est définie directement, sans la création préalable d'un fichier spécifique. La syntaxe générale pour définir une fonction anonyme est :

```
nomFonction = @(entrées) expression
```

Généralement, on utilise les fonctions anonymes pour créer des raccourcis syntaxiques, qui consistent en une seule instruction Matlab.

Exemple : La fonction x^2

Nous allons ici définir la fonction `carre(x)` qui calcule et retourne `x²` sous forme d'une fonction anonyme

```
>> carre = @(x) x^2
>> carre(4)

ans =

    16

>> carre(3)

ans =

     9
```

4. Exercices

4.1. Exercice : Permutation circulaire

Question

[solution n°9 p.75*]*

Écrire un script Matlab qui permet de lire 3 variables `a`, `b` et `c`, puis effectuer une permutation circulaire de leurs valeurs (mettre la valeur de `a` dans `b`, la valeur de `b` dans `c` et la valeur de `c` dans `a`) et finalement afficher les nouvelles valeurs de chaque variable.

4.2. Exercice : if ... end

Question

[solution n°10 p.75*]*

Écrire un script Matlab qui permet de lire la note moyenne d'un étudiant et qui affiche "admis" si cette note est supérieure ou égale à 10, et "ajourné" sinon.

4.3. Exercice : Affichage des dix nombres

Question

[solution n°11 p.75*]*

Écrire un script Matlab qui lit un nombre entier positif, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur saisit le nombre 17, le programme affichera les nombres de 18 à 27.

Remarque : Écrire le script en utilisant la boucle `for` puis en utilisant la boucle `while`.

4.4. Exercice : Insertion

Question

[solution n°12 p.76*]*

Écrire une fonction permettant de prendre comme paramètre un vecteur `A = [`a_1`, `a_2`, ... , `a_n`]` ligne et une valeur `x` et rendre comme résultat le vecteur `[`a_1`, x, `a_2`, x, ... , `a_n`, x]`.

4.5. Exercice : Factorielle (partie 1)

Question

[solution n°13 p.76*]*

Écrire une fonction Matlab qui retourne la factorielle de son argument (pour un nombre `n`, la fonction doit calculer et retourner `n!`).

4.6. Exercice : Factorielle (partie 2)

Question

[solution n°14 p.76*]*

Écrire un script Matlab qui permet de lire un nombre, puis calcule et affiche le factorielle de ce dernier.

Indice :

utiliser la fonction définie dans l'exercice précédent.

4.7. Exercice : PGCD

Question

[solution n°15 p.76*]*

- Écrire une fonction Matlab qui permet de calculer le PGCD (plus grand diviseur commun) de deux nombres entiers positifs passés comme arguments.
- Utiliser la fonction définie pour écrire un script permettant de calculer le PGCD de deux valeurs saisies par l'utilisateur.

Indice :

Utiliser la règle récursive suivante :

- pgcd(a, 0) = a;
- si a >= b alors pgcd(a, b) = pgcd(a - b, b).

4.8. Exercice : Recherche dans un tableau

Question

[solution n°16 p.76*]*

Écrire une fonction Matlab `chercher` qui prend comme arguments un vecteur `v` et un nombre `x`, et qui retourne 1 si `x` appartient à `v`, et 0 sinon.

Indice :

Deux versions de cette fonction peuvent être implémentées, une à l'aide d'une boucle `for`, et d'un test `if` approprié, l'autre directement avec une comparaison globale `==`.