TP4 – LES STRUCTURES DE CONTRÔLE

1 – LES INSTRUCTIONS IF-ELSE

Une instruction **if** est une construction de code qui exécute des blocs de code seulement si certaines conditions sont vérifiées. Ces conditions sont représentées comme des expressions logiques. La construction de l'instruction **if** est comme suit :

if expression logique
Bloc de code
end

Le mot '**if**' est un mot-clé. Une instruction **if** est terminée par le mot clé **end**. Quand MATLAB voit une instruction **if**, il déterminera si l'expression logique associée est vraie. Si c'est le cas, alors le bloc de code sera exécuté par MATLAB. Si l'expression logique est fausse, alors le code ne sera pas exécuté.

Quand il y'a plusieurs cas à considérer, vous pouvez incluse les instructions **elseif** ; si vous voulez une condition qui couvre n'importe quel autre cas, alors vous pouvez utiliser une instruction **else** ; comme dans l'exemple suivant :

if expression logique P
bloc de code 1
elseif expression logique Q
bloc de code 2
elseif expression logique R
bloc de code 3
else
bloc de code 4
end

Dans cet exemple, MATLAB vérifiera d'abord si **P** est vraie. Si **P** est vraie, alors le bloc de code 1 sera exécuté, et alors l'instruction **if** se terminera. En d'autres termes, MATLAB ne vérifiera pas le reste des expressions une fois qu'il atteint une expression vraie. Cependant, si **P** est fausse, alors MATLAB vérifiera si **Q** est vraie. Si **Q** est vraie, alors le bloc 2 sera exécuté, et l'instruction **if** se terminera. Si **Q** est fausse, alors MATLAB vérifiera si **R** est vraie, et ainsi de suite. Si **P**, **Q** et **R** sont toutes fausses, alors le bloc de code 4 est exécuté. Vous pouvez avoir n'importe quel nombre d'instruction **elseif** (ou aucune) après l'instruction **if**. Vous pouvez aussi utiliser au plus une instruction **else** ou aucune. Les expressions logiques après **if** et **elseif** (telles que **P**, **Q** et **R**) seront référées comme des **expressions conditionnelles**.

Remarque 1 : Souvenez-vous que l'expression **a < x < b** est formée de deux expressions conditionnelles : **x > a ET x < b**. Si vous voulez tapez **a < x < b**, vous obtiendrez des résultats inattendus et indésirables.

Astuce 1 : L'indentation rendra votre code plus facile à lire. Pour indenter un bloc de code, il suffit de sélectionner tout votre code en appuyant sur **ctrl+a**, puis de l'indenter avec **ctrl+i**.

Il y'a plusieurs fonctions logiques qui sont conçues dans MATLAB pour vous aider à construire des instructions de branchement. Par exemple, vous pouvez demander si une variable a un certain type de données ou une valeur avec des fonctions comme **isa**, **isreal**, **isnan**, et **isinf**. Il y'a aussi des fonctions qui peuvent nous donner des informations logiques sur des tableaux comme **any**, qui retourne vrai s'il y'a au moins un élément vrai dans un tableau, et faux autrement, et **all**, qui retourne vrai seulement si tous les éléments d'un tableau sont vrais.
Des fois, vous voulez écrire une fonction et vérifier ses entrées pour s'assurer que cette fonction sera utilisée proprement. Par exemple, la fonction **monAdditionneur** du TP3 s'attend à recevoir des entrées de type **double**. Si l'utilisateur entre un **char** comme une des variables d'entrée, alors la fonction retourne une

erreur ou donne des résultats inattendus. Pour empêcher ceci, vous pouvez mettre un contrôle pour dire à l'utilisateur que la fonction n'a pas été utilisée proprement en utilisant la fonction **error**. Cette dernière arrête l'exécution de la fonction et donne une erreur avec le texte de la chaîne d'entrée. La fonction **error** prend des entrées de type **sprint**.

Exemple 1 : La fonction **monAdditionneur** sera modifiée pour générer un message d'erreur si l'utilisateur n'utilise pas des entrées de type **double**. Essayer la nouvelle fonction avec des entrées de type **char** pour montrer que la fonction fonctionne.

```
function [S] = monAdditionneur (a, b, c)
        % Vérification pour les entrées erronées
        if ~isa(a, 'double') || ~isa(b, 'double') || ~isa(c, 'double')
                error('Les entrées de la fonction doivent être de type double')
        end
        S = a + b + c ;
end
```

Ex1 : Ecrire une fonction nommée **estImpair** qui retourne **impair** si l'entrée est impaire et **pair** si l'entrée est paire. Utiliser la fonction **rem** de MATLAB.

Ex2 : Ecrire une fonction nommée **monCalcCerc** qui prend comme entrées **r** de type double et une chaîne **calc**. Vous pouvez supposer que **r** est positif, et que **calc** est soit la chaîne **surface**, soit la chaine **périmètre**. La fonction **monCalcCerc** doit calculer la surface d'un cercle de rayon **r** si la chaîne **calc** est surface, et le périmètre d'un cercle de rayon **r** si la chaîne **calc** est périmètre. Utiliser la fonction **strcmp** de MATLAB. La fonction doit être vectorisée pour r.

2 – L'INSTRUCTION SWITCH-CASE

L'instruction **switch-case** fournit un moyen de choisir un bloc de code à exécuter parmi plusieurs blocs possibles selon la valeur de l'expression. La structure de l'instruction est comme suit :

```
switch  expression
  case  valeur 1
        bloc de code 1
  case valeur 2
        bloc de code 2
  otherwise
        bloc de code 3
end
```

L'expression qui suit le mot-clé **switch** est un scalaire ou une chaîne de caractères. Si elle coïncide avec une valeur qui suit le mot-clé **case**, alors le bloc de code correspondant est exécuté (seulement un bloc de code, celui entre le **case** qui correspond et soit le **case**, soit le mot-clé **otherwise**, soit le mot-clé **end** qui suit, est exécuté).

- S'il y'a plus d'une correspondance, seulement le premier **case** correspondant est exécuté ;
- Si aucune correspondance n'est trouvée et l'instruction **otherwise** (qui est optionnelle) existe, alors le bloc de code entre **otherwise** et **end** est exécuté ;
- Si aucune correspondance n'est trouvée et l'instruction **otherwise** n'existe pas, alors aucun bloc n'est exécuté ;
- La valeur qui suit le **case** peut être sous la forme d'un tableau de cellule : {valeur1, valeur2, valeur3, …}. Dans ce cas, le bloc du **case** est exécuté si au moins une des valeurs correspond à la valeur de l'expression du **switch**.

Ex3 : Ecrire une fonction avec l'entête [f] = maCalculatrice (a, b, operation). L'argument d'entrée **operation** est un caractère soit '+', '-', '*', '/', ou '^', et **f** doit être calculée comme **a+b**, **a-b**, **a*b**, **a/b**, et **a^b** pour les valeurs respectives pour l'opération. Assurez-vous que votre fonction est vectorisée.

Maintenant, nous allons voir dans les sections suivantes deux types de boucle : la **boucle for** et la **boucle while**. Les boucles fournissent un mécanisme pour que le code effectue des tâches répétitives ; qui est l'itération. Les boucles sont importantes pour construire des solutions itératives aux problèmes.

3 – LA BOUCLE FOR

Une boucle **for** est une séquence d'instructions qui est répétée, ou itérée, pour chaque valeur du **vecteur de boucle**. La variable qui contient la valeur courante du vecteur de la boucle est appelée **variable de boucle**. Des fois, les boucles **for** sont référées comme des **boucles définies** parce qu'elles ont un début et une fin prédéfinis. La syntaxe du bloc d'une boucle **for** est comme suit :

```
for variable de boucle = vecteur de boucle
  bloc de code
end
```

Une boucle **for** affecte à la variable de boucle le premier élément du vecteur de boucle. Elle exécute chaque instruction dans le bloc de code. Ensuite, elle affecte à la variable de boucle l'élément suivant du vecteur de boucle et exécute le bloc de code de nouveau. Elle continue de cette façon jusqu'à ce qu'il n'y ait plus d'éléments du vecteur de boucle à affecter.

Ex4 : Etant donné un tableau A de nombres strictement positifs, additionnez tous les éléments de A. Additionnez maintenant uniquement les nombres pairs de A.

Exemple 2 : Ecrivons une fonction qui permet de chercher s'il y'a un 'e' dans une chaîne et retourne la valeur 1 si c'est le cas, sinon 0.

```
function [S] = yatilunE (chaine)
% S est 1 si et seulement s'il y'a un 'e' dans la chaîne d'entrée
% Commencez par supposer qu'il n'y a pas de 'e' dans la chaîne
S=0 ;
% Vérifier chaque lettre dans la chaîne, chaine(i), pour la valeur 'e'
% i es la i ème lettre dans chaîne
   for i = : length(chaine)
        if strcmp(chaine(i),'e) | | strcmp(chaine(i),'E')
                % Si la i ème lettre est 'e' ou 'E', alors il y'a au moins un 'e' dans la chaîne
                S=1 ;
                % On peut arrêter de chercher si on trouve un 'e' dans la chaîne
                break
        end
    end % end for i
end
```

Remarquez le mot-clé **break**. Si l'instruction **break** est exécutée, elle arrête immédiatement la boucle **for** la plus immédiate qui la contient ; c'est-à-dire, si elle est contenue dans une boucle **for** imbriquée, alors elle arrêtera seulement la boucle **for** la plus interne. Dans cet exemple, l'instruction **break** est exécutée si jamais on trouve un 'e'. Le code continuera à fonctionner encore proprement sans cette instruction ; mais puisque la tâche est de trouver s'il y'a un 'e' dans la chaîne, nous n'avons pas à continuer à chercher dans la chaîne dès qu'on trouve un 'e'. Les instructions **break** sont utilisées quand quelque chose arrive dans une boucle **for** qui arrêtera la boucle aussitôt.

Astuce 2 : Il est souvent utile de placer un commentaire avant la boucle **for** en précisant ce que l'index de la boucle représente et ce que la boucle fait.

Des fois, on a besoin de l'instruction **continue** pour éviter le reste du code dans l'itération courante d'une boucle **for**, et continuer sur l'élément suivant du vecteur de boucle.

Exemple 3 : Voici un script qui affiche les chiffres 0, 1, 2, 8, 9.

```
for  i = 0 :9
        if  i>2 && i<8          % si  2 < i < 8
                continue
        end
        sprintf('%d',i) ;
end
```

Ex5 : Soit une matrice A = $\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$ définie dans l'espace de travail courant. Utiliser une boucle imbriquée **for** pour sommer tous les éléments de A. Comment adapteriez-vous ce code pour prendre en charge une matrice A de taille arbitraire ?

Remarque 2 : Bien que cela soit possible, n'essayez pas de changer la variable de boucle à l'intérieur de la boucle for. Cela rendra votre code compliqué et il en résultera probablement des erreurs.

Ex6 : Trouver la somme des carrées des nombres de 1 à 10.

Ex7 : Etant donnée une matrice A à 3 colonnes et un nombre non spécifié de lignes, et où chaque ligne contient 3 nombres. Ecrire une fonction avec l'entête : [S] = monAdditionneurListe (A), où S est un vecteur colonne dans lequel chaque élément correspond à la somme des 3 nombres de chaque ligne de A. Votre fonction doit appeler la fonction monAdditionneur du TP 3 et utiliser une seule boucle **for**.

4 – LA BOUCLE WHILE

Une boucle **while** ou une boucle indéfinie est un ensemble d'instructions qui est répété aussi longtemps que l'expression logique associée est vraie. La syntaxe abstraite du bloc d'une boucle **while** est la suivante :

```
while   expression logique
        bloc de code
end
```

Quand MATLAB atteint un bloc d'une boucle **while**, il détermine d'abord si l'expression logique de la boucle **while** est vraie ou fausse. Si l'expression est vraie, alors le bloc de code sera exécuté. Si l'expression est fausse, alors la boucle **while** se terminera.

Exemple 4 : Une drôle de façon de tester la vitesse de votre ordinateur est de voir jusqu'à quand, il peut compter en une seconde. Noter que la fonction **tic** démarre un timer interne dans MATLAB. La fonction **toc** retourne le temps en fractions de seconde depuis le dernier **tic**. Ce petit test peut vous donner une idée de combien les ordinateurs sont rapides comparés à des humains.

```
N=0
tic
while  toc <= 1
        N=N+1
end
N
```

Astuce 3 : Si vous pensez que votre code est bloqué dans une boucle infinie, ou si vous êtes juste fatigué d'attendre qu'il termine son travail, vous pouvez appuyer sur **ctrl+c** pour forcer votre code à s'arrêter. Une erreur apparaîtra à la ligne qui était en train de s'exécuter quand **ctrl+c** a été appuyé.

Les command **break**, **error**, et **return** sont utiles quand elles opèrent à l'intérieur d'une boucle **for** ou **while** pour arrêter ou contrôler son exécution.

La commande **break** termine sans condition l'exécution d'une boucle et le programme continue avec la première instruction qui suit la commande **end**.

La commande **error**('texte') arrête l'exécution d'une boucle, affiche la chaîne texte sur l'écran, et transfère le contrôle au clavier.

La commande **return** produit une sortie inconditionnelle d'une boucle, ignorant les instructions à l'intérieur de la boucle.

EXERCICES PROPOSES

Exercice 1 :

Ecrire une fonction avec l'entête [nRacines, r] =mesNRacines(a, b, c) où a, b et c sont les coefficients d'une equation du second degré $Q(x) = ax^2 + bx + c$, avec a ≠ 0. nRacines est 2 si Q admet deux racines, 1 si Q admet une racine double, -2 si Q admet deux racines imaginaires, et r est un vecteur contenant les racines de Q.

Exercice 2 :

Ecrire une fonction avec l'entête [M] = monDamier(n), où M est une matrice n × n, ayant la forme suivante :

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Noter que l'élément supérieur le plus à gauche doit être 1. Supposer que n est un entier positif.

Exemple d'exécution :

>> [M] = monDamier(1)

M = 1

>> [M] = monDamier(2)

M = $\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$

>> [M] = monDamier(3)

M= $\begin{matrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{matrix}$

Exercice 3 :

Ecrire une fonction avec l'entête [M] = monMax(A), où M est la valeur maximale dans A. Ne pas utiliser la fonction « max( ) » construite dans MATLAB.

Exercice 4 :

Le paradoxe de Zeno peut être écrit en termes d'une somme infinie :

$$\sum_{n=1}^{\infty} \frac{1}{2^n} = 1$$

Pour voir comment cette série converge rapidement à 1, calculer la somme pour :

a) N= 5 ; b) n = 10 ; c) n=40.

Pour chaque partie, créer un vecteur n dans lequel le premier élément est 1, l'incrément est 1, et le dernier terme est 5, 10 ou 40. Ensuite, les calculs élément par élément pour créer un vecteur dans lequel les éléments sont $\frac{1}{2^n}$. Finalement, utiliser la fonction sum construite dans MATLAB pour additionner les termes de la série. Comparer les valeurs obtenues dans les parties a, b, et c avec la valeur 1.

Exercice 5 :

Ecrire une fonction avec l'entête [P] = monPoly (A, x) qui calcule la valeur numérique en x d'un polynôme de degré n et où A est un vecteur qui contient les coefficients du polynôme dans l'ordre décroissant. On utilisera l'algorithme de Hörner, qui évite les exponentiations.

Soit le polynôme : $P(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2 x^2 + a_1 x^1 + a_0$, On peut l'écrire sous la forme :$P(x) = x\,(\,x\,(\,x\,(\ldots(a_n x + a_{n-1}) + a_{n-2}) + \ldots) + a_2) + a_1) + a_0$, (On calculera d'abord $a_n x + a_{n-1}$, puis x $(a_n x + a_{n-1}) + a_{n-2}$, et ainsi de suite.

Exercice 6 :

Ecrire une fonction avec l'entête [S] = maFibonacci(n) qui détermine la n-ième valeur de U_n de la suite de Fibonacci définie comme suit :

$U_1 = 1$

$U_2 = 1$

$U_n = U_{n-1} + Un\text{-}2$, pour n > 2.