TP1 : Programmes simples avec Matlab

Dans ce TP, nous verrons les mot clef **if** permettant de faire exécuter du code plus complexe à Matlab, et obtenir ainsi des programmes plus intéressants. Nous verrons aussi d'autres fonctionnalités de Matlab afin de pouvoir écrire du code plus efficace.

Vous devez avoir terminé le TP0 avant de commencer ce TP. De plus, l'utilisation explicite des boucles est interdite (sauf mention contraire) dans ce TP.

1 Encore des matrices...

1.1 L'opérateur " : "

L'opérateur **:** permet, comme vous l'avez vu au précédent TP, de récupérer des ensembles de lignes et/ou colonnes d'une matrice, et former une sous-matrice. Pour rappel :

```
>> disp(a)
   6 6 2 6 4 2
   7 2 0 3 3 4
   7 6 1 9 7 4
   4 0 7 0 7 6

>> b = a(2:3, 4:6);
>> disp(b)
   3 3 4
   9 7 4
```

On peut aussi récupérer l'intégralité des éléments d'une matrice, sous forme d'un vecteur colonne (les éléments sont placés selon leur numéro d'ordre dans la matrice), en faisant :

```
>> c = b(:);
>> disp(c)
   3
   9
   3
   7
   4
   4
```

Cependant, l'opérateur **:** peut être utilisé en dehors de toute matrice. Regardez par exemple ce code :

```
>> d = 2:8;
>> disp(d)
    2 3 4 5 6 7 8
```

L'opérateur **:** permet en réalité de générer un vecteur ligne de tous les entiers entre le nombre donné à gauche (2), et le nombre donné à droite inclus (8).

Pour ceux qui connaissent Python : En Python, un opérateur similaire existe, qui s'appelle **range**. Cependant, en Python, le nombre donné en second paramètre de **range** (ici, ce serait le 8) est exclu de la liste des nombres obtenue.

On peut rajouter un troisième paramètre pour contrôler le pas d'incrémentation entre les éléments générés. Par exemple, pour obtenir tous les entiers allant de 3 à 30 de 5 en 5, on ferait :

```
>> d = 3:5:30;
>> disp(d)
    3 8 13 18 23 28
```

Ce nouveau paramètre, qui vient s'intercaler entre le nombre contrôlant le début de la liste et le nombre contrôlant la fin de la liste permet de choisir le pas entre chaque élément de la liste résultante. Si aucun pas n'est précisé, alors un pas de 1 est utilisé par défaut.

Dans tous ces exemples, des nombres entiers étaient utilisés ; cependant, il est tout à fait possible d'utiliser des nombres réels avec l'opérateur **:**.

Questions

Générez un vecteur ligne contenant tous les nombres réels entre 0 et 1 avec au plus deux chiffres significatifs après la virgule.

1.2 Une nouvelle façon de lire les éléments d'une matrice

Il est maintenant possible de comprendre ce que fait un tel code :

```
>> b = a(2:3, 5);
```

L'utilisation du symbole **:** génère un vecteur d'indices qui est utilisé pour indiquer toutes les lignes que l'on souhaite récupérer dans la matrice **b**. On peut donc utiliser un vecteur ligne afin de récupérer les colonnes et/ou lignes de son choix dans une matrice :

```
>> disp(a)
   6 6 2 6 4 2
   7 2 0 3 3 4
   7 6 1 9 7 4
   4 0 7 0 7 6

>> f = a(2:3, [1 3 4 6]);
>> disp(f)
    7 0 3 4
    7 1 9 4
```

On peut aussi utiliser, comme vu précédemment, un troisième paramètre afin d'appliquer un pas itératif différent de 1 à l'opérateur **:** :

```
>> g = a(:, 1:2:6);
>> disp(g)
    6 2 4
    7 0 3
    7 1 7
    4 7 7
```

Si l'on demande, par erreur, des indices de lignes ou de colonnes non entiers, Matlab nous le signale par une erreur :

```
>> g = a(:, 1:0.1:6);
Warning: Integer operands are required for colon operator when used as index
```

Questions

1. Ecrivez un script qui demande à l'utilisateur deux entiers **n** et **m**, et génère une matrice **M** de nombres aléatoires entre 0 et 100 de taille **n** $\times$ **m**. A la suite de votre script, générez une sous matrice avec uniquement les éléments de **M** à un indice de ligne et de colonne pair. Votre code doit fonctionner quel que soit la taille de M.
2. Ecrivez un script qui demande à l'utilisateur un entier **n** et génère un vecteur ligne **v** de taille **n** de nombres aléatoires entre 5 et 10. Ensuite, générez un vecteur **w** qui est le miroir de **v** (le premier élément de **v** est à la dernière place de **w**, le seconde élément de **v** est à l'avant dernière place de **w**, etc) sans utiliser le mot clef **flip**.
3. Ecrivez un script qui génère un vecteur ligne **v** de nombres aléatoires entre 0 et 10, possédant **7n** éléments. Puis, calculez dans un vecteur ligne **s** la somme partielle, par paquets de 7 éléments, du vecteur **v**. Le premier élément de **s** est donc la somme des 7 premiers éléments

de **v**, le second élément de **s** est la somme des sept éléments suivants de **v**, etc. Par exemple :

```
>> disp(v)
    3 1 4 1 3 3 4 5 3 1 1 1 4 1

>> disp(s)
    19 16
```

*Les fonctions **reshape** et **sum** pourraient vous être utiles.*

2 Le test conditionnel if

Le test if

Le mot clef **if** permet d'exécuter du code seulement si une condition est vraie. Regardez ce code :

```
x=-2;
y=0;
if x<0
    disp('x est negatif')
    y=-1;
end
disp(y)
disp('FIN')
```

A l'aide du debugger, explorez ce que fait ce code. Quelle différence y a-t-il si **x** est plus petit que zéro ou plus grand ? Les lignes 4 et 5 du code est-elle toujours exécutée ? Et les lignes 7 et 8 ? A quoi sert le mot clef **end** ?

Le test if/else

Le mot clef **else**, qui doit impérativement suivre un **if**, permet aussi d'exécuter du code si la condition du if n'a pas été respectée :

```
x=-2;
y=0;
if x<0
    disp('x est negatif')
    y=-1;
else
    disp('x est positif')
    y=1;
end
disp('FIN')
```

A l'aide du debugger, explorez ce que fait ce code. Que fait le mot clef **else** ici ? Que se passe-t-il si **x** est positif ? Et si **x** est négatif ? Quelles sont les lignes de code qui sont toujours exécutées ?

Le test if/elseif/else

Enfin, il est possible d'utiliser le mot clef **elseif** qui permet d'exécuter du code si la condition du **if** n'a pas été respectée, en y ajoutant une condition :

```
x=-2;
y=0;
if x<0
    disp('x est negatif')
    y=-1;
elseif x==0
    disp('x est nul')
    y=0;
else
    disp('x est positif')
```

```
    y=1;
end
disp('FIN')
```

A l'aide du debugger, explorez ce que fait ce code. Que fait le mot clef **elseif** ici ? Que se passe-t-il si **x** est positif, négatif ou nul ? Quelles sont les lignes de code qui sont toujours exécutées ?

Questions

1. Ecrivez un script qui demande un nombre à l'utilisateur et affiche sa valeur absolue.
2. Ecrivez un script qui demande un nombre **x** à l'utilisateur entre 0 et 1. Si l'utilisateur se trompe et que **x** est négatif, on remplace sa valeur par zéro, et s'il est supérieur à 1, on remplace sa valeur par 1.
3. Ecrivez un script qui demande deux nombres à l'utilisateur, et affiche le plus grand des deux.

Les règles du if

Voici quelques règles pour écrire ce que l'on appelle un **bloc if** :
— En tout premier, vient un seul mot clef **if**, avec une condition.
— Ensuite, zéro, un ou plusieurs mot clefs **elseif** suivent, chacun avec une condition.
— Ensuite, zéro ou un mot clef **else** suit, sans aucune condition marquée après.
— Enfin, le mot clef **end** marque la fin du bloc **if**.

Dans le cas où Matlab rencontre un bloc **if** correctement écrit, il agit ainsi :
— Il évalue la condition du **if** et, si elle est vraie, il exécute le code après le **if** jusqu'à rencontrer le mot clef **end**, **elseif** ou **else**.
— Sinon, il évalue la condition du **elseif** (s'il y en a un) suivant le **if**, et si elle est vraie, exécute le code après le **elseif** jusqu'à rencontrer le mot clef **end**, **elseif** ou **else**.
— Il recommence l'étape précédente tant qu'il existe des **elseif** à évaluer.
— Enfin, s'il rencontre le mot clef **else**, il exécute le code situé après jusqu'à renontrer le mot clef **end**.
— A chaque étape, après l'exécution d'un code après un **if**, **elseif** ou **else**, le programme saute jusqu'au mot clef **end** et reprend ici son exécution.

Si un bloc **if** ne possède pas le mot clef **else**, alors il est tout à fait possible qu'aucune ligne de code de ce bloc ne soit exécutée :

```
x=-2;
y=0;
if x<0
    disp('x est negatif')
    y=-1;
elseif x==0
    disp('x est nul')
    y=0;
end
disp('FIN')
```

Que se passe-t-il ici si **x** vaut 3 ?

Il est tout à fait possible, à l'intérieur d'un code après un **if, elseif** ou **else**, de placer de nouveau un bloc **if** :

```
x=-2;

if x>0
    if (x<10)
        disp('x est positif et petit')
    else
        disp('x est positif et grand')
    end
elseif x<0
    if (x<-10)
        disp('x est negatif et petit')
```

```
    else
        disp('x est negatif et grand')
    end
else
    disp('x est nul')
end

disp('FIN')
```

Utilisez le debugger pour comprendre ce qu'il se passe si **x** vaut -20, -5, 0, 5 ou 20.

Les conditions après un if

Il est possible d'écrire beaucoup de différents test conditionnels dans un bloc **if** :

Test Matlab	Effet	Exemple
$>, >=, <, <=$	Teste si la variable située à gauche du symbole est supérieure, supérieure ou égale, inférieure ou inférieure ou égale à celle située à droite	if $x <= y$
$==, \sim=$	Teste si les deux variables situées de part et d'autres du symbole sont égales ou différentes	if $x \sim= y$
$\&\&, \|\|$	Permet de combiner deux conditions avec un **et** ou un **ou**	if $x > y \ \|\| \ x < 3$
$\sim$	Permet d'inverser la condition donnée ensuite	if $\sim (x > 3 \ \&\& \ x < 4)$

Le symbole | (appelé pipe en anglais) peut-être écrit sur un clavier azerty en appuyant sur la touche ALT (à droite de la part espace) et la touche 6 située au-dessus des lettres.

Le symbole $\sim$ (appelé tilde) peut-être écrit sur un clavier azerty en appuyant sur la touche ALT (à droite de la part espace) et la touche 2 située au-dessus des lettres.

Questions

1. A l'aide d'un seul bloc **if**, ré-écrivez le dernier code qui vous a été donné dans cet énoncé.
2. Ecrivez un script qui demande un nombre à l'utilisateur et affiche BINGO si ce nombre est entre 3 et 4.
3. Ecrivez un script qui demande un nombre à l'utilisateur et affiche BINGO si ce nombre est inférieur à 0 ou supérieur à 10. Attention, vous n'avez pas le droit d'utiliser le symbole | dans votre code.
4. Voici un script pour tester si **x** vaut 2 ou non :

   ```
   x = input('Il faut deviner la bonne valeur. Entrez un nombre : ");

   if x=3
       disp('Bravo')
   else
       disp('Perdu')
   end
   ```

   Avant de tester ce code sous Matlab, fonctionne-t-il d'après vous ? Testez-le ensuite sous Matlab et voyez ce qu'il en est.
5. Voici un code pour trouver le plus grand élément entre trois valeurs :

   ```
   x=2;
   y=3;
   z=4;

   if x>y && x>z
       max=x;
   elseif y>x && y>z
       max=y;
   ```

```
else
   max=z;
end

disp('La plus grande des valeurs est')
disp(max)
```

Code fonctionne-t-il pour les valeurs de **x, y** et **z** données ? Et si on a :

```
x=2;
y=2;
z=4;
```

Et si on a :

```
x=4;
y=4;
z=2;
```

N'hésitez pas à utiliser le debugger Matlab pour comprendre ce qu'il se passe.

3 Le test conditionnel sur une matrice

Il est possible en Matlab de faire un test conditionnel sur chaque élément d'une matrice, à l'aide de deux méthodes distinctes.

La fonction find

La fonction **find** permet de trouver les indices (sous forme de numéro d'ordre, et non de coordonnées) de tous les éléments d'une matrice. Par exemple :

```
>> disp(A)
         7 5 8 8 3
         7 1 8 4 7
         1 2 1 6 6
         7 4 8 1 8

>> E = find(A > 6);
>> disp(E)
    1
    2
    4
    9
   10
   12
   13
   18
   20
```

Ici, on récupère dans le vecteur colonne **E** les positions des éléments de **A** (sous forme de numéro d'ordre, et non pas sous forme de coordonnées) qui sont supérieurs à 6. On rappelle que les éléments d'une matrice sont tous numérotés par un numéro d'ordre : le premier élément est celui de coordonnées (1,1), puis vient celui de coordonnées (2,1), puis (3,1), etc... Vérifiez bien que le vecteur **E** donné ici est correct.

Pour afficher les éléments de **A** qui correspondent au critère donné, on fera :

```
>> disp(A(E))
         7
         7
         7
         8
         8
```

```
         8
         8
         7
         8
```

Enfin, si on souhaite faire une opération uniquement sur ces éléments (par exemple, leur ajouter un), on fera :

```
>> A(E) = A(E) + 1;
>> disp(A)
         8 5 9 9 3
         8 1 9 4 8
         1 2 1 6 6
         8 4 9 1 9
```

Questions

1. Etant donné une matrice de nombres aléatoires entre -1 et 1 de 5 lignes et 7 colonnes, affichez combien d'éléments supérieurs ou égaux à 0 elle possède.
2. Réalisez, sur la matrice précédente, l'opération de valeur absolue sans utiliser la fonction **abs**.

Création d'une matrice de booléens

En réalité, la commande **find** prend en paramètre une matrice, et renvoie la position de tous les éléments non nuls de la matrice. Cependant, l'écriture $A > 6$ génère une matrice de booléens qui est ensuite scannée par **find** :

```
>> disp(A)
    7 5 8 8 3
    7 1 8 4 7
    1 2 1 6 6
    7 4 8 1 8

>> D = A>6;
>> disp(D)
    1 0 1 1 0
    1 0 1 0 1
    0 0 0 0 0
    1 0 1 0 1

>> E = find(D);
>> disp(E);
    1
    2
    4
    9
   10
   12
   13
   18
   20
```

Ici, **D** est une matrice de booléens, c'est à dire de valeurs dans $\{0; 1\}$. Si un élément de **A** est plus grand que 6, alors il vaudra 1 dans **D**, et 0 sinon. On peut utiliser cette matrice de booléens pour faire des opérations sur **A**. Par exemple, pour ajouter 2 à tous les éléments de **A** qui sont plus grands que 6, on fera :

```
>> A = A + D;
>> disp(A)
    8 5 9 9 3
    8 1 9 4 8
```

```
    1 2 1 6 6
    8 4 9 1 9
```

On peut aussi faire :

```
>> disp(A)
    7 5 8 8 3
    7 1 8 4 7
    1 2 1 6 6
    7 4 8 1 8

>> D = A>6;
>> A(D) = A(D) + 1;
>> disp(A)
    8 5 9 9 3
    8 1 9 4 8
    1 2 1 6 6
    8 4 9 1 9
```

Questions

1. Etant donné une matrice de nombres aléatoires entre -1 et 1 de 5 lignes et 7 colonnes, affichez combien d'éléments supérieurs ou égaux à 0 elle possède sans utiliser **find**.
2. Réalisez, sur la matrice précédente, l'opération de valeur absolue sans utiliser la fonction **abs** ou **find**.

Quelle méthode utiliser ?

On préférera utiliser la fonction **find** lorsque l'on aura besoin de récupérer la position des éléments d'une matrice qui respectent une certaine condition. Si l'on n'a pas besoin de cette information, et que l'on souhaite faire une opération sur les éléments d'une matrice qui respectent une certaine condition, on utilisera les matrices de booléens.

Cependant, la façon de faire ces opérations peut avoir un impact sur le temps de calcul. Regardez ce code :

```
n=3000;
A = rand(n,n);
B=A;
D = A>0.5;

tic
A(D) = cos(A(D)).^2;
toc;

tic
B= D.*cos(B).^2 + B.*(~D);
toc;

G = find(B~=A);
```

Il propose deux façons de faire la même opération : remplacer tous les éléments supérieurs à 0.5 d'une matrice par leur cosinus au carré. La première méthode utilise la matrice de booléen pour indexer les éléments à modifier, tandis que la seconde méthode fait l'opération sur toute la matrice et ne conserve que les résultats qui nous intéressent. Les mots clefs **tic** et **toc** permettent de démarrer et arrêter un chronomètre, afin de mesurer quel est le code le plus rapide.

Questions

1. Exécutez ce code, et augmentez petit à petit (par pas de 1000) la valeur de **n** afin que le temps d'exécution des deux codes soit de l'ordre de la seconde.

2. Quel code est le plus rapide : celui où l'on ne fait l'opération que sur les éléments qui nous intéressent, ou celui où l'on fait l'opération sur toute la matrice puis l'on conserve uniquement les éléments d'intérêt ?
3. Que se passe-t-il si vous modifiez la condition à la quatrième ligne pour ne s'intéresser qu'aux nombres plus grands que 0.3 ? Et si vous modifiez la condition pour ne s'intéresser qu'aux nombres plus grands que 0.9 ?

L'indexage des éléments à modifier est en général plus lent que le fait de réaliser l'opération sur toute la matrice et n'en conserver que certains éléments. Cependant, si le nombre d'éléments concernés par l'opération à réaliser devient petit, l'indexage est plus rapide.

Attention lorsque vous faîtes des mesures de temps de calcul, la méthodologie à utiliser doit être sans faille. Par exemple, dans ce code, quelle serait d'après vous la méthode la plus rapide :

```
n=3000;
A = rand(n,n)*2-1;


tic
D = A<0;
A(D) = -A(D);
toc;


tic
A(A<0) = -A(A<0);
toc;
```

Dans les deux cas, on calcule la valeur absolue de la matrice **A**. La première méthode utilise la matrice **D** pour conserver les éléments de **A** qui sont négatifs, et inverser leur signe. La seconde méthode fait la même chose, à part que l'on calcule deux fois les éléments de **A** qui sont négatifs.

Questions

1. La première méthode consistant à ne calculer qu'une seule fois les éléments de **A** qui sont négatifs, et les stocker dans **D**, devrait être logiquement la plus rapide. Pouvez-vous expliquer pourquoi elle est beaucoup plus lente ?
2. Une fois le bug de méthodologie corrigé dans le code précédent, quelle méthode est la plus rapide ? Pouvez-vous l'expliquer ?
3. Et dans ce cas, quelle méthode est la plus rapide ?

```
n=3000;
A = rand(n,n)*2-1;
B=A;


tic
D = A<0;
A(D) = A(D)+A(D)+A(D)+A(D);
toc;


tic
B(B<0) = B(B<0)+B(B<0)+B(B<0)+B(B<0);
toc;
```

4 Quelques exercices pour finir

Afin de terminer cette séance, et s'assurer que vous avez bien compris les différents concepts abordés, voici quelques exercices **qui devront être réalisés sans utiliser de boucles** :

1. Ecrivez un script qui demande une hauteur et une largeur à l'utilisateur, et génère une matrice aléatoire **A** de cette taille. Ensuite, construisez une matrice **M** qui est la plus grande matrice carrée extraite des éléments en haut à droite de **A**. Par exemple, on doit avoir

```
>> disp(A)
         9 8 5 5 1
         7 5 2 4 0
        10 4 8 9 9

>> disp(M)
         5 5 1
         2 4 0
         8 9 9
```

2. Etant donné une matrice **A** de taille 3000 par 3000, générée avec des nombres aléatoires entre 0 et 1, calculez le pourcentage d'éléments de **A** qui sont supérieurs ou égaux à 0.5. Ces résultats sont-ils en accord avec le fait que **rand** réaliser une distribution uniforme de nombres entre 0 et 1 ?
3. Ecrivez un script qui, étant donné un nombre **n** donné par l'utilisateur, génère la matrice identité, sans utiliser les fonctions **eye** ou **diag**.
4. Ecrivez un script qui demande une hauteur et une largeur à l'utilisateur, et génère une matrice aléatoire **A** de cette taille. Ensuite, construisez la matrice **B** qui est le miroir de **A** sans utiliser le mot clef **flip**. Par exemple, on doit avoir

```
>> disp(A)
   3 3 3 6 8 1
   9 5 4 6 3 6
   4 8 6 2 5 2
   2 6 7 8 3 6

>> disp(B)
   1 8 6 3 3 3
   6 3 6 4 5 9
   2 5 2 6 8 4
   6 3 8 7 6 2
```

5. La suite $u_n = (\frac{n-3}{n+3})^n$ converge vers e^{-6}. Ecrivez un programme calculant les mille premiers termes de cette suite, et regardez si cette convergence est vérifiée.
6. Voici un morceau de code permettant de générer un vecteur ligne de **n** entiers entre 0 et **m**.

```
n=1000;
m=10;
v = double(int8(rand(1,n)*m));
```

   On souhaite faire l'histogramme de ce vecteur ligne, c'est à dire compter combien de fois chaque valeur entre 0 et **m** apparaît dans le vecteur ligne. Votre code doit fonctionner quelles que soient les valeur de **n** et **m**, sans utiliser la fonction **hist**. Par exemple, on devrait avoir :

```
>> disp(v)
  Columns 1 through 16

     6 3 6 8 9 5 3 0 8 9 0 7 7 5 3 8

  Columns 17 through 20

     9 9 7 4

>> disp(histogram)
     2
     0
     0
     3
```

```
1
2
2
3
3
4
0
```

Les valeurs obtenues dans **histogram** montrent qu'il y a deux fois 0, jamais 1 ou 2, 3 fois 3, 1 fois 4, etc... dans le vecteur **v**. A la fin, si vous augmentez la valeur de **n**, constatez-vous que chaque valeur entre 0 et 10 apparaît de façon uniforme (le même nombre de fois) ?

*Indice : La fonction **repmat** pourrait vous être utile.*

7. Ecrivez un script qui génère une matrice carrée avec des nombres aléatoires entre 0 et 1, et force à 0 tous les nombres sous la diagonale de la matrice, sans utiliser les fonctions **trilu** ou **trill**.