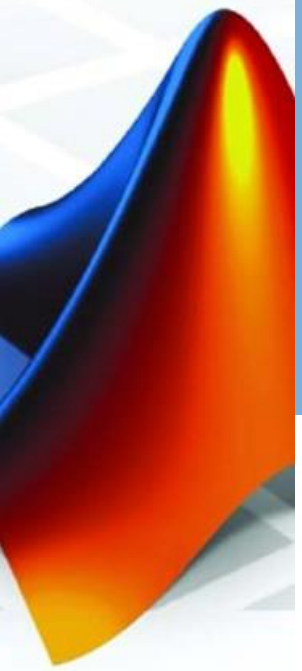


Introduction à Matlab

Ecole Sup Galilée - Cursus Ingénieur - 1ère année

2016-2017

John Chaussard
LAGA – Université Paris 13
chaussard@math.univ-paris13.fr

La programmation

Qu'est ce que la programmation

Un ordinateur ne sait faire que certaines opérations très basiques :

- Additionner, soustraire, multiplier, diviser des valeurs numériques
- Déplacer des données d'un endroit de sa mémoire à un autre endroit
- Sauter d'une ligne de code à une autre si une condition est vraie

La programmation consiste à **combiner ces instructions** qui agissent sur des données afin de réaliser une tâche spécifique à l'aide de l'ordinateur (tester si un nombre est premier, détecter des éléments dans une image, etc...)

Différents langages de programmation

Années 1940

- Assembleur

- Langage très proche des capacités de la machine : on peut y réaliser directement les opérations que peut exécuter la machine, et on doit penser son programme de façon à ce qu'il corresponde à ces instructions.

- Très fastidieux pour écrire de longs programmes, et pas très facile de relire un code

```
PlayerEndWorld:
               lda WorldEndTimer          ;check to see if world end timer expired
               bne EndExitOne             ;branch to leave if not
               ldy WorldNumber            ;check world number
               cpy #World8                ;if on world 8, player is done with game,
               bcs EndChkBButton          ;thus branch to read controller
               lda #$00
               sta AreaNumber             ;otherwise initialize area number used as offset
               sta LevelNumber            ;and level number control to start at area 1
               sta OperMode_Task          ;initialize secondary mode of operation
               inc WorldNumber            ;increment world number to move onto the next world
               jsr LoadAreaPointer        ;get area address offset for the next area
               inc FetchNewGameTimerFlag  ;set flag to load game timer from header
               lda #GameModeValue
               sta OperMode               ;set mode of operation to game mode
```

L'assembleur est toujours utilisé de nos jour . Ici, un exemple du code du jeu Super Mario World sur Super Nintendo.

Différents langages de programmation

Années 1950, 1960

- FORTRAN, LISP, COBOL, BASIC

- Langage plus moderne, avec un syntaxe proche des mathématiques modernes.

  Ex : En Assembleur, pour multiplier deux valeurs (par exemple 2 et 3), il faut charger chacune des valeurs dans une case mémoire, effectuer la multiplication des deux cases mémoires, et recopier le résultat sur le disque.

  En BASIC, il suffit d'écrire 2*3.

Différents langages de programmation

Années 1970

- C, Prolog, Pascal, SQL

- L'ère de la programmation structurée, avec une syntaxe plus rigoureuse qui permet de plus facilement relire un code et le comprendre.

Différents langages de programmation

Années 1980

- Matlab, C++, Objective-C, Perl

- Recherche de performance (programmes qui s'exécutent vite même si le code n'est pas optimisé) et de modularité (possibilité de facilement réutiliser des morceaux d'un programme dans un autre programme) afin de réduire les temps de développement.

- Débuts de la programmation orientée objet.

Différents langages de programmation

Années 1990

- Python, R, Java, PHP, JavaScript (rien à voir avec Java)
- Perfectionnement des concepts de l'orienté objet, débuts de l'ère Internet : les langages incorporent beaucoup de facilité de communication réseau.

Différents langages de programmation

Années 2000 et de nos jours

- D, Cuda, Go, Rust
- Recherche de sécurité dans les langages, afin d'éviter les failles dans les logiciels.
- Recherche de performance et facilité de programmation des ordinateurs avec des processeurs en parallèle (comme les cartes graphiques)
- Nouvel engouement pour d'anciens langages revenant à la mode, grâce à des nouveaux « frameworks » disponibles (Matlab, Python, JavaScript (HTML5), ou pour le développement sur plateforme mobile (Objective-C pour Iphone, Java pour Android)

Quelques définitions supplémentaires

Langage de programmation

- C'est une façon d'écrire des instructions qui seront ensuite « traduites » en opérations de base pour l'ordinateur

- Les instructions ont en général une tâche précise : elle forment ensemble un logiciel.

- Un langage de programmation est à l'informatique ce qu'une langue est à la littérature : c'est une façon de dire les choses afin de raconter une histoire.

Quelques définitions supplémentaires

Logiciel

- C'est une suite d'instructions écrites dans un langage de programmation qui permet de réaliser une ou plusieurs tâches (éditeur de texte, jeux vidéo, lecteur vidéo, navigateur internet, …)

- La plupart du temps, possède une interface graphique pour faciliter les interactions avec l'utilisateur

- Un logiciel est à l'informatique ce qu'un livre est à la littérature : c'est une histoire écrite dans une langue.

- Tout comme un livre peut être traduit dans différentes langues, un logiciel peut être traduit dans différents langages de programmation (ex : Angry Birds écrit en Java pour Android, et Objective-C pour IOS)

Quelques définitions supplémentaires

Framework

- Un framework est une collection de sous-programmes, écrits dans un langage spécifique, permettant de réaliser des tâches complexes plus simplement.

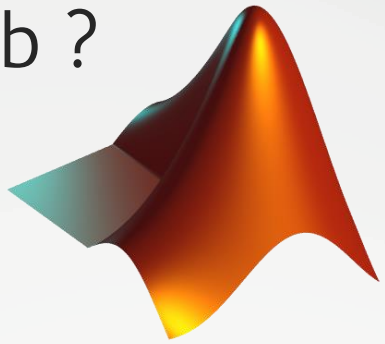
  Ex : En C++, pour multiplier deux matrices A et B, il faut le faire comme en maths – parcourir les lignes de l'une, les colonnes de l'autre, et faire des produits et des sommes.

  Avec le framework BOOST, on écrit prod(A,B)

- Un framework permet donc de réutiliser le travail effectué par d'autres pour son propre programme, et perdre moins de temps en évitant d'écrire ce que d'autres ont déjà écrit.

Présentation de Matlab

Qu'est-ce que Matlab ?



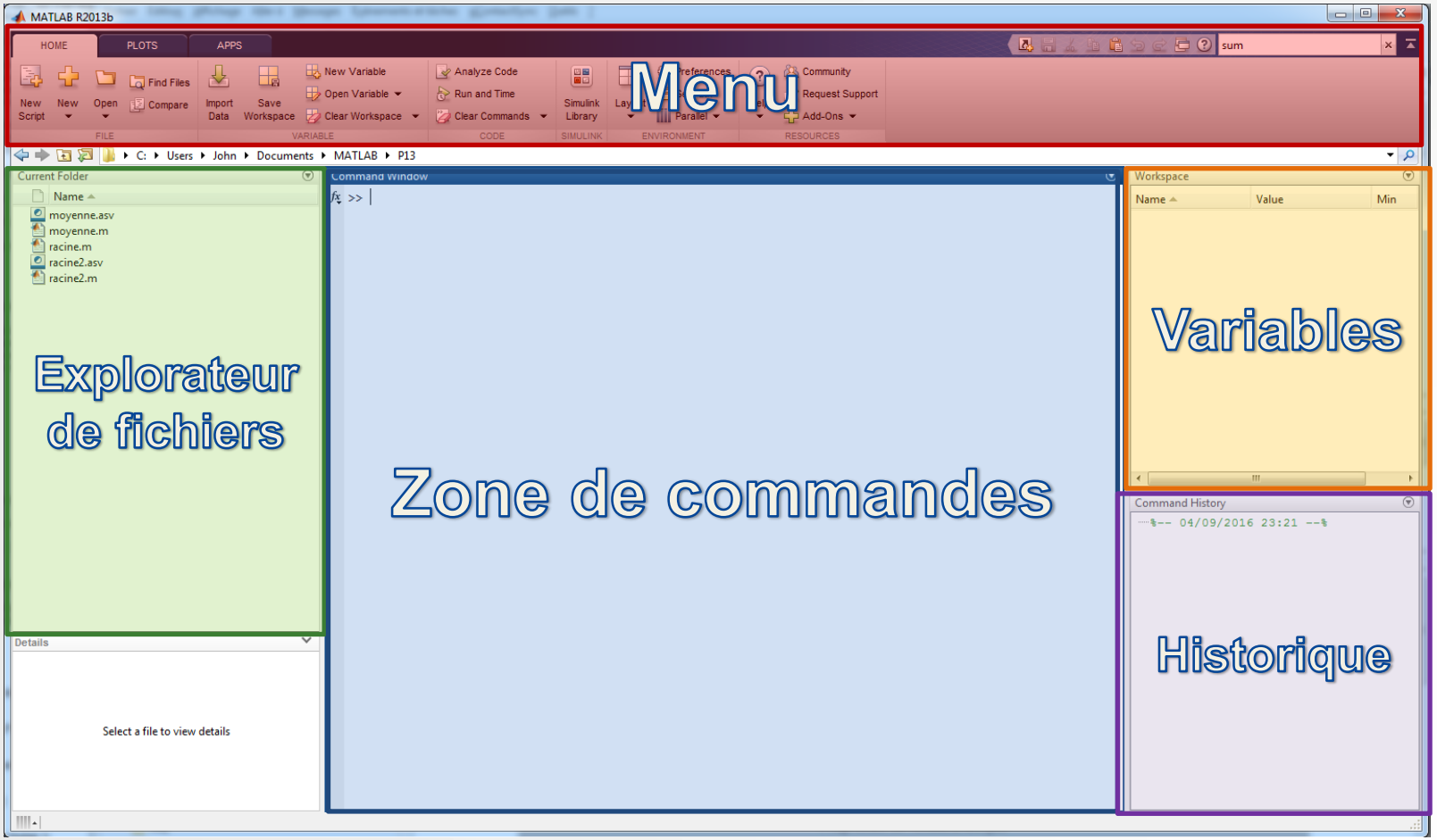
Matlab (MATrix LABoratory) est un langage de programmation orienté calcul scientifique, pensé pour rendre le calcul matriciel simple à programmer et efficace en temps.

Matlab est un logiciel (payant) proposant une interface graphique vers un éditeur de code en Matlab, et un outil de debugging pour exécuter des programmes en mode pas à pas.

Matlab propose différents frameworks (payants, appelés toolbox), proposant des fonctionnalités très avancées permettant de réaliser facilement des tâches complexes (ex : toolbox équation différentielle, toolbox aérospatial).

Présentation de l'interface

Comme vu en TP, l'interface de Matlab se compose de plusieurs zones



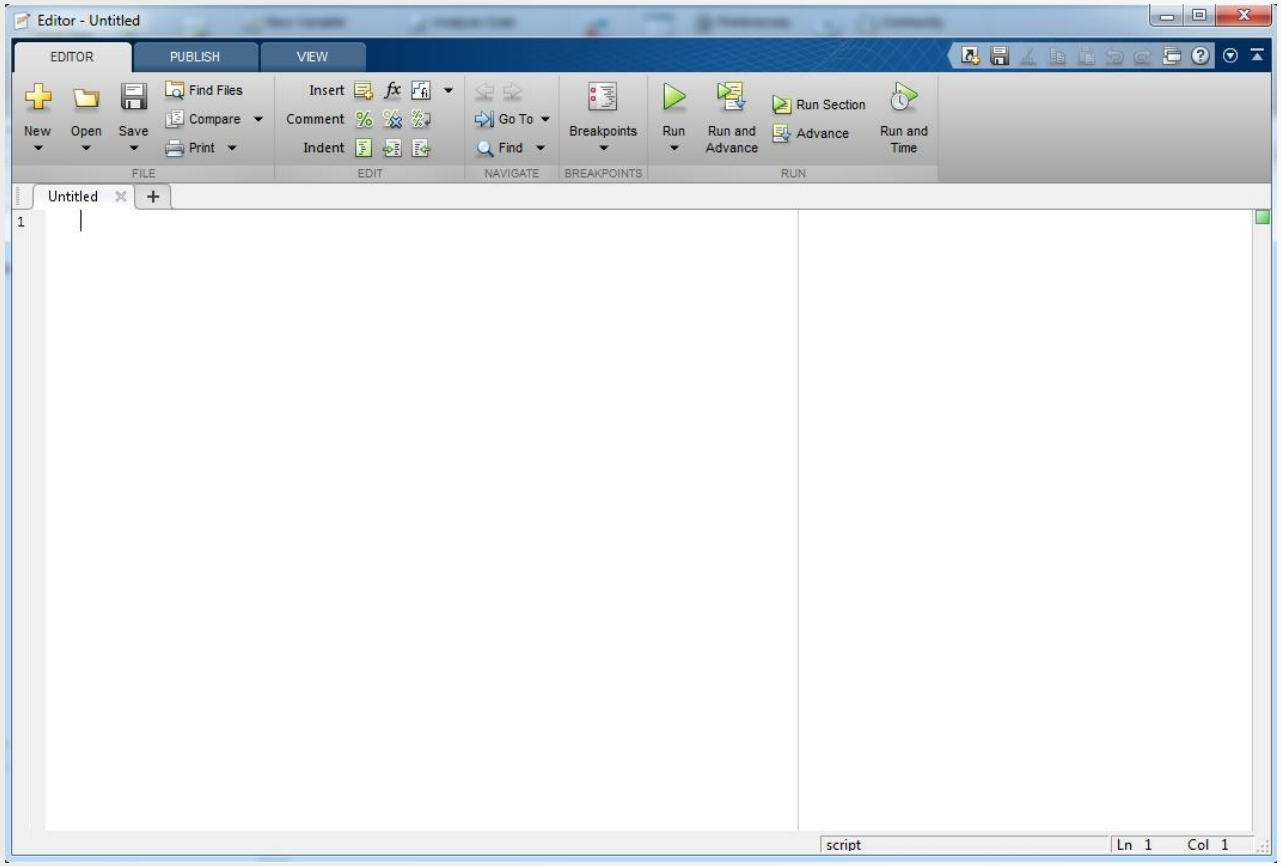
Présentation de l'interface

Chaque zone possède un objectif précis :

- Le menu regroupe des commandes de base de Matlab, comme enregistrer, afficher les préférences, etc…
- L'explorateur de fichiers permet de visualiser ses fichiers scripts et de les ouvrir pour les éditer.
- La zone de commande permet d'écrire des commandes et de visualiser leur résultat
- La zone des variables permet de visualiser toutes les variables en mémoire à l'instant présent (leur nom ainsi que visualiser leur contenu).
- L'historique permet de visualiser l'historique des commandes précédemment exécutées.

Présentation de l'interface

On peut écrire des commandes simples dans Matlab. Cependant, quand on souhaite écrire un programme complet, on utilise l'éditeur de script Matlab :



Matlab à la maison

Pour travailler sur Matlab chez soi, il existe plusieurs solutions :

- Acheter une version étudiante de Matlab (environ 60€)
- En trouver une sur Internet… 
- S'inscrire, sur Coursera, au cours **Introduction to Programming with MATLAB** (https://www.coursera.org/learn/matlab) qui est gratuit et permet d'obtenir un accès temporaire à Matlab Online (Matlab dans un navigateur internet)
- Se connecter par SSH aux ordinateurs de l'école : tutoriel complet sur le site johnchaussard.com, rubrique « Other »
- Installer le logiciel Octave (https://www.gnu.org/software/octave) qui est une sorte de clone de Matlab

La version de Matlab actuellement installée à l'école est la R2013b.

Les commandes de base en Matlab

Les opérations de base

Comme vous l'avez vu en TP, on peut saisir des commandes dans la zone de commande, et Matlab les exécute comme le ferait une calculatrice.

Par exemple, la commande

```
>> 3+6
```

affiche

```
ans =

     9
```

Ici, **ans** (pour answer) est une variable qui contient toujours le résultat de la dernière opération réalisée.

Les opérations de base

Voici une liste (incomplète) des opérations de base en Matlab

Symbole	Description	Exemple
+ - * /	Les opérations de base en mathématiques (addition, soustraction, multiplication et division)	7+9 3/4
pi	La constante Pi	pi/3
cos sin tan	Les fonctions trigonométriques usuelles	cos(3*pi/2)
log exp	Le logarithme népérien et l'exponentielle	exp(3)
sqrt	La racine carrée	sqrt(5)
^	La puissance	4^7

On peut combiner ces symboles en une seul commande. Les règles de priorité usuelles entre opérateurs sont alors appliquées.

Ex :

```
>> sqrt((3+4)*cos(5*pi/7))*exp(2^4)
```

Les nombres réels

Comme dans tous les langages de programmation, les nombres réels s'écrivent avec un point pour séparer la partie entière de la partie décimale, et non pas une virgule.

```
>> 2.7 * 3.9
ans =

   10.5300
```

Les variables (simples) en Matlab

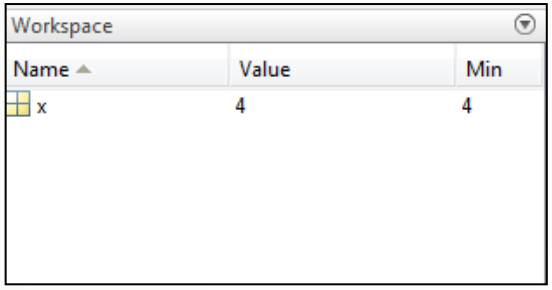
Comme dans la plupart des langages de programmation (comme le C), il est possible de stocker des valeurs numériques dans des variables.

Pour déclarer une variable et lui attribuer une valeur, il suffit d'utiliser le symbole égal :

```
>> x = 4
```

Cette commande créé une variable nommée x, et lui attribue la valeur 4.

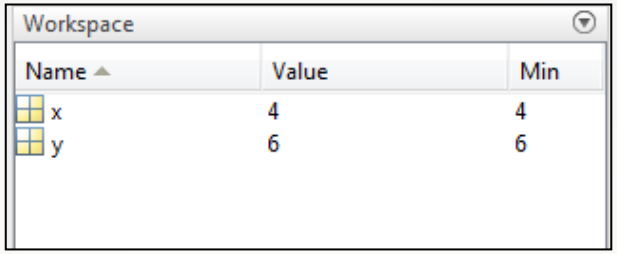
La variable apparait alors dans la zone des variables :



Les variables (simples) en Matlab

Il est possible d'utiliser des variables à la place de valeurs numériques dans les opérations :

```
>> y = x+2
```



Contrairement au langage C où il est nécessaire de déclarer une variable avant de lui attribuer une valeur, en Matlab, on attribue directement une valeur à la variable pour la créer.

Attention de ne pas confondre les deux syntaxes !

Les variables (simples) en Matlab

Pour afficher la valeur contenue dans une variable, on utilise le mot clef **disp**, ou bien on peut écrire directement le nom de la variable :

```
>> disp(y)
     6
```

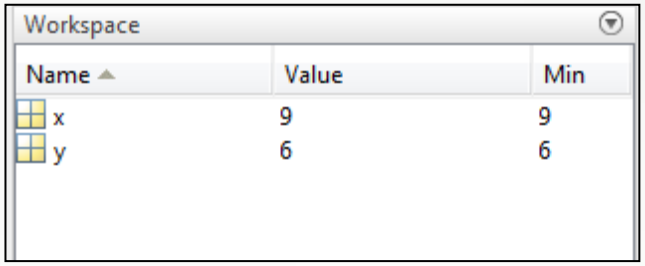
```
>> y
y =
     6
```

Lors du TP, on a parlé souvent de « mots clefs ». En réalité, le terme exact est « fonction » : on parlera désormais de la fonction **disp**, tout comme les fonctions **cos**, **sqrt**, etc…

Les variables (simples) en Matlab

Pour initialiser une variable avec une valeur entrée au clavier par l'utilisateur, on utilisera la fonction **input** :

```
>> x = input('Entrez une valeur : ');
Entrez une valeur : 9
```



Les matrices dans Matlab

Matlab et les matrices

Comme son nom l'indique, Matlab est un langage de programmation pensé pour le calcul matriciel, où il excelle en terme de performances.

Tout le but d'un programme en Matlab est de faire passer un calcul classique à travers un calcul matriciel pour que Matlab le réalise très rapidement.

Cela s'appelle la **vectorisation** (nous y reviendrons plus tard).

Nous allons voir maintenant comment Matlab gère les matrices.

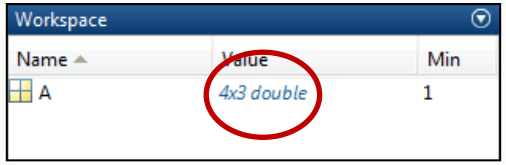
Déclarer des matrices dans Matlab

On distinguera plusieurs types de matrices dans Matlab. Tout d'abord, il y a les **matrices classiques**

- Pour déclarer une matrice dans une variable A, on énumère entre crochets ses éléments (séparés par des espaces), et on utilise un point virgule pour passer d'une ligne à l'autre :

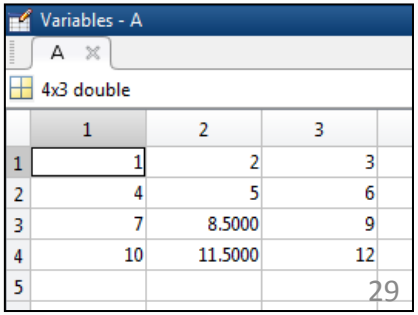
```
>> A = [1 2 3; 4 5 6; 7 8.5 9; 10 11.5 12]
```

- D'après le code ci-dessus, la matrice A possède quatre ligne et trois colonnes. On peut vérifier cela dans la zone des variables



- Si on double clique sur la variable dans la zone des variables, son contenu s'affiche.

  On peut aussi utiliser la fonction **disp** pour afficher une matrice.



Déclarer des matrices dans Matlab

A vous de jouer : écrivez cette matrice sous forme de tableau

```
>> B = [4.2 7.6 8.2; 4.1 0.5 0]
```

Solution :

4.2	7.6	8.2
4.1	0.5	0

Et cette matrice, que donne-t-elle ?

```
>> C = [5.7 0.2 6.2 5.1; 7.1 2.4 8.4; 1.2 0.4 8.4 6.4]
```

Solution : Une erreur de syntaxe car il n'y a pas le même nombre d'éléments à chaque ligne.

Déclarer des matrices dans Matlab

Il y a ensuite les matrices avec une seule ligne ou une seule colonne, que l'on appelle les **vecteurs**.

- Il y a les vecteurs ligne, constitués d'une seule ligne

```
>> D = [4 2 9 4 2]
```

D

4	2	9	4	2

- Il y a les vecteurs colonne, constitués d'une seule colonne

```
>> E = [3.5 ; 7; 9; 8; 0]
```

E

3,5
7
9
8
0

Déclarer des matrices dans Matlab

Si on déclare une matrice A d'une certaine taille

```
>> A = [1 2 3; 4 5 6; 7 8.5 9; 10 11.5 12]
```

Peut-on ensuite déclarer, dans A, une matrice d'une autre taille ?

```
>> A = [4.2 7.6 8.2; 4.1 0.5 0]
```

Réponse : Oui, cela ne pose aucun problème en Matlab. Il est même possible de déclarer dans A une simple valeur numérique

```
>> A = 3
```

Accéder aux éléments des matrices

Pour accéder aux éléments d'une matrice, on utilise la syntaxe suivante :

```
>> A(3,2)
```

Ici, on accède à l'élément de A qui est à la 3° ligne et 2° colonne.

A vous de jouer : que vaut y dans cet exemple ?

```
>> A = [5.7 0.2 6.2 5.1; 8.7 7.1 2.4 8.4; 1.2 0.4 8.2 6.4]

>> y = A(2,4)
```

Ici, y vaut 8.4.

Notez qu'en Matlab, on commence la numérotation des lignes/colonnes d'une matrice à 1. En C, elles commencent à 0 (ne confondez pas).

Accéder aux éléments des matrices

On peut aussi accéder aux éléments d'une matrice par un unique numéro qui est leur ordre dans la matrice. Le premier élément d'une matrice est celui à la 1° ligne et 1° colonne, le second élément est celui à la 2° ligne et 1° colonne, etc…

Si on a

```
>> A = [5.7 0.2 6.2 5.1; 8.7 7.1 2.4 8.4; 1.2 0.4 8.2 6.4]
```

Voici la matrice A avec les positions de ses éléments en rouge :

1 5.7	4 0.2	7 6.2	10 5.1
2 8.7	5 7.1	8 2.4	11 8.4
3 1.2	6 0.4	9 8.2	12 6.4

En général, on ne se sert pas de cette méthode. Mais certaines fonctions renvoient, comme résultat, les numéros d'ordre de certains éléments d'une matrice.

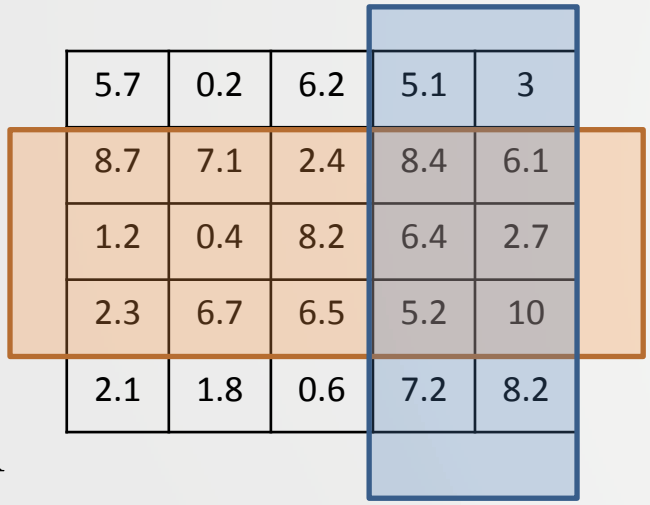
```
>> A(5)

ans =

    7.1000
```

Accéder aux éléments des matrices

On peut aussi facilement extraire des sous-matrices d'une matrice à l'aide de cette syntaxe :

```
>> B = A(2:4,4:5)
```

Ici, B sera la matrice constituée des éléments de A aux lignes 2 à 4 et aux colonnes 4 à 5.



A

8.4	6.1
6.4	2.7
5.2	10

B

Accéder aux éléments des matrices

On peut aussi récupérer l'intégralité des lignes ou des colonnes d'une matrice dans une sous matrice en laissant des « deux points » seuls :

```
>> B = A(2:4,:)
```

Ici, B sera la matrice constituée des éléments de A aux ligne 2 à 4.

5.7	0.2	6.2	5.1	3
8.7	7.1	2.4	8.4	6.1
1.2	0.4	8.2	6.4	2.7
2.3	6.7	6.5	5.2	10
2.1	1.8	0.6	7.2	8.2

A

8.7	7.1	2.4	8.4	6.1
1.2	0.4	8.2	6.4	2.7
2.3	6.7	6.5	5.2	10

B

Taille d'une matrice

On peut obtenir, avec la fonction **size**, la taille d'une matrice. La fonction génère un vecteur ligne où le premier élément est le nombre de lignes de la matrice, et le second élément est le nombre de colonnes.

Par exemple, si on a la matrice A suivante :

5.7	0.2	6.2	5.1
8.7	7.1	2.4	8.4
1.2	0.4	8.2	6.4
2.3	6.7	6.5	5.2
2.1	1.8	0.6	7.2

A

Alors que donnera ce code ?

```
>> s = size(A)
```

s

5	4

← La matrice A possède 4 colonnes

La matrice A possède 5 lignes

Taille d'une matrice

La fonction **numel** renvoie le nombre d'éléments d'une matrice passée en paramètre.

Par exemple, si on a la matrice A suivante :

5.7	0.2	6.2	5.1
8.7	7.1	2.4	8.4
1.2	0.4	8.2	6.4
2.3	6.7	6.5	5.2
2.1	1.8	0.6	7.2

A

Alors on aura:

```
>> numel(A)

ans =

    20
```

Opérations de base sur les matrices

Matlab propose tout un ensemble d'opérations usuelles sur les matrices

Symbole	Description	Exemple
+ - *	Les opérations de base (addition, soustraction, produit matriciel). Les tailles des matrices doivent être compatibles	A+B A*B
^	La puissance matricielle (itération du produit matriciel)	A^3
'	Transposée d'une matrice	A'
inv	L'inversion d'une matrice (si son inverse existe)	inv(A)

D'autres opérations peuvent être réalisées sur chaque élément de A

Symbole	Description	Exemple
+ - * /	Réalise l'opération entre un scalaire et chaque élément de la matrice.	5.4*A
.*	Réalise la multiplication terme à terme de deux matrices de même taille.	A.*B
.^	Met à une certaine puissance chaque élément de la matrice	A.^3

Opérations de base sur les matrices

Il est aussi possible d'appeler des fonctions usuelles (**sqrt, cos, sin, exp**, ...) sur une matrice, afin qu'elles s'exécutent sur chaque élément de la matrice.

Par exemple, si on a cette matrice A :

4	25	36
2	9	49

A

Alors on aura :

2	5	6
1.4	3	7

sqrt(A)

Opérations avancées sur les matrices

On peut calculer la somme des éléments d'une matrice, le long des lignes ou des colonnes, avec la fonction **sum**

Par exemple, si on a cette matrice A :

2.6	3.9	10
1.0	-3	3.2

A

Alors **sum(A,1)** calculera la somme des éléments de A le long des colonnes, tandis que **sum(A,2)** calculera la somme le long des lignes :

3.6	0.9	13.2

sum(A,1)

16.5
1.2

sum(A,2)

La fonction **prod** fait le même travail, mais en faisant le produit des éléments.

Opérations avancées sur les matrices

A vous de jouer : on considère la matrice A ci-dessous. Que donne la commande ci-dessous comme résultat ?

```
>> b = sum(A,2)
```

3	2	1	8
4	1	3	6
2	2	0	5

A

14
14
9

b

Opérations avancées sur les matrices

A vous de jouer : on considère la matrice A ci-dessous. Que donne la commande ci-dessous comme résultat ?

```
>> b = sum(A,1)
```

3	2	1	8

A

3	2	1	8

b

Opérations avancées sur les matrices

A vous de jouer : comment réaliser la somme de tous les éléments d'une matrice A ?

```
>> b = sum(A,1)
>> c = sum(b,2)
```

Ou, plus court

```
>> c = sum(sum(A,2),1)
```

Opérations avancées sur les matrices

On peut aussi, avec les fonctions **min** et **max**, calculer les éléments maximum ou minimum le long des lignes ou des colonnes d'une matrice.

Par exemple, si on a cette matrice A :

2.6	3.9	10
1.0	-3	3.2

A

Alors **max(A, [ ], 1)** calculera les éléments maximaux de A le long des colonnes, tandis que **max(A, [ ], 2)** calculera les éléments maximaux le long des lignes :

2.6	3.9	10

max(A,[],1)

10
3.2

max(A,[],2)

Filtrage de matrices

La fonction **find** permet de trouver les éléments d'une matrice qui respectent un certain critère. Cette fonction renvoie un vecteur colonne qui contient les numéro d'ordre des éléments recherchés.

Par exemple, si on a cette matrice A :

8.7	7.1	2.4	8.4	6.1
1.2	0.4	8.2	6.4	2.7
2.3	6.7	6.5	5.2	10

A

Alors **find(A > 8)** renvoie les numéros d'ordre des tous les éléments de A supérieurs à 8 :

1
8
10
15

find(A>8)

Filtrage de matrices

On peut ensuite utiliser ces numéros d'ordre pour indexer directement A et influencer sur ces éléments.

8.7	7.1	2.4	8.4	6.1
1.2	0.4	8.2	6.4	2.7
2.3	6.7	6.5	5.2	10

A

Par exemple, on peut mettre à 0 tous les éléments de A supérieurs à 8 :

```
>> l = find(A>8)
>> A(l) = 0
```

0	7.1	2.4	0	6.1
1.2	0.4	0	6.4	2.7
2.3	6.7	6.5	5.2	0

A

On peut aussi faire plus court :

```
>> A(find(A>8))=0
```

Filtrage de matrices

A vous de jouer : comment rajouter 1 à tous les éléments d'une matrice A plus petits que 1 ?

```
>> k = find(A<1)
>> A(k) = A(k)+1
```

Filtrage de matrices

La fonction **find** cherche les éléments d'une matrice respectant un certain critères. Voici les différents critères que l'on peut utiliser :

Symbole	Description	Exemple
> < >= <=	Les comparaisons de base (supérieur, inférieur, supérieur ou égal, inférieur ou égal)	A >= 3 A < 5
==	L'égalité	A == 7.2
~=	L'inégalité	A ~= 8

On peut combiner des critères :

Symbole	Description	Exemple
&	Combine deux critères avec un et logique.	(A > 3) & (A<8)
|	Combine deux critères avec un ou logique.	(A > 6) | (A < 3)
~	Inverse un critère	~((A > 3) & (A < 4))

Filtrage de matrices

A vous de jouer : Réalisez l'opération de valeur absolue sur une matrice, à l'aide du mot clef **find**.

```
>> k = find(A<0)
>> A(k) = -A(k)
```

Filtrage de matrices

On peut aussi générer une matrice de booléen indiquant si les éléments d'une matrice respectent un certain critère.

Par exemple, si on a cette matrice A :

8.7	7.1	2.4	8.4	6.1
1.2	0.4	8.2	6.4	2.7
2.3	6.7	6.5	5.2	10

A

Si on exécute ce code :

```
>> B = (A < 3)
```

On aura

B

0	0	1	0	0
1	1	0	0	1
1	0	0	0	0

Un 1 dans B indique que la case correspondante dans A respecte le critère, et un 0 indique qu'elle ne le respecte pas.

Filtrage de matrices

A vous de jouer : Réalisez l'opération de valeur absolue sur une matrice, sans le mot clef **find** mais en utilisant les matrices de booléens.

```
>> k = (A<0)
>> k = -2*k + 1
>> A = A.*k
```

Générer de nouvelles matrices

Matlab propose des fonctions permettant de générer de nouvelles matrices.

Fonction	Description	Exemple
rand(n,m)	Génère une matrice remplie de nombre aléatoires entre 0 et 1, de n lignes et m colonnes.	A = rand(3,5)
ones(n,m)	Génère une matrice remplie de 1, de n lignes et m colonnes	A = ones(4,4)
zeros(n,m)	Génère une matrice remplie de 0, de n lignes et m colonnes	A = zeros(1,3)

Concaténation de matrices

On peut générer de nouvelles matrices en concaténant d'anciennes matrices.

Si l'on a ces matrices :

4	5	6
7	1	1
2	8	7

A

4	3
1	2
5	8

B

5	4	4

C

On peut les concaténer de la même façon que l'on déclare les matrices :

```
>> D = [A B]
>> E = [A ; C]
```

Et on a :

4	5	6	4	3
7	1	1	1	2
2	8	7	5	8

D

4	5	6
7	1	1
2	8	7
5	4	4

E

Le mot clef if – exécuter du code sous condition

Le mot clef if

Le mot clef **if** (qui est bien un mot clef et non pas une fonction) permet d'exécuter du code si une condition est vraie.

Ce mot clef est utilisé dans la plupart des langages de programmation existant aujourd'hui : il est donc nécessaire de comprendre ce qu'il fait, car il n'est pas limité qu'au langage Matlab.

La syntaxe du **if** peut légèrement varier d'un langage à un autre, mais son sens reste le même.

Un premier programme avec if

A votre avis, à la fin de ce programme, que vaudra la variable **y** ?

```
x=input('Entrez une valeur svp');
y=2;

if x<4
    y=0;
end
```

A la fin du programme, la variable y vaudra

- sera mise à 0 si la valeur de x est inférieure à 4
- gardera sa valeur initiale sinon

Syntaxe du if

Voici la syntaxe générale de ce que l'on appelle « un bloc if » :

```
//Bloc de code 0

if <condition 1>
    //Bloc de Code 1
end

//Bloc de Code 2
```

Tout le **Bloc de code 0** est lu et exécuté. A l'issue de son exécution, la **condition 1** est évaluée.

Si elle est vraie, le **Bloc de code 1** est exécuté, puis le **Bloc de code 2.**

Sinon, c'est uniquement le **Bloc de code 2** qui est exécuté.

Notez la présence du mot clef **endif** pour fermer le « bloc if ».

Un exemple

Quelle fonction mathématique ce code permet-il de réaliser ?

```
x = input('Entrez une valeur : ');

if x<0
    x = -x;
end

disp(x)
```

Ce code permet de réaliser la valeur absolue :

Si x est négatif, il est multiplié par -1,

Puis, quoiqu'il soit arrivé auparavant, x est affiché.

L'écriture de conditions

On peut utiliser différents symboles, comme on avait vu avec la fonction **find**, pour construire une condition. Une différence existe cependant pour combiner les conditions :

Symbole	Description	Exemple
&&	Combine deux conditions avec un et logique.	(A > 3) & (A<8)
||	Combine deux conditions avec un ou logique.	(A > 6) | (A < 3)
~	Inverse une condition	~((A > 3) & (A < 4))

Exemple :

```
x = input('Entrez une valeur : ');
y = input('Entrez une autre valeur : ');

if x>0 && y>0
    disp('Les deux valeur entrées sont positives')
end
```

Le mot clef elseif

Le mot clef **elseif**, qui accompagne toujours un **if**, permet d'exécuter du code si les conditions du **if** ou des autres **elseif** situés auparavant étaient fausses, et que sa propre condition est vraie.

Comme pour le **if**, ce mot clef est présent dans la plupart des langages de programmation modernes, même si sa syntaxe peut différer quelque peu selon le langage.

Un premier programme avec elseif

A votre avis, à la fin de ce programme, que vaudra la variable **y** ?

```
x=input('Entrez une valeur svp');
y=2;

if x<4
    y=0;
elseif x>9
    y=1;
end
```

A la fin du programme, la variable y

- sera mise à 0 si la valeur de x est inférieure à 4
- sera mise à 1 si la valeur de x est supérieure à 9
- gardera sa valeur initiale sinon

Un second programme avec elseif

Il est tout à fait possible, après un **if**, de placer plusieurs elseif. Qu'est ce que ce programme affichera ?

```
a=input('Entrez votre age');

if a>=100
    disp('Vous êtes assez âgé');
elseif a>=18
     disp('Vous êtes majeur');
elseif a>=16
     disp('Vous pouvez conduire (accompagné)');
end
```

Ici, ce programme affichera

- Que la personne est âgée si elle a 100 ans ou plus
- Sinon, que la personne est majeure si elle a 18 ans ou plus
- Sinon, que la personne peut conduire si elle a 16 ans ou plus

Un second programme avec elseif

Quelle est la différence entre ces deux programmes ?

```
a=input('Entrez votre age');

if a>=100
    disp('Vous êtes assez âgé');
elseif a>=18
     disp('Vous êtes majeur');
elseif a>=16
     disp('Vous pouvez conduire');
end
```

```
a=input('Entrez votre age');

if a>=100
    disp('Vous êtes assez âgé');
endif

if a>=18
     disp('Vous êtes majeur');
endif

if a>=16
     disp('Vous pouvez conduire');
end
```

Dans le programme de gauche, le **programme n'affichera toujours qu'une seule phrase** ou rien du tout. A droite, il peut afficher zéro, une, deux ou trois phrases (ex : a=110).

Syntaxe du if/elseif

Voici la syntaxe générale de ce que l'on appelle « un bloc if » avec **elseif** :

```
//Bloc de code 0

if <condition 1>
    //Bloc de Code 1
elseif <condition 2>
    //Bloc de Code 2
elseif <condition 3>
    //Bloc de Code 3
elseif ...
    //...
end

//Bloc de Code n
```

Tout le **Bloc de code 0** est exécuté. Ensuite, la **condition 1** est évaluée.

Si elle est vraie, le **Bloc de code 1** est exécuté, puis le **Bloc de code n.**

Sinon, si la condition 2 est vraie, le **Bloc de code 2** est exécuté, puis le **Bloc de code n.**

Sinon, si la condition 3 est vraie, le **Bloc de code 3** est exécuté, puis le **Bloc de code n.**

...

Sinon, **si toutes les conditions sont fausses**, c'est **uniquement le Bloc de code n** qui est exécuté.

Syntaxe du if/elseif

Questions

```
//Bloc de code 0

if <condition 1>
    //Bloc de Code 1
elseif <condition 2>
    //Bloc de Code 2
elseif <condition 3>
    //Bloc de Code 3
elseif ...
    //...
end

//Bloc de Code n
```

A quelles conditions le bloc de code 4 s'exécutera ?

Si les conditions 1, 2 et 3 sont fausses, et que la condition 4 est vraie

Est-ce que les blocs 3 et 4 peuvent être exécutés l'un après l'autre ?

Non, c'est le principe du elseif : soit le 3, soit le 4 sera exécuté.

Syntaxe du if/elseif

Questions

```
a=input('Entrez votre note de contrôle Matlab');

if a<14
    disp('Vous n'êtes pas très bon');
elseif a>=15
     disp('Pas mal');
elseif a>18
     disp('Vous pourrez passer en deuxième année');
elseif a<4
     disp('Il y a des places de libres en MACS...');
```

A quelles conditions le programme invitera l'utilisateur à aller en MACS ?

Si on n'a pas a<14, ni a>=15, ni a>18, et que l'on a a<4.

C'est impossible, ça n'arrivera jamais…

Que se passe-t-il si a vaut 14.5 ?

Le programme n'affiche rien, car aucune condition n'est satisfaite.

Autre chose ?

Oui, il manque le mot clef **end** à la fin du bloc if ! Matlab affichera une erreur !

Se passer du elseif

Question : Pouvez-vous réécrire le code en bas à gauche afin de ne plus utiliser de **elseif** mais plusieurs **if** ?

```
a = input('Entrez votre age : ');

if a>=100
    disp('Vous êtes assez âgé');
elseif a>=18
     disp('Vous êtes majeur');
elseif a>=16
     disp('Vous pouvez conduire');
end
```

```
a = input('Entrez votre age : ');

if a>=100
    disp('Vous êtes assez âgé');
end

if a>=18
     disp('Vous êtes majeur');
end

if a>=16
     disp('Vous pouvez conduire');
end
```

Ne fonctionne pas : si age vaut 110, le programme de gauche affichera un seul message tandis que le programme de droite en affichera trois !

Se passer du elseif

Question : Pouvez-vous réécrire le code en bas à gauche afin de ne plus utiliser de **elseif** mais plusieurs **if** ?

```
a = input('Entrez votre age : ');

if a>=100
    disp('Vous êtes assez âgé');
elseif a>=18
     disp('Vous êtes majeur');
elseif a>=16
     disp('Vous pouvez conduire');
end
```

```
a = input('Entrez votre age : ');

if a>=100
    disp('Vous êtes assez âgé');
end

if ~(a>=100) && a>=18
     disp('Vous êtes majeur');
end

if ~(a>=100) && ~(a>=18) && a>=16
     disp('Vous pouvez conduire');
end
```

Là, c'est bon !

Le mot clef else

Le mot clef **else**, qui accompagne toujours un **if**, permet d'exécuter du code si les conditions du **if** ou des autres **elseif** situés auparavant étaient fausses. Le **else** ne possède aucune condition l'accompagnant : il est exécuté si tous les autres choix ont été rejetés.

Comme pour le **if**, ce mot clef est présent dans la plupart des langages de programmation modernes, même si sa syntaxe peut différer quelque peu selon le langage.

Un exemple

Voici un code affichant un menu permettant d'additionner, de multiplier, de soustraire ou de diviser deux valeurs selon le choix d'un utilisateur :

```
x = input('Entrez une valeur : ');
y = input('Entrez une valeur : ');
z = input('Entrez votre choix d operation : ');

if z==1
    disp(x+y);
elseif z==2
    disp(x*y);
elseif z==3
    disp(x-y);
elseif z==4
    disp(x/y);
end
```

Que se passe-t-il si l'utilisateur se trompe et entre une valeur qui n'a rien à voir avec le menu (comme 7) ?

Le programme n'affichera rien (aucun message d'erreur).

Un exemple

Le mot clef **else** va nous aider à gérer la situation d'un mauvais choix. Avec ce mot clef, on dira à Matlab d'exécuter un code à la condition qu'aucune ligne de code des if /elseif précédents n'ait été exécutée.

```
x = input('Entrez une valeur : ');
y = input('Entrez une valeur : ');
z = input('Entrez votre choix d operation : ');

if z==1
    disp(x+y);
elseif z==2
    disp(x*y);
elseif z==3
    disp(x-y);
elseif z==4
    disp(x/y);
else
    disp('Mauvais choix');
end
```

De cette façon, si l'utilisateur entre un mauvais choix, on le lui dira…

Un second exemple avec else

Le mot clef **else** permet de mettre, dans un bloc if, du code qui ne sera exécuté que si aucune des conditions précédentes du bloc if n'étaient vraie.

```
a=input('Entrez votre age');

if a>=100
    disp('Vous êtes assez âgé');
elseif a>=18
     disp('Vous êtes majeur');
elseif a>=16
     disp('Vous pouvez conduire (accompagné)');
else
    disp('Vous êtes jeune');
end
```

Ici, ce programme affichera

- Que la personne est âgée si elle a 100 ans ou plus
- Sinon, que la personne est majeure si elle a 18 ans ou plus
- Sinon, que la personne peut conduire si elle a 16 ans ou plus
- Sinon, que la personne est jeune

Syntaxe du if/elseif/else

Voici la syntaxe générale de ce que l'on appelle « un bloc if » avec **toutes les options possibles** :

```
//Bloc de code 0

if <condition 1>
    //Bloc de Code 1
elseif <condition 2>
    //Bloc de Code 2
elseif <condition 3>
    //Bloc de Code 3
elseif ...
    //...
else ...
    //Bloc de code n

end

//Bloc de Code n+1
```

Tout le **Bloc de code 0** est exécuté. Ensuite, la **condition 1** est évaluée.

Si elle est vraie, le **Bloc de code 1** est exécuté, puis le **Bloc de code n+1.**

Sinon, si la condition 2 est vraie, le **Bloc de code 2** est exécuté, puis le **Bloc de code n+1.**

Sinon, si la condition 3 est vraie, le **Bloc de code 3** est exécuté, puis le **Bloc de code n+1.**

Sinon, **si toutes les conditions sont fausses**, c'est **le Bloc de code n** puis **le Bloc de code n+1** qui sont exécutés.

Syntaxe du if/elseif/else

Il y a quelques règles à suivre lorsqu'on écrit un **bloc if** :

```
//Bloc de code 0

if <condition 1>
    //Bloc de Code 1
elseif <condition 2>
    //Bloc de Code 2
elseif <condition 3>
    //Bloc de Code 3
elseif ...
    //...
else ...
    //Bloc de code n

end

//Bloc de Code n+1
```

On **commence toujours par écrire le mot clef if,** suivi d'une condition est d'un bloc de code.

Ensuite, on peut mettre **zéro, un ou plusieurs mots clefs elseif**, chacun suivis d'une condition et d'un bloc de code.

Enfin, on peut mettre **zéro ou un mot clef else,** suivi d'un bloc de code.

On termine le bloc if avec le mot clef **end**

Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');

if x==0
    disp('x est nul');
elseif x>0
    disp('x est positif');
else x<0
    disp('x est négatif');
end
```

Ce programme possède une erreur de syntaxe : il ne doit y avoir aucune condition d'écrite après le mot clef else.

Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');

if x==0
    disp('x est nul');
else
    disp('x est négatif');
elseif x>0
    disp('x est positif');
end
```

Ce programme possède une erreur de syntaxe : après un **else**, le bloc if est terminé et il faut nécessairement mettre le mot clef **end**. Il faut mettre le else à la toute fin !

Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');

if x==0
    disp('x est nul');
else
    disp('x est négatif');
else
    disp('x est positif');
end
```

Ce programme possède une erreur de syntaxe : il ne peut y avoir qu'un seul mot clef **else** dans un bloc **if**.

Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');

if x==0
    disp('x est nul');
elseif x<0
    disp('x est négatif');
elseif
    disp('x est positif');
end
```

Ce programme possède une erreur de syntaxe : le dernier **elseif** n'a pas de condition. Il faut lui en rajouter une ou le remplacer par **else**.

Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');

if x==0
    disp('x est nul');
if x<0
    disp('x est négatif');
if x>0
    disp('x est positif');
end
```

Ce programme possède une erreur de syntaxe : il y a plusieurs **if** mais un seul **end**. Soit on veut faire plusieurs blocs **if** et il faut terminer chacun d'entre eux par un **end**, soit on veut un seul bloc **if**, et il faut remplacer les **if** par des **elseif**.

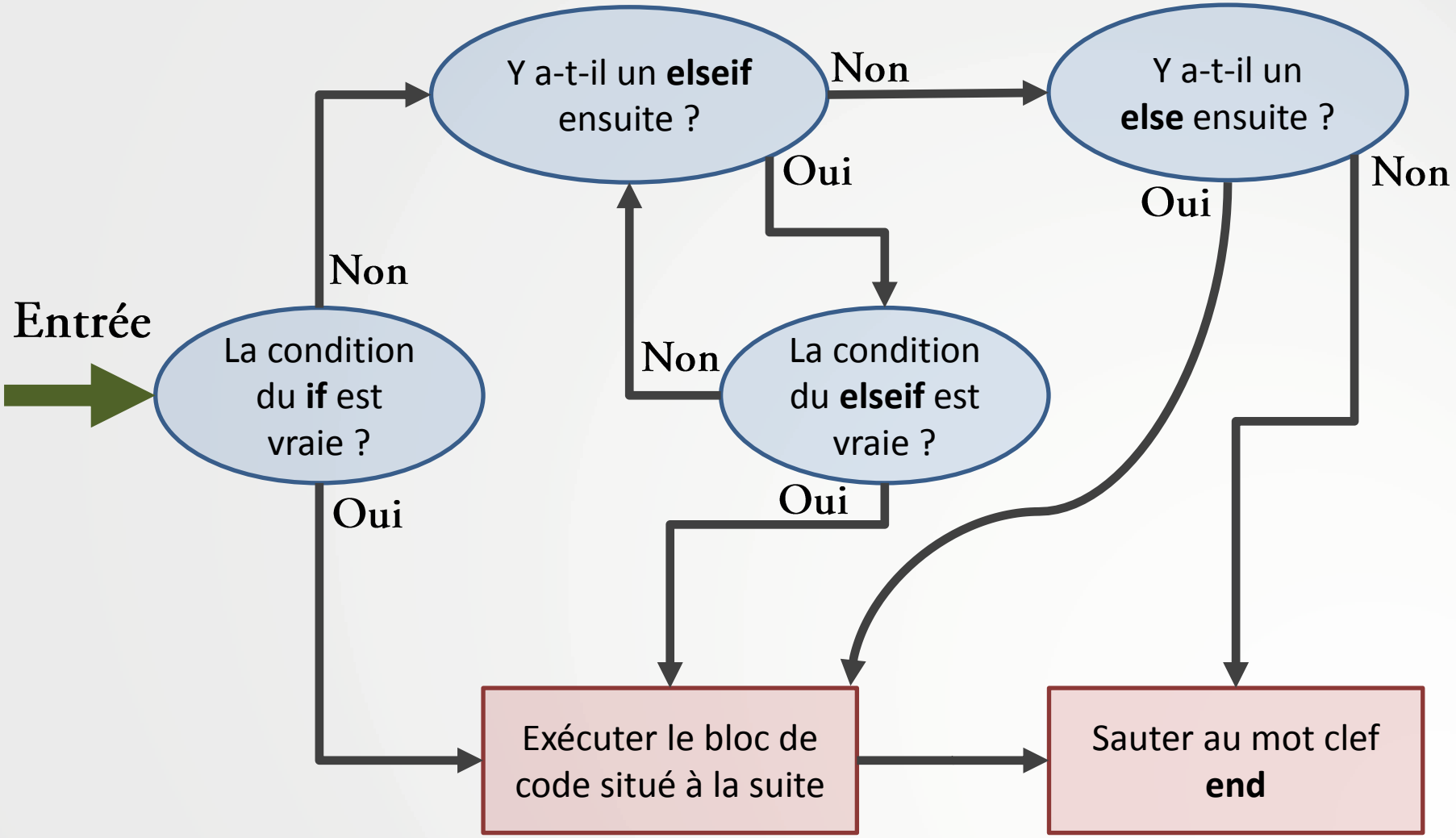
Exercice

Que pensez-vous de ce programme ?

```
x = input('Entrez une valeur ');
y = input('Entrez une valeur ');

if x>=0
    if y >=0
        disp('x et y sont positifs');
    else
        disp('x est positif et y est négatif');
    end
else
    if y >=0
        disp('x est négatif et y est positif');
    else
        disp('x et y sont négatifs');
    end
end
```

Il n'y a aucune erreur de syntaxe dans ce programme : il est tout à fait autorisé de mettre, dans un **if**, un bloc de code qui contient lui-même un **if**.

Diagramme d'évaluation d'un bloc if



La fonction modulo

La fonction modulo, qui s'écrit **mod**, permet de récupérer le reste de la division euclidienne d'un nombre par un autre.

Par exemple, ce code affichera le reste de la division euclidienne de 23 par 7 :

```
>> a = mod(23,7);
>> disp(a)
     2
```

Le résultat est 2 car on a bien $23 = 7*3 + 2$ ← Le reste

Le quotient

Pourquoi parler de la fonction modulo maintenant ?

La fonction modulo

Le modulo est très pratique pour tester si un nombre est divisible par un autre : dans ce cas, le reste de la division euclidienne est 0. On l'utilise souvent dans une condition pour tester la divisibilité d'un nombre par un autre

Par exemple, ce code permet de tester si le nombre **a** est divisible par 7 :

```
a = input('Entrez une valeur : ');

if mod(a,7) == 0
    disp('a est divisible par 7');
else
    disp('a n est pas divisible par 7');
end
```

En langage C, le modulo est un opérateur représenté par le symbole %. Pour tester si a est divisible par 7, on écrira (en C) :

```
if (a%7 == 0)
{ ... }
```

La fonction modulo

Que fait ce code ?

```
a = input('Entrez une valeur : ');

if mod(a,2) == 0
    disp('a est un nombre pair');
else
    disp('a est un nombre impair');
end
```

Ce code teste la parité d'un nombre.

Conclusion

Le mot clef **if** permet d'exécuter une partie du code si une condition est vraie. Cela permet d'obtenir des programmes dont le comportement change selon les actions/données de l'utilisateur.

Les mots clefs **elseif** et **else** complètent les « pouvoirs » du **if**, en enrichissant ses possibilités.

On peut complètement se passer d'eux, en utilisant d'autres mots clefs **if** avec des conditions plus complexes. Cependant, leur existence rend la lecture et l'écriture d'un programme plus agréable et compréhensible.

Par-contre, le mot clef **if** ne peut pas être remplacé par d'autres mots clefs : sans lui, impossible de faire un programme qui change de comportement selon les données.

13

Les boucles

Les boucles

Cependant, ce qui est intéressant avec un ordinateur, c'est qu'il est capable d'exécuter des milliards de lignes de code à la seconde.

Or, s'il fallait écrire soi-même chacune de ces lignes de code, ce serait bien trop long et l'intérêt d'un ordinateur serait vraiment diminué.

Les boucles permettent de répéter plusieurs fois un même bloc de code, permettant de faire faire à la machine des milliards d'opérations alors que l'on a soi même écrit qu'une dizaine de lignes de code.

Il y a deux types de boucles que l'on va voir : le **for** et le **while**.

Première utilisation de la boucle for

Voici un exemple de boucle **for**. Imaginons que l'on a ce code

```
>> v = rand(1,5);
>> disp(v);

0.8003    0.1419    0.4218    0.9157    0.7922
```

Alors ce code affichera

```
>> for k = v
>>     disp('La valeur de k est ‘)
>>     disp(k)
>> end

La valeur de k est 0.8003

La valeur de k est 0.1419

La valeur de k est 0.4218

La valeur de k est 0.9157

La valeur de k est 0.7922
```

A l'aide d'une boucle **for**, on a transformé deux lignes de code **disp** en 10 affichages.

La variable **k** prend, à chaque tour de boucle, une des valeurs contenues dans le vecteur **v**.

A chaque tour de boucle **for**, le programme exécutera les deux lignes de codes contenues dans le bloc **for** mais la valeur de **k** changera.

Un autre exemple de boucle for

Voici un exemple de boucle **for**.

```
>> for i = 1:8
>>      disp(i);
>> end

    1
    2
    3
    4
    5
    6
    7
    8
```

L'instruction **1:8** génère, comme on l'a vu, un vecteur ligne contenant les entiers de 1 à 8.

La boucle for répète le code **disp(i)** pour chaque élément du tableau, et i prend une des valeurs du tableau à chaque fois.

Voilà pourquoi ce code affiche tous les entiers de 1 à 8.

Un autre exemple avec une matrice

Que se passe-t-il si l'on parcourt une matrice ?

```
>> A = rand(3,2);
>> disp(A);
 0.8147    0.9134
 0.9058    0.6324
 0.1270    0.0975

>> for i = A
>>      disp('La valeur de i est ')
>>      disp(i);
>>  end

    La valeur de i est
     0.8147
     0.9058
     0.1270

    La valeur de i est
     0.9134
     0.6324
     0.0975
```

Dans ce cas, la boucle **for** parcourt toutes les colonnes de **A**.

A chaque tour de boucle, **i** vaut une des colonnes de **A**.

En réalité, que ce soit une matrice ou un vecteur, la boucle **for** parcourt toujours les colonnes de l'élément.
Dans le cas d'un vecteur ligne (qui possède une ligne et plusieurs colonnes), on a l'impression que ce sont les éléments du vecteur qui sont parcourus.

Un autre exemple avec un vecteur colonne

Que se passe-t-il si l'on parcourt un vecteur colonne ?

```
>> v = rand(4,1);
>> disp(v);
    0.9572
    0.4854
    0.8003
    0.1419

>> for i = v
>>      disp('La valeur de i est ')
>>      disp(i);
>>  end

    La valeur de i est
    0.9572
    0.4854
    0.8003
    0.1419
```

Comme on pouvait s'y attendre, la boucle **for** utilise la variable **i** pour parcourir toutes les colonnes de **v**.

Ici, **v** possède une seule colonne (et quatre lignes), donc la boucle ne parcourt qu'un seul élément.

Syntaxe du for

Voici les règles à suivre pour écrire une **boucle for** :

```
//Bloc de code 0

for <variable> = <matrice>
    //Bloc de Code 1
end

//Bloc de Code 2
```

On **commence toujours par écrire le mot clef for,** suivi d'un nom de variable, du signe égal, et d'un nom de matrice

Ensuite, on écrit un bloc de code (qui peut contenir, comme tout bloc de code, des opérations, des **blocs if** ou d'autres **bloc for**).

On termine le **bloc for** avec le mot clef **end**

Le **bloc de code 0** est d'abord exécuté. Ensuite, pour chaque colonne **c** de **<matrice>**, le **bloc de code 1** est répété : à chaque fois, **<variable>** est égale à **c**. Enfin, le **bloc de code 2** est exécuté.

Un exemple

Que fait ce code ?

```
v = rand(1,100000);
s = 0;

for i = v
    s = s+i;
end
disp(s);
```

Cette boucle permet de calculer, dans la variable **s**, la somme des éléments du vecteur v.

On peut aussi faire

```
s = sum(v);
```

Quel code est le plus rapide ?

Temps d'exécution

Voici un graphique affichant, selon la taille du vecteur **v** de l'exemple précédent, le temps d'exécution des deux programmes (avec ou sans boucle) permettant de calculer la somme des éléments de **v**

Taille du vecteur	Temps d'exécution avec for	Temps d'exécution sans for
$1*10^6$	0,92s	1,1ms
$2*10^6$	1,65s	1,6ms
$3*10^6$	2,48s	3,6ms
$4*10^6$	3,20s	3,0ms
$5*10^6$	3,96s	3,7ms
$6*10^6$	4,90s	6,7ms
$7*10^6$	5,63s	5,0ms
$8*10^6$	6,46s	5,8ms
$9*10^6$	7,33s	6,7ms

Temps d'exécution

La version du code utilisant la fonction **sum** est bien plus rapide que celle utilisant la boucle **for**. Enlever une boucle afin de la remplacer par une fonction de Matlab (**sum, find, prod, .*, …**) permet d'accélérer grandement le code : cela s'appelle la **vectorisation du code**.

Pourquoi les boucles **for** de Matlab sont-elles lentes ?

Le langage C est un langage **compilé**, c'est-à-dire qu'un compilateur (**gcc**) analyse le code entièrement pour le traduire en assembleur avant de l'exécuter.

Pendant cette analyse, des optimisations de code ont lieu, particulièrement dans les boucles for.

Matlab est un langage **interprété** (un langage de script) : le code est lu au moment de l'exécution, et très peu d'optimisation est réalisée.

Temps d'exécution

Pourquoi les fonctions internes de Matlab (**sum, find,** …) sont-elles efficaces ?

Ces fonctions ne sont pas écrites en Matlab, mais en C ou un autre langage compilé.

Par exemple, la fonction **sum** appelle un programme écrit en C (ou autre) pour faire la somme des éléments d'un vecteur.

Ces programmes sont optimisés de façon très efficace par l'équipe de développement de Matlab. En général, **le code est parallélisé et optimisé** pour profiter pleinement de tous les processeurs d'une machine ainsi que de leur architecture particulière.

La commande **mex** permet de prendre un programme C et interagir avec depuis Matlab.

Syntaxe du while

L'autre boucle à connaître (et, dans Matlab, en général plus utile que la boucle **for**) est la **boucle while** :

On **commence toujours par écrire le mot clef while,** suivi d'une condition (comme avec le **if**).

```
//Bloc de code 0

while <condition 1>
    //Bloc de Code 1
end

//Bloc de Code 2
```

Ensuite, on écrit un bloc de code (qui peut contenir, comme tout bloc de code, des opérations, des **blocs if**, **for** ou **while**).

On termine le **bloc while** avec le mot clef **end**

Le **bloc de code 0** est d'abord exécuté. Ensuite, si la **condition 1** est vraie, le **bloc de code 1** est exécuté. Il est répété jusqu'à ce que la **condition 1** devienne fausse. Ensuite, le **bloc de code 2** est exécuté.

Syntaxe du while

L'autre boucle à connaître (et, dans Matlab, en général plus utile que la boucle **for**) est la **boucle while** :

```
//Bloc de code 0

while <condition 1>
    //Bloc de Code 1
end

//Bloc de Code 2
```

Il est impératif que le **bloc de code 1** fasse évoluer la **condition 1**.

Si ce n'est pas le cas et que le programme « entre » dans la boucle, il n'en sortira jamais (ce n'est pas souhaitable).

Première utilisation de la boucle while

Voici un premier programme avec une boucle **while**. Que fait ce programme ?

```
a = input('Entrez une valeur entiere : ');

s=1;
while a>1
    s = s*a;
    a = a-1;
end
disp(s);
```

Ce programme calcule, dans la variable **s**, le produit de toutes les valeurs entre 1 et a.

Il calcule donc la factorielle de a.

Seconde utilisation de la boucle while

Voici un second programme avec une boucle **while**. Que fait ce programme ?

```
a = input('Entrez une valeur positive : ');

while a<0
    a = input('Entrez une valeur positive : ');
end

disp(a);
```

Ce programme demande une valeur positive à l'utilisateur et lui casse les pieds (en redemandant) tant qu'il ne le fait pas…

Troisième utilisation de la boucle while

Voici un troisième programme avec une boucle **while**. Que fait-il ?

```
a = input('Entrez une valeur : ');
b = input('Entrez une valeur : ');

n = min(a,b);
while ~(mod(a,n)==0 && mod(b,n)==0)
    n=n-1;
end

disp(n)
```

Ce programme cherche un diviseur commun à a et b, en partant de la plus petite de ces deux valeurs : il calcule pgcd(a,b).

Quelle boucle utiliser ?

Vaut-il mieux utiliser une boucle **for** ou **while** dans un programme ?

En général, si on sait à l'avance combien de fois la boucle doit se répéter, on utilisera une boucle **for**. Sinon, on utilisera un **while**.

Parmi les trois programmes que l'on vient de voir en exemple du **while**, lesquels devraient utiliser une boucle **for** ?

Programme	for ou while ?	Explications
factorielle(a)	**for/vectorisé**	La boucle se répétera (a-1) fois
nombre positif	**while**	On ne sait pas combien de fois l'utilisateur va saisir un nombre négatif
pgcd(a,b)	Les deux	Soit on teste tous les entiers entre 1 et min(a,b), soit on teste ceux entre a et 1 jusqu'à en trouver un qui divise.

Factorielle avec for

Pour réaliser une factorielle avec une boucle **for**, on peut faire ainsi :

```
a = input('Entrez une valeur entiere : ');

s=1;
for i = 1:a
    s = s*i;
end
disp(s);
```

On peut évidemment vectoriser ce code et éviter la boucle **for** :

```
a = input('Entrez une valeur : ');
s = prod(1:a);
disp(s);
```

On peut aussi utiliser la fonction factorielle de Matlab : **factorial**

Le mot clef **break**

Le mot clef **break** permet d'interrompre une boucle **for** ou **while** et de poursuivre l'exécution du code après la boucle.

Il est parfois considéré comme une mauvais pratique de programmation, car il est souvent possible de modifier les conditions du **while** afin de l'éviter.

De plus, on ne peut pas l'utiliser dans le cas d'une vectorisation de code.

Nous n'examinerons pas en détail ce mot clef : sachez qu'il existe, mais évitez de trop souvent l'utiliser.

Conclusion

Les boucles étendent les possibilités d'un programme, en **répétant plusieurs fois un même comportement**.

Ceci permet de réaliser un nombre important d'opérations sans pour autant avoir à coder chacune d'entre elles.

Pour pouvoir profiter des boucles, il faut être capable de penser un programme comme une série d'actions similaires.

Pour conserver un programme rapide, il est préférable **d'éviter les boucles si possible, et d'utiliser les fonctions de Matlab agissant sur les matrices** : cela s'appelle la vectorisation de code.

Avant d'optimiser un code, il est recommandé de l'écrire avec des boucles afin de tester son fonctionnement, puis de tenter de le vectoriser.