الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire

Ministre de L'Enseignement Supérieure
et de la Recherche Scientifique.
Université : 8 Mai 1945 Guelma
Faculté : Mathématiques, Informatique
et de Sciences de la Matière.
Département : informatique.

وزارة التعليم العالي والبحث العلمي.
جامعة 8 ماي 1945 قالمة.
كلية الرياضيات، الإعلام الآلي
وعلوم المادة.
قسم الإعلام الآلي.

Polycopié du Cours

Outils d'informatique

Cours et Exercices

Niveau : 1ère année Master, Mathématiques

2018/2019

Par : Dr. Chemseddine CHOHRA

Table des matières

Objectifs

La nécessité des mathématiques appliquées suggère à l'étudiant la maîtrise de l'outil informatique, notamment la manipulation des logiciels de calcul numérique et symbolique. Le logiciel Matlab répond suffisamment à ces besoins.

À l'issue de ce module, l'étudiant va être capable de :

- Concevoir des algorithmes pour résoudre des problèmes mathématiques.
- Traduire un algorithme donné en programme Matlab.
- Résoudre des problèmes numériques.

Introduction

Matlab est un logiciel largement utilisé dans le domaine de l'industrie, la recherche et l'enseignement. Il est même devenu un outil standard dans de nombreux domaines (traitement d'image, statistiques, analyse, calcul scientifique ... etc.). Ce cours s'adresse aux étudiants en master de mathématique dans le but de leur apprendre à écrire des programmes de calcul scientifique en utilisant l'environnement Matlab.

Le premier chapitre de ce cours rappelle les concepts de base de l'algorithmique. Nous présentons dans ce chapitre les éléments de base permettant de concevoir un algorithme pouvant être implémenté et exécuté sur un ordinateur. A l'issue de ce chapitre, l'étudiant doit comprendre les instructions d'entrée-sortie, la manipulation des variable à l'aide de l'opération d'affectation, les structure de contrôle (conditions et boucles) et l'utilisation des sous programmes (principalement les fonction) pour simplifier les problèmes compliqués.

Dans le deuxième chapitre nous introduisons l'environnement Matlab, ce chapitre explique l'utilité des différents composants de l'interface Matlab, l'espace de travail, l'aide en utilisant la fonction `help` ou même l'aide en ligne. Dans la deuxième partie de ce chapitre nous allons aussi fixer le regard sur la manipulation des vecteurs et matrices dans Matlab et les différentes opérations d'algèbre linéaire.

Le troisième chapitre introduit les scripts et fonctions, nous passons ici de l'exécution d'une seule instruction interprété immédiatement à l'implémentation d'un algorithme composé de plusieurs instructions permettant de résoudre un problème. A l'issue de ce chapitre, l'étudiant sera capable de déclarer des variables, utiliser des structure de contrôle et des sous programmes pour interpréter n'importe quel algorithme basique en un script (ou fonction) Matlab dont il peut observer l'exécution.

Le quatrième chapitre met le point sur la résolution des problèmes numériques avec Matlab. Nous présentons d'abord les différentes méthodes de résolution des équations non linéaires, puis nous implémentons les méthodes directes et les méthodes itératives de résolution des systèmes d'équations linéaires.

Finalement, dans le dernier chapitre, nous expliquons les différentes fenêtres graphiques. On aborde en particulier la visualisation des fonctions en deux et trois dimensions à l'aide des deux fonctions `plot` et `plot3`.

Prérequis : Informatique de base (installer des logiciels et éditer des fichiers texte).

Chapitre 1 - Rappel sur l'algorithmique

I

L'objectif de ce premier chapitre de ce cours de rappeler rapidement les concepts de base d'algorithmique déjà vus en première année. Nous allons d'abord commencer par la définition de ce que c'est un algorithme, puis nous allons présenter les instructions de base qu'on a le droit d'utiliser pour concevoir un algorithme permettant de résoudre un problème donné. Finalement, il faut savoir qu'un algorithme n'est pas un code exécutable, mais juste une description détaillée des étapes de résolution d'un problème pouvant facilement être traduite en programme écrit dans n'importe quel langage de programmation, dans notre cas nous allons utiliser Matlab.

1. Qu'est ce qu'un algorithme

Définition : Algorithme

Un algorithme est une séquence d'étapes obéissant à un enchaînement bien déterminé, permettant de résoudre un problème spécifique. Un algorithme est correcte s'il est capable de résoudre le problème pour chaque instance donnée.

Définition : Problème

Un problème est un questionnement nécessitant une solution.

Complément : Résoudre un problème

La résolution d'un problème est de concevoir une méthode de raisonnement qui permet de construire en un nombre fini d'étapes élémentaires l'énoncé de la solution. ce qu'on peut appeler autrement *trouver un algorithme*.

Fondamental : Étape élémentaire

Chacune des étapes élémentaires qui composent un algorithme doit satisfaire les deux critères suivants :

- *non-ambigue* : c'est à dire définie d'une façon rigoureuse et ne peut pas être ouverte à différentes interprétations ;
- *effective* : pouvant être effectivement réalisée par une machine.

Définition : Programme

Un programme destiné à être exécuté par un ordinateur est la description d'un algorithme dans un langage accepté par cette machine (par exemple : un programme écrit en langage Matlab).

2. Structure de données

Définition

Une structure de données est un ensemble organisé d'informations reliées, ces informations peuvent être traitées collectivement ou individuellement.

Exemple : Les tableaux

L'exemple le plus simple et le plus répandu d'une structure de données est les tableaux monodimensionnels (appelés aussi vecteurs). Un tableau est constitué d'un certain nombre de composantes de même type. On peut effectuer des opérations sur chaque composante prise individuellement mais on dispose aussi d'opérations globales portant sur le tableau considéré comme un seul objet.

Attention

Une structure de données est caractérisée par ses composantes et leur arrangement mais surtout par son mode de traitement. Ainsi deux structures ayant les mêmes composantes et les mêmes arrangements peuvent être considérées comme différentes si leurs modes d'exploitation sont différents.

3. Structure d'un algorithme

Fondamental

Un algorithme se compose de trois parties fondamentales : l'*entête*, la *partie déclaration* et le *corps de l'algorithme*.

- Dans l'entête, on donne un nom à l'algorithme en utilisant la syntaxe suivante : `algorithme nom_algorithme;` il est recommandé que le nom de l'algorithme soit significatif du problème qu'il résout :
  - exemple d'un identifiant significatif : `algorithme somme;`
  - exemple d'un identifiant non-significatif : `algorithme Alg01;`
- Dans la partie déclaration, il faut définir toutes les variables qui seront utilisées dans le corps de l'algorithme en utilisant la syntaxe suivante : `var nom_variable : type;` il est recommandé que le nom d'une variable soit significatif de son rôle dans le corps de l'algorithme pour faciliter la lecture de ce dernier.
- Le corps de l'algorithme consiste en une séquence d'instruction placé entre les deux mots clés `début` et `fin`.

Remarque : La partie déclaration et types de variables

Vu que nous allons utiliser le langage Matlab (qui n'oblige pas à déclarer des variables avant de les initialiser, et qui ne limite pas une variable à un seul type de données) pour traduire les algorithmes en programmes, nous allons ignorer la partie déclaration pour garder une cohérence entre les algorithmes et leurs programmes correspondants. Nous supposons aussi qu'une variable peut changer de type (on peut affecter des valeurs de types différents à la même variable).

3.1. Types de données

La notion de type est très importantes en algorithmique (ainsi qu'on programmation) que toute valeur affacté à une variable doit forcément avoir un type. Nous distinguons quatre types élémentaires :

- le type `entier` : utilisé pour stocker des valeurs entières positives ou négatives ;
- le type `réel` : utilisé pour stocker des nombres réels (à virgule) ;
- le type `caractère` : utilisé pour stocker des caractères ;
- le type `booléen` : utilisé pour stocker des valeur binaire ou logiques (0/1 ou vrai/faux).

Remarque : Retour sur les structures de données

Les types de bases peuvent être utilisés comme briques de base pour créer des types plus complexes, c'est ce que nous appelons *les structures de données*. Les tableaux sont la structure de données la plus importante pour la suite de ce cours, on peut définir des tableaux d'entiers, de réels, de caractères (ou aussi chaînes de caractères) ou de booléen.

On peut aussi utiliser les structure de données qu'on a personnalisées comme composantes d'autres structures encore plus complexes. Prenons l'exemple d'un tableau de tableau d'entiers, aussi appelé tableau bi-dimensionnel ou matrice.

3.2. Variables

Les variables sont utilisés dans un algorithme pour pouvoir manipuler des données, chaque variable doit posséder :

- *Un nom* : aussi appelé un identificateur, ce dernier doit être unique pour permettre à l'algorithme de reconnaître d'une façon précise quelle donnée manipuler.
- *Une valeur* : qui évolue au cours d'évolution de l'algorithme. Cette valeur peut être lue ou modifiée en évoquant l'identifiant de la variable correspondante.

Cette notion (variable) est fondamentale, parce qu'elle permet de manipuler des valeurs (numériques ou non) et est à la base de la plupart des calculs et traitements que l'on peut réaliser avec un langage de programmation.

Remarque : Retour sur les structures de données

Les types de bases peuvent être utilisés comme briques de base pour créer des types plus complexes, c'est ce que nous appelons *les structures de données*. Les tableaux sont la structure de données la plus importante pour la suite de ce cours, on peut définir des tableaux d'entiers, de réels, de caractères (ou aussi chaînes de caractères) ou de booléen.

On peut aussi utiliser les structure de données qu'on a personnalisées comme composantes d'autres structures encore plus complexes. Prenons l'exemple d'un tableau de tableau d'entiers, aussi appelé tableau bi-dimensionnel ou matrice.

3.3. Le corps de l'algorithme

C'est dans cette partie qu'on décrit les étapes élémentaires à suivre pour résoudre le problème. Ces étapes élémentaires sont aussi appelées *des instructions*. Nous expliquons de quoi s'agit le terme instructions avec plus de détails dans la prochaine section.

4. Instructions de base et opérations

Les instructions décrivent les actions à effectuer par l'algorithme pour résoudre le problème. Chaque deux instructions consécutives sont séparées par un point-virgule. Nous allons diviser les instructions en plusieurs catégories.

4.1. Les instructions de lecture et écriture

Il s'agit des deux instruction `lire` et `écrire`. L'instruction `lire` permet d'initialiser une variable à partir d'une saisie faite au clavier, alors que l'instruction `écrire` comme son nom l'indique sert à afficher du texte ou la valeur d'une variable.

Complément : L'instruction de lecture

Pour initialiser une variable saisie par l'utilisateur, on utilise la fonction `lire`, suivi du nom de la variable que l'on veut saisir entre deux parenthèses. L'instruction `lire` a pour seul 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), aucun message ne s'affiche pour indiquer à l'utilisateur ce qu'il doit faire, il est donc recommandé d'utiliser la fonction `écrire` pour afficher un message à l'utilisateur lui expliquant qu'il doit saisir une valeur.

Complément : L'instruction d'écriture

Pour afficher un élément à l'écran, on utilise la fonction `écrire`, suivi entre parenthèses de l'élément à faire apparaître . Cet élément est soit du texte brut écrit entre doubles 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 : Utilisation des instructions lire et écrire

```
1 algorithme lire_et_écrire;
2 début
3 écrire("Veuillez saisir la valeur de a : ");
4 lire(a);
5 écrire(a);
6 fin.
```

4.2. L'affectation

L'opération d'affectation permet de donner (ou d'affecter) une valeur à une variable en utilisant la syntaxe suivante :

```
variable  expression;
```

La valeur de la variable à gauche de l'opération d'affectation est mise à jour par le résultat d'évaluation de l'expression à droite. Cette expression peut être soit une valeur constante, soit une variable, une expression arithmétique ou autre types d'expressions que nous allons voir par la suite.

Exemple : Utilisation de l'affectation

```
1 algorithme affectation;
2 début
3 a ← 1;
4 b ← 2;
5 c ← a + 3; % c vaut 4
6 d ← c * b; % d vaut 8
7 fin.
```

4.3. Conditions et boucles

Ces structures permettent au programme de ne pas être purement séquentiel. Suite à la complexité de ces structures nous allons consacrer la prochaine section pour les expliquer avec plus de détails.

5. Les structures de contrôle

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

5.1. Exécution conditionnelle d'un bloc d'instructions

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 : si `a > b`).

Complément : Opérateurs logiques de base

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

- L'opérateur `et : cond1 et cond2` retourne la valeur logique vrai si et seulement si la condition `cond1` et la condition `cond2` retourne toutes les deux la valeur logique `vrai`, sinon le résultat retourné est la valeur logique `faux`.
- L'opérateur `ou : cond1 ou cond2` retourne la valeur logique vrai si et seulement si au moins l'une des deux conditions `cond1` et `cond2` retourne la valeur logique `vrai`, sinon le résultat retourné est la valeur logique `faux`.
- L'opérateur `non : non cond` retourne la valeur logique `vrai` si et seulement si `cond` retourne la valeur logique `faux`.

La structure `si ... alors` permet d'exécuter des instructions en fonction de la valeur d'une condition (ou le résultat d'un test). La syntaxe utilisée est la suivante :

```
si cond alors
début
   instruction_1;
   instruction_2;
   ..........
   instruction_n;
fin;
```

Cette structure fonctionne de la manière suivante :

- si la condition `cond` est `vraie`, les instructions écrites entre `début` et `fin` sont exécutées ;
- alors que lorsque la condition `cond` est fausse les instructions ne sont pas exécutées.

Dans la suite nous appelons une séquence d'instructions entre les deux mots clés début et fin un bloc d'instructions.

Exemple : Utilisation de la structure "si ... alors"

Dans cet exemple nous allons écrire un algorithme qui attend que l'utilisateur saisisse 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 n'est pas exécutée et le résultat n'est pas affiché.

```
 1 algorithme division;
 2 début
 3 lire(a);
 4 lire(b);
 5 si b ≠ 0 alors
 6    début
 7    c ← a / b;
 8    écrire(c);
 9    fin; % fin pour la condition avec un ';'
10 fin.    % fin de l'algorithme avec un '.'
```

Complément : si ... alors ... sinon

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 `si ... alors ... sinon`, dont la syntaxe est la suivante :

```
si cond alors
bloc_instructions_1;
sinon
bloc_instructions_2;
```

Cette structure fonctionne de la manière suivante :

- si la condition `cond` est vraie, alors le premier bloc d'instructions est exécuté ;
- sinon, le deuxième bloc d'instructions est exécuté.

Pour clarifier encore plus le fonctionnement de cette structure, nous allons donner un exemple en changeant l'algorithme précédent (division de `a` par `b`).

Exemple : Utilisation de la structure "si ... alors ... sinon"

```
1 algorithme division;
2 début
3 lire(a);
4 lire(b);
5 si b ≠ 0 alors
6    début
7    c ← a / b;
8    écrire(c);
9    fin;
10 sinon
11    début
12    écrire("Il n'est pas possible de diviser par zéro");
13    fin;
14 fin.
```

Dans l'exemple précédent la division par b n'est effectuée que lorsque la valeur de b est différente de zéro. Dans le cas inverse un message est affiché pour indiquer qu'il est impossible de diviser par zéro.

5.2. Les structures itératives

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

Définition : La boucle "tant que ... faire"

Sa syntaxe de la boucle `tant que` est la suivante :

```
tant que cond faire
   bloc_instructions;
```

Lorsque l'ordinateur rencontre cette structure, il procède de la manière suivante :

1. La condition est testée.
2. Si la condition est vraie, l'instruction ou les instructions du bloc sont exécutées, après on revient à l'étape 1 (tester la condition).
3. Si la condition est fausse, l'instruction ou les instructions du bloc ne sont pas exécutées et on passe aux instructions suivantes (après la boucle).

Exemple : Utilisation de la structure "tant que ... faire"

Dans cet exemple nous allons écrire un algorithme qui affiche tous les nombres entiers positifs entre 0 et 10 (inclus).

```
1 algorithme boucleTQ;
2 début
3 x ← 0;
4 tant que x ≤ 10 faire
```

```
5    début
6    écrire(x);
7    x ← x + 1;
8    fin;
9 fin.
```

Complément : La boucle "pour ... faire"

Lorsque le nombre d'itérations dont un bloc d'instructions doit être exécuté est connu à l'avance, la boucle pour est préférable a la boucle tant que. L'usage principal de la boucle pour est de faire la gestion d'un compteur qui évolue d'une valeur à une autre. La boucle pour est utilisée suivant la syntaxe suivante :

```
pour compteur allant de valeur1 à valeur2 faire
   bloc_instructions;
```

Lorsque l'ordinateur rencontre cette structure, il procède de la manière suivante :

1. la variable, jouant le rôle de compteur, est initialisée à la valeur1 ;
2. l'ordinateur teste si la variable est inférieure ou égale à la valeur2 :
    - si c'est le cas, l'instruction ou le bloc d'instruction est exécuté, la variable jouant le rôle de compteur est augmentée de 1, et puis on retourne à l'étape 2, et non à l'étape 1 qui initialise la variable ;
    - sinon, le bloc d'instructions n'est pas exécuté, et l'ordinateur passe aux instructions après la boucle

Exemple : Utilisation de la structure "pour ... faire"

Dans cet exemple nous allons réécrire l'algorithme précédent (affichage des nombres entre 0 et 10) en utilisant la boucle pour au lier de la boucle `tant que`.

```
1 algorithme bouclePour;
2 début
3 pour x allant de 0 à 10 faire
4    début
5    écrire(x);
6    fin;
7 fin.
```

Remarque

En réalité, les deux boucles `pour` et `tant que` peuvent être utilisées de façons différentes pour résoudre le même problème. Toute boucle `pour` peut être reformulée et écrite sous la forme d'une boucle `tant que`.

Conseil

Il est recommandé d'utiliser la boucle `pour` si le nombre d'itérations est connu à l'avance (avant d'entrer dans la boucle) et la boucle `tant que` lorsque le nombre d'itérations n'est pas connu.

6. Fonctions

Le rôle d'une fonction est de regrouper des instructions qui doivent être exécutées d'une manière répétitive dans ce qu'on appelle un sous programme. Une fonctions peut avoir plusieurs entrées et une sortie. Les entrées de la fonction sont les valeurs qu'on lui fournit pour qu'elle puisse les traiter et donner un résultat. La sortie est une valeur retourner par la fonction. cette valeur peut être de n'importe quelle type (types prédéfinis, tableaux ou autres structures de données).

Attention

Le type des entrées et de la sortie de la fonction doivent être définis, mais comme nous avons signalé dans les précédentes sections, nous supposons qu'on peut affecter à une variable des valeurs de différents types. Autrement dit, une variable dans notre cas (comme dans plusieurs langage de programmation, Matlab inclu) une variable peut changer de type. Nous partons du même principe pour les fonctions. Donc le type de résultat retourné par une fonction peut changer pour chaque appel.

Complément : Définir une fonction

Une fonction doit être définie avant d'être utilisée, c'est-à-dire que l'on doit indiquer son entête (nom de la fonction et ses paramètres ou entrées) et son corps (les instructions qui la composent). La syntaxe de l'entête est la suivante :

```
fonction nom_fonction(entrée1, entrée2, ........., entréen);
```

Alors que le corps est un bloc d'instructions (suite d'instructions délimitées par les deux mots clés `début` et `fin`).

Fondamental : L'instruction de retour

Les fonctions utilisent une instruction spécifique permettant de communique sa sortie à l'algorithme qui l'utilise. La sortie d'une fonction (il y en a au maximum une) est en fait une valeur (numérique ou autre) que celle-ci fournit. Il nous faudrait donc une instruction dont l'effet serait : « répondre à l'algorithme appelant la valeur qu'il demande ». Cette instruction existe, son nom est retourner. la syntaxe de cette instruction de retour est la suivante.

```
retourner expression;
```

Attention

L'instruction retourner est toujours la dernière instruction exécutée dans une fonction. Toute instruction placée après l'instruction `retourner` est ignorée.

Exemple

Une analogie avec une fonction mathématique permet d'illustrer clairement les notions d'entrées et de sorties. Soit la fonction suivante définie avec le formalisme des mathématiques :

$$\begin{aligned} F &: \mathbb{R}\times\mathbb{R}\to\mathbb{R} \\ x,y &\to x^2+y^2+2xy \end{aligned} \qquad (1)$$

```
fonction identitéRemarquable1(x, y);
début
res = x * x + y * y + 2 * x * y;
retourner res;
fin;
```

Dans l'exemple précédent, la variable `res` est utilisée pour calculer la valeur de l'expression évaluée `(x² + y² + 2xy)`. Puis l'instruction suivante retourne la valeur enregistrée dans la variable `res`.

Remarque : Fonction sans valeur retournée

Il est possible qu'une fonction n'ait aucune valeur à retourner. Dans ce cas il n'est pas indispensable d'utiliser l'instruction retourner. Les fonctions qui ne retournent aucune valeur sont très souvent appelées des *procédures.*

6.1. Appels de fonctions

L'appel d'une fonction correspond à une demande de son utilisation. Afin d'appeler une fonction, on doit préciser son nom, ainsi que les valeurs que l'on fournit pour les entrées (paramètres ou arguments). La fonction appelée est en charge de fournir la sortie. La syntaxe suivante est utilisée pour faire appel à une fonction :

```
nomFonction(param1, param2, ...., paramn);
```

Exemple : Trouver la valeur maximale dans un tableau

Pour expliquer comment marche l'appel aux fonctions nous allons utiliser un exemple ou on cherche la valeur maximale dans un tableau unidimensionnel contenant des valeurs numériques. Nous allons premièrement définir une fonction max qui prend comme paramètres deux valeurs numériques et retourne la plus grande valeur des deux. Puis nous utilisons la fonction max pour définir la fonction maxTab qui prend comme paramètres un tableau contenant des valeurs numériques et un entier indiquant la taille de ce tableau, et qui retourne comme résultat la valeur maximale dans le tableau.

Complément : Définir la fonction max

```
fonction max(v1, v2);
début
si v1 > v2 alors
   début
   retourner v1;
   fin;
sinon
   début
   retourner v2;
   fin;
fin;
```

L'idée de la fonction max est de tester si le premier argument est supérieur au deuxième avant de décider quelle valeur retourner. Si la valeur du test est vraie elle retourne le premier argument, sinon le deuxième argument est retourné.

Complément : Définir la fonction maxTab

```
fonction maxTab(T, n);
début;
```

```
vMax = T[1];
pour i allant de 2 à n faire
   début
   vMax = max(vMax, T[i]);
   fin;
retourner vMax;
fin;
```

La première instruction de la fonction `maxTab` déclare une variable `vMax` et l'initialise avec la valeur dans la première case du tableau (`T[1]`). Ensuite toutes les valeurs contenues dans le tableau `T` sont parcourues et comparées à la variable `vMax`, la valeur maximale est gardée jusqu'à la fin de la boucle et retournée en utilisant l'instruction `retourner`.

7. Tableaux

Un tableau permet de regrouper plusieurs valeurs dans une seule variable, au sein de laquelle chaque valeur sera désignée par un numéro. En d'autres termes, un ensemble de valeurs portant le même nom (ou identifiant) et repérées par un nombre (appelé indice) s'appelle un tableau.

7.1. Manipuler les tableaux

Les instructions de base manipulent les tableaux élément par élément. Pour accéder à un élément du tableau il faut utiliser l'identifiant du tableau auquel il appartient suivi par l'indice de l'élément entre crochets. Les indice du tableau sont numérotés à partir de 1 jusqu'à la taille du tableau. L'exemple suivant montre comment lire ou écrire les élément d'un tableau.

Exemple : Manipuler les éléments d'un tableau

```
lire(A[1]);      % lire l'élément avec l'indice 1 du tableau A.
écrire(A[1]);    % afficher l'élément avec l'indice 1 du tableau A.
A[2] ← A[1] * 2; % Utiliser les éléments d'un tableau dans une affectation
```

Remarque

Il ne faut pas confondre l'indice d'un élément du tableau avec la valeur de cet élément. L'indice peut être considérer comme l'adresse fixe d'un élément alors que sa valeur est accessible via cet indice et peut changer au fur et à mesure que l'algorithme avance.

7.2. Tableaux et boucles

Les boucles sont extrêmement utiles pour les algorithmes associés aux tableaux. En effet, de nombreux algorithmes nécessitent de parcourir les éléments d'un tableau dans un certain ordre, le plus souvent dans le sens des indices croissant. Le traitement de chacun des éléments étant souvent le même, seule la valeur de l'indice est amenée à changer. Une boucle est donc parfaitement adaptée à ce genre de traitements.

Chapitre 2 - Introduction à Matlab

II

1. Qu'est ce que Matlab

Le logiciel Matlab (MATtrix LABoratory) constitue un système interactif et convivial de calcul numérique et de visualisation graphique. Destiné aux ingénieurs, aux techniciens et aux scientifiques, c'est un outil très utilisé, dans les universités comme dans le monde industriel, qui intègre des centaines de fonctions mathématiques et d'analyse numérique. Matlab permet de manipuler des matrices, d'afficher des courbes et des données, de mettre en œuvre des algorithmes et de créer des interfaces utilisateur.

Matlab est constitué d'un noyau capable d'interpréter puis d'évaluer les expressions numériques matricielles qui lui sont adressées sous différentes formes :

- soit directement au clavier depuis une fenêtre de commande ;
- soit sous forme de séquences d'expressions ou scripts enregistrées dans des fichiers texte appelés m-files (ou fichiers .m) et exécutées depuis la fenêtre de commande.

Ce noyau est complété par une bibliothèque de fonctions prédéfinies, très souvent sous forme de fichiers m-files, et regroupés en paquages ou toolboxes. Il est possible d'ajouter des toolboxes spécifiques à un problème mathématique précis (exemple : Optimization Toolbox ou Signal Processing Toolbox), ou encore des toolboxes crées par l'utilisateur lui même.

2. Une session Matlab

L'interface utilisateur de Matlab varie légèrement en fonction de la version installée (nous utilisons ici la version 7.5). Cependant, elle est constituée (principalement) d'une fenêtre de commandes et un éditeur de texte permettant d'écrire des scripts et des fonctions). La figure suivante illustre les deux composants principaux de l'interface Matlab.

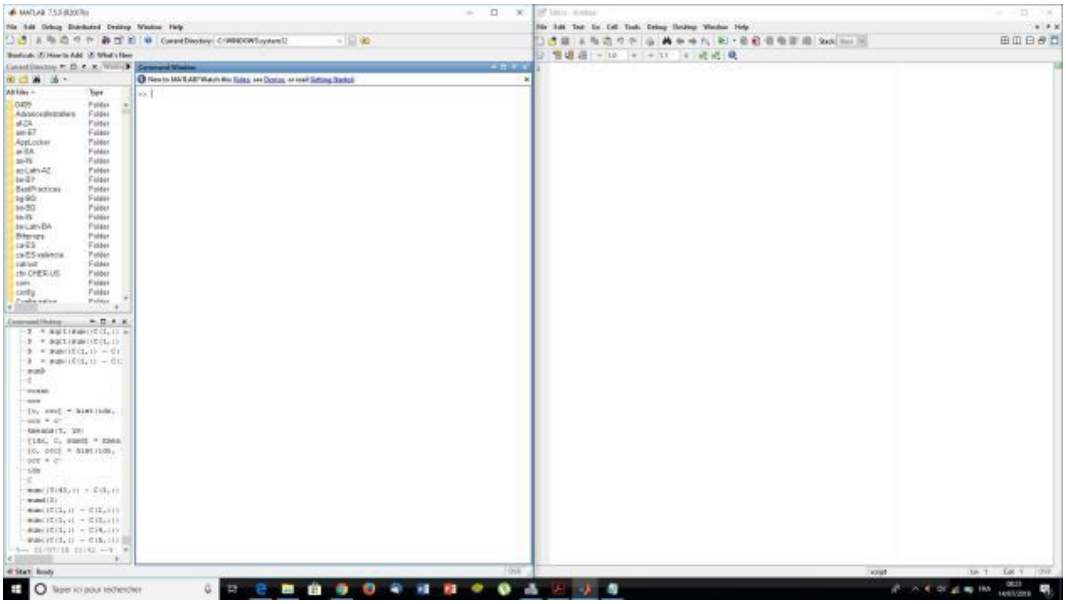

Figure 1 : l'interface utilisateur de Matlab

2.1. Lancer, quitter Matlab

Pour lancer Matlab, il suffit de double cliquer sur l'icône du logiciel. La fenêtre de commande de Matlab s'ouvre alors et on tape les commandes ou les expressions à évaluer à droite du prompt ">>" . Le processus d'évaluation est déclenché par la frappe de la touche « Enter ».

2.2. La fenêtre de commandes

La fenêtre de commandes est la fenêtre principale de l'environnement Matlab. Elle donne accès à l'interpréteur de commandes et exécute les commandes Matlab saisies par l'utilisateur. Les chevrons ">>" indiquent l'endroit ou il faut saisir la commande.

La fenêtre de commandes donne aussi accès à l'historique de toutes les commandes que vous avez tapées. Vous pouvez les retrouver et les modifier grâce aux touches de direction. Il est possible d'utiliser les touches ⇑ et ⇓ pour parcourir les commandes exécutées précédemment, puis les éditer. Pour relancer une commande, il suffit d'appuyer sur la touche « Enter » (aussi appelée retour à la ligne).

Il est aussi possible de retrouver toutes les commandes commençant par un groupe de lettres. Pour cela, il faut taper les premières lettres de la commande recherchée, puis appuyer plusieurs fois sur ⇑ pour parcourir les commandes correspondant à cette recherche.

🔎 *Remarque : Fonctions et commandes*

Certaines commandes de Matlab calculent des valeurs numériques ou vectorielles, celles là sont appelées des fonctions. Elles sont caractérisées par le fait que leurs arguments (lorsqu'ils existent) sont placés entre parenthèses. Elles se comportent de façon assez similaire aux fonctions mathématiques.

Attention

Certaines fonctions peuvent être appelées avec des paramètres qui sont différents (par leur nombre ou par leur nature). Le traitement effectué par ces fonctions dépend principalement de leurs arguments. Par exemple nous verrons plus loin que la fonction `diag` appelée avec une matrice pour argument retourne sa diagonale principale alors que lorsqu'elle est appelée avec un vecteur, elle retourne une matrice diagonale dont le vecteur diagonal est le vecteur donné comme argument.

Donc une fonction (ou une commande) n'est pas caractérisée seulement par son nom, mais par sa signature (nom + arguments).

3. Aide et documentation

Matlab dispose d'une aide fournie et fonctionnelle. Elle est accessible de différentes manières :

- aide fournie par l'environnement Matlab ;
    - la commande `helpwin` : Donne une liste de fonctions Matlab classées par thème. Par exemple, la commande "`helpwin elfun`" ouvre une fenêtre graphique qui liste les fonctions élémentaires les plus utilisées ;
    - les commandes `doc` et `help` : La commande "`help fonction`" (ou "`doc fonction`") donne la définition de la fonction indiquée et ces paramètres ainsi que quelques exemples d'utilisation. la différence entre les deux commandes est que `help` affiche ces messages dans la fenêtre de commandes alors que `doc` les affiche dans une fenêtre de navigation ;
- aide en ligne ;
    - en consultant la documentation officielle en ligne sur l'adresse *https://fr.mathworks.com/help/* ;
    - la commande `lookfor` : la commande "`lookfor sujet`" donne une liste de rubriques de l'aide en ligne en relation avec le sujet indiqué.

Exemple : La commande help

```
>> help sin
 SIN    Sine of argument in radians.
    SIN(X) is the sine of the elements of X.

    See also asin, sind.

    Overloaded methods:
       darray/sin
       sym/sin

    Reference page in Help browser
       doc sin
```

Remarque

Dans tous les exemples de ce cours, les chevrons "`>>`" seront présents pour que vous puissiez faire la différence entre la commande saisie et le résultat produit par Matlab.

```
>> Commande saisie par l'utilisateur

```

```
3   Résultat produit par Matlab
```

4. Les “objets" de Matlab - Listes, vecteurs et tableaux

Objets et classes dans Matlab

La classe principale dans Matlab est la classe `double`, c'est à partir de cette classe que sont définis des types de données plus comme : les nombres complexes et les tableaux. Une autre classe qui est moins fréquemment utilisée est la classe `char`, qui modélise des caractères (alphabets, chiffres et caractères spéciaux).

Dans les nouvelles versions de Matlab (à partir de la version 7) une classe `logical` a été introduite. Cette classe est utilisée pour modéliser des valeurs logiques : vrai ou faux (`true` ou `false`).

4.1. Types de données et valeurs littérales

Le terme *valeur littérale* désigne les valeurs (de tout type de données) qu'on peut directement saisir à partir du clavier. Les types de données qui nous intéressent dans cette section sont les suivants :

- Les types numériques.
- Les tableaux.
- Les chaînes de caractères.

4.1.1. Les types numériques

Les nombres réels ou entiers sont écrits soit sous la forme décimale (en utilisant le point "`.`"comme séparateur décimal) soit en utilisant la notation scientifique. Les nombres complexes sont écrits sous la forme "`a + bi`" tel que `a` et `b` sont des nombres réels. L'exemple suivant montre comment initialiser un nombre complexe.

Exemple : Types numériques dans Matlab

```
1 >> 5       % type entier
2
3 ans =
4
5      5
6
7 >> 1.21E33 % type réel
8
9 ans =
10
11   1.2100e+033
12
13 >> 2 + 3i  % type complexe
14
15 ans =
16
17    2.0000 + 3.0000i
```

Complément : Fonctions relatives aux nombres complexes

L'environnement Matlab fournit quelques fonctions prédéfinies qui facilitent la manipulation des nombres complexes.

- `real` : extrait la partie réelle d'un nombre complexe ;

- `imag` : extrait la partie imaginaire d'un nombre complexe ;
- `abs` : calcule le module d'un nombre complexe ;
- `conj` : renvoie le nombre complexe conjugué de son argument.

4.1.2. Les tableaux

La syntaxe utilisée par Matlab pour saisir un tableau à une ou deux dimensions (autrement dit : matrice) est la suivante :

- le tableau est délimité par deux crochets (`[]`) ;
- les éléments de la même ligne sont séparés par des espaces ou par des virgules ;
- les lignes sont séparées par des points-virgules ou des retours à la ligne ;
- toutes les lignes doivent contenir le même nombre d'éléments.

L'exemple suivant montre comment initialiser des tableaux (de différentes dimensions).

Exemple : Initialiser des tableaux dans Matlab

```
>> [1 2 3 4] % ou [1, 2, 3, 4] pour initialiser un vecteur ligne

ans =

     1     2     3     4

>> [1; 2; 3; 4 + 5i] % pour initialiser un vecteur colonne

ans =

   1.0000
   2.0000
   3.0000
   4.0000 + 5.0000i

>> [1, 2; 3, 4] % initialiser une matrice

ans =

     1     2
     3     4

>> [1, 2, 3; 4, 5] % les lignes doivent contenir le même nombre d'elements
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

Remarque : Vecteur ligne, vecteur colonne et matrice

Dans le reste de ce cours, nous allons utiliser le terme matrice pour décrire un tableau de deux dimensions, le terme vecteur ligne pour décrire un tableau qui comporte une seule ligne (dans certaines références le terme liste est utilisé) et le terme vecteur colonne pour décrire un tableau qui ne comporte qu'une seule colonne (dans certaines références, le terme vecteur "tout court" est utilisé).

a) L'opérateur colon ":"

Avant d'aller plus loin, il est nécessaire de se familiariser avec une notation que l'on retrouve partout sur Matlab, celle de l'opérateur "`:`". Pour deux nombres entiers `E1` et `E2` tels que `E2 > E1`, la commande `E1:E2` génère un vecteur ligne qui comporte tous les entiers dans intervalle `[E1, E2]`.

Pour trois nombres réels `R1`, `R2` et `x`, la commande `R1:x:R2` génère un vecteur ligne qui comporte une liste des nombres qui appartiennent à l'intervalle `[R1, R2]`, et qui s'écrivent sous la forme `R1 + n * x` tel que `n` est un entier positif (si `x` est positif, `R1` doit être inférieur à `R2`, sinon `x` est négatif, `R1` doit être supérieur à R2).

Complément : Générer un vecteur colonne

Il est possible d'utiliser le même opérateur pour générer des vecteurs colonnes en transposant le résultat obtenu avec l'opérateur apostrophe (`'`).

Exemple : Utiliser l'opérateur ":"

```
>> 1.2:0.5:3

ans =

    1.2000    1.7000    2.2000    2.7000

>> 1:3

ans =

     1     2     3

>> (1:3)'

ans =

     1
     2
     3
```

b) La fonction linspace

La fonction `linspace(`v_i`, `v_f`, n)` crée une liste de `n` nombres uniformément répartis entre les valeurs v_i et v_f.

Exemple : Utiliser la fonction linspace

```
>> linspace(1, 2, 5)

ans =

    1.0000    1.2500    1.5000    1.7500    2.0000

>> linspace(1, 2, 5)'

ans =
```

```
10
11     1.0000
12     1.2500
13     1.5000
14     1.7500
15     2.0000
```

4.1.3. Les caractères et les chaînes de caractères

Une chaîne de caractères est suite finie de caractères. En phase de saisie, une chaîne de caractères est écrite entre des apostrophes (**'**). Matlab ne fait pas la différence entre une chaîne de caractères écrite entre deux apostrophes et un tableau de caractères. En d'autres termes, Matlab considère chaque chaîne de caractère comme un vecteur ligne dont chaque élément contient un caractère.

Exemple : Initialiser des chaînes de caractères

```
1 >> 'test'
2
3 ans =
4
5 test
6
7 >> ['t', 'e', 's', 't']
8
9 ans =
10
11 test
```

5. Calculs élémentaires

L'utilisation la plus basique de Matlab consiste à utiliser l'interpréteur de commandes comme une calculatrice. Les opérations arithmétiques de base utilisées par Matlab sont les suivantes :

- l'addition (+) ;
- la soustraction (-) ;
- la multiplication (*) ;
- la division (/) ;
- la puissance (^).

Dans le cas d'une expression arithmétique compliquée (qui comporte plusieurs opérations) la priorité la plus élevée est donnée à l'opération de puissance. La division et la multiplication sont de leur côté plus prioritaire que l'addition et la soustraction. Notez aussi que les parenthèses peuvent s'utiliser de manière classique pour tenir compte des priorités des opérations.

Exemple : Effectuer des calculs élémentaires

```
1 >> 3 + 2 * (3 + 2)
2
3 ans =
4
5     13
6
7 >> (3 * 2) ^ 3 + 2
8
```

```
ans =

   218
```

Remarque : La variable spéciale "ans"

Le résultat est affecté automatiquement à une variable spéciale appelée ans. Celle-ci pourrait être utilisée dans un autre calcul comme nous allons le voir dans les prochains exemples.

5.1. Fonctions classiques

Dans Matlab, de nombreuses fonctions de calcul sont prédéfinies :

- `sqrt()` : calcule la racine carrée.
- `cos()`, `sin()`, `tan()` et `cotg()` : pour les fonctions trigonométriques de base.
- `acos()`, `asin()` et `atan()` : pour leurs réciproques.
- `cosh()`, `sinh()`, `tanh()`, `acosh()`, `asinh()` et `atanh()` : pour les fonctions trigonométriques hyperboliques .
- `log()` : pour le logarithme népérien (ln)
- `log10()` : pour le logarithme en base 10.
- `log2()` : pour le logarithme en base 2.
- `exp()` : pour la fonction exponentielle.
- `floor()` : pour l'arrondi vers l'entier inférieur.
- `ceil()` : pour l'arrondi vers l'entier supérieur.
- `fix()` : pour la troncature (arrondit à l'entier inférieur en valeur absolue).
- `round()` : pour l'arrondi à l'entier le plus proche (0,5 → 1).
- `abs()` : pour la valeur absolue.
- `rand()` : pour avoir un nombre aléatoire entre 0 et 1, suivant une loi uniforme.

Exemple : Utiliser des fonctions

```
>> 3 + 2 * (3 + 2)

ans =

    13

>> (3 * 2) ^ 3 + 2

ans =

   218
```

5.2. Format d'affichage

Par défaut, Matlab affiche les résultats en format `short`. On peut modifier le format d'affichage n utilisant la commande `format` :

- `format short` : valeur par défaut, notation fixe à 4 décimales.
- `format long` : notation fixe à 7 décimales pour un réel au format simple, 14 ou 15 décimales pour un réel au format double.
- `format short e` ou `format long e` : notation scientifique exponentielle à virgule flottante.

6. Variables et affectation

Dans la plupart des langages, et Matlab n'échappe pas à la règle, on appelle variable un objet correspondant à une zone mémoire, et identifié par un nom. Dans cette section, nous verrons comment créer une variable, lui donner une valeur et l'utiliser dans Matlab. Les variables dans Matlab ne doivent pas être déclarées, elles sont créées systématiquement à la première opération d'affectation (=). Le type et la dimension de la variable sont déterminés de manière automatique à partir de l'expression mathématique ou de la valeur affectée à la variable.

Une variable est désignée par un identificateur qui est formé d'une combinaison de lettres et de chiffres. Le premier caractère de l'identificateur doit nécessairement être une lettre. Cet identificateur doit être unique pour distinguer les différentes variables utilisées par le programme.

Cette notion (variable) est fondamentale parce qu'elle permet de manipuler des valeurs (numériques ou non) et est à la base de la plupart des calculs et traitements que l'on peut réaliser avec un langage de programmation.

Exemple : Utiliser des variables

```
>> x = 2

x =

     2

>> y = x * 2

y =

     4
```

Attention : Majuscules et minuscules dans un identificateur

MATLAB différencie majuscules et minuscules, c'est-à-dire que la variable avec l'identifiant `x` est différente de la variable `X` comme le montre l'exemple suivant.

Exemple : Différence entre majuscules et minuscules

```
>> x = 2

x =

     2

>> X = 3

X =

     3

>> y = x + 2

y =

     3

```

```
>> z = Y + 5
??? Undefined function or variable 'Y'.
```

6.1. Quelques variables spéciales

Certains identificateurs sont réservés par Matlab pour conserver quelques valeurs importantes, nous présentons ici certaines de ces variables :

- La variable `ans` : cette variable garde la valeur de la dernière expression non-affecté à une variable.
- La variable `pi` : cette variable comporte la valeur de π (= 3.146...).
- La variable `i` : cette variable garde le nombre imaginaire $\sqrt{-1}$

Exemple : Utiliser des variables

```
>> x = i ^ 2

x =

    -1

>> y = pi * 2

y =

    6.2832
```

7. Manipuler des tableaux

Matlab est un outil de calcul matriciel, alors il considère chaque nombre comme une matrice de taille (`1 x 1`), un vecteur ligne de `n` éléments comme une matrice de taille (`1 x n`) et un vecteur colonne de `m` éléments comme une matrice de taille (`m x 1`). Matlab est particulièrement adapté aux calculs d'algèbre linéaire.

Pour deux matrices `x` et `y`, si le nombre de lignes de `y` est égale au nombre de colonnes de `x`, alors l'opération `x * y` effectue une multiplication matricielle de `x` par `y`. L'opérateur `^` peut également être utilisé pour multiplier une matrice carrée par elle même.

La compréhension de la gestion des matrices (tableaux à deux dimensions) par Matlab est une étape essentielle dans la prise en main de ce langage. En effet, Matlab est avant tout un logiciel de calcul matriciel et donc, maîtriser la manipulation des matrices, permet d'améliorer les performances des programmes par un codage propre et efficace.

Exemple : Opération de multiplication et matrices

```
>> a = [1 2; 3 4]

a =

     1     2
     3     4

>> b = [5 6; 7 8]

b =

```

```
     5     6
     7     8

>> a * b

ans =

    19    22
    43    50

>> a .* b

ans =

     5    12
    21    32
```

L'opérateur "+" est utilisé pour faire l'addition de deux matrices qui ont la même dimension, ou pour additionner un scalaire à tous les composants d'une matrice. L'opérateur "-" peut être utilisé de la même façon pour effectuer des soustractions.

Exemple : Addition et soustraction des matrices

```
>> a = [1 2; 3 4];
>> b = [5 6; 7 8];
>> b + a

ans =

     6     8
    10    12

>> b - a

ans =

     4     4
     4     4

>> b - 1

ans =

     4     5
     6     7
```

Conseil : Le point-virgule

Comme le montre l'exemple précédent, seulement les valeurs des expressions qui ne sont pas suivies d'un point-virgule sont affichées. Lorsqu'on met un point-virgule à la fin d'une instruction le résultat de son évaluation n'est pas affiché (l'instruction est exécutée d'une façon normale, c'est seulement l'affichage qui n'est pas effectué). Vu que dans certains cas l'affichage des résultats peut s'avérer fastidieux et inutile (comme le résultat de multiplication de deux grandes matrices) nous recommandons de mettre un point-virgule à la fin de ces commandes pour indiquer à Matlab qu'il ne doit pas afficher les résultats.

7.1. Accéder aux éléments d'un tableau

Les éléments d'un tableau peuvent être manipulés grâce à leurs indices. Soit `T` un tableau de dimension *(L x C)*, et soit *l* et *c* deux nombres entiers positifs tel que $l < L$ et $c < C$. La commande `T(l, c)` nous permet d'accéder à l'élément à la c_{ieme} colonne de la l_{ieme} ligne du tableau `T`.

Pour un vecteur ligne (ou colonne) `v` de taille *n*, et pour un nombre $i < n$ la commande `v(i)` permet d'accéder à l'élément dans la ligne (ou la colonne) *i* du vecteur *v*.

Exemple : Accéder aux éléments d'un tableau

```
>> A = 2:2:10

A =

     2     4     6     8    10

>> A(3)

ans =

     6

>> k = 4

k =

     4

>> A(k)

ans =

     8

>> B = [2 4; 6 8]

B =

     2     4
     6     8

>> B(1, 2)

ans =

     4

>> B(2, 2)

ans =

     8
```

7.2. Extraire un sous-tableau

Soit `T` un tableau de dimension *(L x C)*, et soit *l1 < l2 < L*, *c1 < c2 < C*. La commande `T(:, c1)` extrait la colonne *c1* du tableau `T`, alors que la commande `T(l1, :)` extrait la ligne *l1*. La commande `T(l1:l2, c1:c2)` extrait un sous-tableau formé par les éléments de `T` dont l'indice de ligne appartient à `[l1, l2]` et l'indice de colonne appartient à `[c1, c2]`. L'identificateur `end` peut être utilisé pour accéder à la dernière ligne ou la dernière colonne d'un tableau.

Exemple : Extraction des sous-tableaux

```
>> A = [1 2 3 4; 2 3 4 5; 3 4 5 6; 4 5 6 7]

A =

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

>> A(:, 3)

ans =

     3
     4
     5
     6

>> A(2, :)

ans =

     2     3     4     5

>> A(3:end, 2:3)

ans =

     4     5
     5     6
```

7.3. Sous-matrices spéciales

Pour extraire les éléments de la diagonale principale d'une matrice, on utilise la fonction `diag`. La même fonction peut être paramétrée pour extraire les éléments de la $k^{ième}$ diagonale. La fonction `diag` retourne un vecteur-colonne formé des éléments de la diagonale qu'elle extrait. Selon le type de son argument, la fonction `diag` se comporte de deux façon différentes.

- si le paramètre de la fonction `diag` est une matrice elle retourne un vecteur-colonne contenant les éléments de la diagonale de son paramètre.
- si le paramètre de la fonction `diag` est un vecteur, elle retourne une matrice diagonale dont le vecteur diagonal est composé à partir de son paramètre.

La fonction `triu` (`tril`) extrait les éléments sur et au dessus (dessous) de la diagonale. Comme `diag`, les fonctions `triu` et `tril` peuvent prendre un deuxième paramètre pour extraire les éléments à partir de la diagonale `k`. Notez que les fonctions `diag`, `triu` et `tril` peuvent prendre comme argument une matrice qui n'est pas carrée.

Exemple

```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> triu(A)

ans =

     1     2     3
     0     5     6
     0     0     9

>> diag(A, 1)

ans =

     2
     6

>> diag(diag(A))

ans =

     1     0     0
     0     5     0
     0     0     9
```

7.4. Concaténer des tableaux

Nous utilisons l'opérateur `[]` pour concaténer deux tableaux (ou plus). on sépare les tableaux par un espace ou virgule pour faire une concaténation en colonnes (les tableaux doivent avoir le même nombre de lignes), et on les sépare par des points-virgules pour faire une concaténation en lignes (les tableaux doivent avoir le même nombre de colonnes).

Pour concaténer un tableau à lui-même (construire un tableau en répétant le même bloque), la fonction `repmat` peut-être utilisée.

Il peut arriver que l'on souhaite obtenir la réplication d'une matrice de façon entrelacée en répétant un certaine nombre de fois chaque colonne et/ou chaque ligne. Dans ce cas, il faut utiliser la fonction `kron`.

Exemple

```
>> A = [1 2];
```

```
>> B = [3 4; 5 6];
>> [A; B]

ans =

     1     2
     3     4
     5     6

>> repmat(A, 2, 2)

ans =

     1     2     1     2
     1     2     1     2

>> kron(B, ones(2, 3))

ans =

     3     3     3     4     4     4
     3     3     3     4     4     4
     5     5     5     6     6     6
     5     5     5     6     6     6

>> kron(B, [1 2; 3 4]) % multiplier chaque valeur par des coéfficients

ans =

     3     6     4     8
     9    12    12    16
     5    10     6    12
    15    20    18    24

```

7.5. L'indice "end"

`end` est un mot-clé de Matlab qui peut être employé comme opérateur d'indexage. Dans le cas d'un vecteur, le dernier élément est retourné, alors que dans le cas d'une matrice (ou un tableau à plusieurs dimensions) il retourne le dernier élément d'une de ces dimensions.

Exemple : Utilisation de l'indice "end"

```
>> M = [4 7 2]

M =

     4     7     2

>> M(end)

ans =

     2

>> X = [8 1 ; 2 7]

```

```
X =

     8     1
     2     7

>> X(end,1) % Dernier élément de la première colonne

ans =

     2

>> X(end,2) % Dernier élément de la seconde colonne

ans =

     7

>> X(1,end) % Dernier élément de la première ligne

ans =

     1

>> X(2,end) % Dernier élément de la seconde ligne

ans =

     7

>> X(end,end) % Dernier élément de la matrice (indexage classique)

ans =

     7

>> X(end) % Dernier élément de la matrice (indexage linéaire)

ans =

     7

>> X(end,1:end) % Tous les éléments de la derniere ligne

ans =

     2     7
```

⚠ *Attention : end, un mot clé à usage multiple*

En effet, le mot clé `end` peut également être utilisé pour marquer la fin d'une structure de contrôle dans un script ou une fonction. plus de détails serons donnés dans le troisième chapitre.

7.6. Initialiser des matrices spéciales

Les fonctions suivantes construisent des matrices usuelles: identité, Hilbert ...

Fonction	Matrice à générer
`eye`	prend comme paramètre un entier positif *n* et retourne une matrice identité carrée d'ordre *n*.
`ones`	prend comme paramètre un entier positif *n* et retourne une matrice carrée *M* d'ordre *n* tel que $M_{i,j} = 1$.
`zeros`	prend comme paramètre un entier positif *n* et retourne une matrice carrée *M* d'ordre n tel que $M_{i,j} = 0$.
`rand`	prend comme paramètre un entier positif *n* et génère une matrice carrée aléatoire d'ordre *n* (comporte des valeurs aléatoires entre 0 et 1).
`vander`	prend comme paramètre un vecteur (ligne ou colonne) `s` et retourne une matrice de Vandermonde de taille `length(s)` engendrée par le vecteur `s`.
`hilb`	prend comme paramètre un entier positif `n` et génère une matrice `M` de Hilbert d'ordre *n* ($m_{i,j} = 1 / (i + j - 1)$).
`magic`	prend comme paramètre un entier *n* (> 3) et génère un carré magique d'ordre *n* (construit à partir d'entiers entre 1 et n^2, tel que les sommes des lignes, des colonnes et des deux vecteurs diagonaux de la matrice générée sont égales).
`pascal`	prend comme paramètre un entier positif *n* et génère une matrice *M* de Pascal d'ordre *n*. Pour tout $i, j \leq n$: $M_{i,1} = 1$, $M_{1,j} = 1$ et $M_{i,j} = M_{i-1,j} + M_{i-1,j-1}$.
`wilkinson`	prend comme paramètre un entier positif *n* et génère une matrice de Wilkinson d'ordre *n*.
`hadamard`	prend comme paramètre un entier $n = 2k$ et génère une matrice *M* de Hadamard d'ordre *n* (tous les éléments de *M* sont soit 1, soit -1 et `M' * M = n * eye(n)`).

Tableau 1 : Quelques matrices spéciales

7.7. Fonctions sur les tableaux

Les fonctions présentées dans cette section effectuent des opérations arithmétiques itérativement sur les éléments d'un vecteur (ligne ou colonne). Appliquées à une matrice, elles effectuent les mêmes opérations sur chaque colonne et retournent un vecteur ligne comme résultat.

Fonction	Opération à effectuer
`length`	Retourne la taille d'un vecteur ligne ou un vecteur colonne.
`size`	Une fonction qui peut être utilisée pour retourner le nombre de lignes ou/et le nombre de colonnes d'une matrice selon les arguments qui sont utilisés : - `size(X, 1)` retourne le nombre de lignes de la matrice `X`. - `size(X, 2)` retourne le nombre de colonnes de la matrice `X`.

	- `size(X)` retourne un tableau contenant deux éléments, le premier élément représente le nombre de ligne de `X`, alors que le deuxième élément représente le nombre de colonnes `X`.

`sum`	- Pour un vecteur *T*, `sum(T)` retourne la somme des éléments de *T*.
	- Pour une matrice *M* :
	  - `sum(M, 1)` retourne un vecteur ligne contenant la somme de chaque colonne de *M*.
	  - `sum(M, 2)` retourne un vecteur colonne contenant la somme de chaque ligne de *M*.
	  - `sum(M)` : équivalent à `sum(M, 1)`.

`prod`	- Pour un vecteur *T*, `prod(T)` retourne le produit des éléments de *T*.
	- Pour une matrice *M* :
	  - `prod(M, 1)` retourne un vecteur ligne contenant le produit des éléments de chaque colonne de *M*.
	  - `prod(M, 2)` retourne un vecteur colonne contenant le produit des éléments de chaque ligne de *M*.
	  - `prod(M)` : équivalent à `prod(M, 1)`.

`max`	- Pour un vecteur *T*, `max(T)` retourne la maximum des éléments de *T*.
	- Pour une matrice *M* :
	  - `max(M, 1)` retourne un vecteur ligne contenant le maximum de chaque colonne de *M*.
	  - `max(M, 2)` retourne un vecteur colonne contenant le maximum de chaque ligne de *M*.
	  - `max(M)` : équivalent à `max(M, 1)`.

`min`	Usage similaire à la fonction `max`, mais retourne le(s) minimume(s).
`mean`	Usage similaire à la fonction `sum`, mais retourne la/les moyenne(s).
`cov`	- Appliquée à un vecteur, cette fonction retourne la variance de ces éléments. - Appliquée à une matrice *M*, elle retourne une autre matrice symétrique de covariance des éléments de *M* (colonne à colonne). Pour obtenir la variance de chaque colonne, il suffit d'extraire le vecteur diagonal du résultat.
`abs`	Retourne une matrice contenant les valeur absolues de son argument.
`norm`	Calcul les normes vectorielles ou matricielles usuelles ($\|\|.\|\|_1$, $\|\|.\|\|_2$, $\|\|.\|\|_\infty$ ... etc.).
`sort`	- Appliquée à un vecteur, elle ordonne ces éléments par ordre croissant. - Appliquée à une matrice *M*, selon le deuxième paramètre, elle ordonne les lignes ou bien les colonnes de la matrice. - `sort(M, 1)` : Ordonne toutes les colonnes de *M*.

	- `sort(M, 2)` : Ordonne toutes les lignes de *M*.

Tableau 2 : Fonctions sur les tableaux

8. Espace de travail

L'ensemble des variables initialisées par l'utilisateur constituent l'espace de travail ou la session en cours. Le contenu de cet espace de travail va changer tout au long du déroulement de la session (exécution des commandes ou scripts). Plusieurs commandes ou fonctions permettent de gérer l'espace de travail d'une façon efficace.

8.1. Informations sur l'espace de travail

Pour obtenir des informations détaillées sur des variables dans l'espace de travail, on utilise les instructions suivantes :

- La commande `who` : Afficher la liste des variables dans l'espace de travail (les identifiants seulement sans les détails).
- La commande `whos` : Affichage plus détaillé des variables dans l'espace de travail (en donnant des informations sur le type, la dimension et l'espace mémoire occupé par chaque variable).

Pour supprimer toutes les variables de l'espace de travail, la commande `clear` peut être utilisée. Pour supprimer une variable `x` spécifique, il faut exécuter la commande `clear x`.

Exemple : Utilisation des commandes : who, whos et clear

```
>> x = 2;
>> y = [3 4 5];
>> who

Your variables are:

x  y

>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
  y         1x3                24  double

>> clear x
>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double

```

8.2. Les commandes save et load

La commande `save` permet d'enregistrer toutes les variables dans un fichier `matlab.mat` (appelé aussi mat-file). Pour spécifier le nom de fichier à enregistrer, il faut utiliser la commande `save nomfichier`. Pour enregistrer quelques variables seulement, il faut écrire la commande suivante :

`save nomFichier var1 var2 ... varn`.

La commande `load` permet d'ajouter le contenu d'un fichier ".mat" à l'espace de travail courant en utilisant la syntaxe suivante :

`load nomFichier`.

9. Exercices

9.1. Exercice : Créer un vecteur

Question

[solution n°1 p.74*]*

Écrire la commande Matlab qui permet d'initialiser un vecteur ligne contenant les valeurs suivantes (1, 2, 3, 4, 5).

9.2. Exercice : Produit scalaire

Question

[solution n°2 p.74*]*

- Écrire la commande Matlab qui permet d'initialiser un vecteur ligne contenant 5 valeurs aléatoires et mettre le résultat dans la variable `x`.
- Écrire la commande Matlab qui permet d'initialiser un vecteur colonne contenant 5 valeurs aléatoires et mettre le résultat dans la variable `y`.
- Écrire la commande Matlab permettant d'effectuer le produit scalaire des deux vecteurs `x` et `y`.

9.3. Exercice : Résoudre un système linéaire

Question

[solution n°3 p.74*]*

Utiliser Matlab pour résoudre le système linéaire suivant

$$\begin{cases} 2x_1 - 2x_2 + 4x_3 &= 6 \\ x_1 + x_3 &= 7 \\ 4x_1 + x_2 - 5x_3 &= 8 \end{cases}$$

Indice :

Utiliser la représentation matricielle d'un système linéaire

9.4. Exercice : Matrice magique

Question

[solution n°4 p.74*]*

1. Créer une matrice magique de taille 5.
2. Calculer la somme de chaque colonne de la matrice magique créée.
3. Calculer la somme de chaque ligne de la même matrice.
4. Calculer la somme des deux diagonaux.

9.5. Exercice : Fonctions trigonométriques

Question

[solution n°5 p.74*]*

- Définir un vecteur `x` contenant les valeurs [π/6, π/3, π/2].
- Calculer `y1 = sin(x)` et `y2 = cos(x)`.
- Calculer `tan(x)` en utilisant `y1` et `y2`.

9.6. Exercice : Manipuler les vecteurs

Question

[solution n°6 p.74*]*

- Écrire la commande Matlab qui permet de définir le vecteur ligne `x` contenant les valeurs [1, 2 ... 49, 50]
- Définir le vecteur `y` contenant les 5 premières valeurs du vecteur `x`.
- Définir le vecteur `z` contenant les 5 dernières valeurs du vecteur `x`.
- Définir le vecteur `n` contenant les éléments à indice pair du vecteur `x`.

9.7. Exercice : Manipuler les matrices

Question

[solution n°7 p.74*]*

Écrire la commande Matlab qui permet d'initialiser un vecteur ligne contenant les valeurs suivantes (1, 2, 3, 4, 5).

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, B = \begin{bmatrix} A & 0 \\ 0 & A \end{bmatrix}$$

Enlever de la matrice B les deux premières colonnes et la dernière ligne.

9.8. Exercice : Complément sur les matrices

Question

[solution n°8 p.75*]*

1. Créer une matrice aléatoire carrée d'ordre 8, contenant des valeurs entre 1 et 10.
2. Utiliser la fonction `tril` pour extraire la partie triangulaire inférieure de la matrice générée.
3. Utiliser les deux fonctions `diag` et `prod` pour calculer le déterminant de la matrice que vous venez d'extraire.

Indice :

Pour calculer le déterminant d'une matrice triangulaire, il suffit de multiplier ses éléments diagonaux.

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 `==`.

Chapitre 4 - Matlab et analyse numérique

IV

Dans ce chapitre nous abordons les notions centrales de l'analyse numérique. En effet, beaucoup de méthodes numériques conduisent à la résolution d'un système d'équations (linéaire ou non linéaire). Quelques méthodes de résolution des équations sont détaillées ici, ainsi que des méthodes d'intégration et manipulation des polynômes.

1. Fonctions "numériques"

Les fonctions numériques de Matlab généralisent les fonctions numériques usuelles, avec une différence cependant, elles sont vectorisées. C'est à dire qu'elles s'appliquent aussi bien à des nombres qu'à des tableaux. Lorsqu'une de ces fonctions à pour argument un tableau, la fonction est appliquée à chacun de ces éléments. Le résultat obtenu est donc un tableau du même format que le tableau donné comme argument.

2. Polynômes

Sur Matlab, les vecteurs ligne sont utilisés pour représenter les polynômes, en effet, un polynôme étant déterminé par ses coefficients, il suffit de stocker ceux-ci. Le coefficient du monôme de degré 0 dans la première coordonnée du vecteur et le coefficient du monôme de plus haut degré dans la dernière. Par exemple pour représenter le polynôme `P(x) = 2x² - 3x + 1` on utilise le vecteur `[2 -3 1]`.

2.1. Manipuler les polynômes

La fonction la plus utilisée sur les polynômes est évidemment celle qui permet d'évaluer la valeur d'un polynôme `P` pour une valeur `x` donnée. La fonction `polyval` prend comme arguments un polynôme `p` (sous forme d'un vecteur) et un nombre `x`, et fournit comme résultat la valeur `p(x)`.

Exemple : Évaluer un polynôme

Dans cet exemple nous allons évaluer la valeur du polynôme `P(x) = 2x² - 3x + 1` pour `x = 2`.

```
>> P = [2 -3 1];
>> polyval(P, 2)

ans =

     3
```

Complément

D'autres fonctions qui sont assez répandues dans Matlab pour manipuler les polynômes sont listées dans le tableau suivant :

Fonction	Opération à effectuer
`poly(r)`	Créer un polynôme à partir de ces racines (dans le vecteur `r` passé comme argument).
`polyfit(x, y, n)`	Retourner un polynôme `p` de degré `n` donnant la meilleurs approximation (en méthode des moindres carrés) pour les points définis par les vecteurs `x` et `y`.
`roots(p)`	Retourner les racines du polynôme `p` passé comme argument.
`conv(p, q)`	Effectuer une multiplication de `p` par `q`.
`deconv(p, q)`	Effectue une division euclidienne du polynôme `p` par `q`. Il est aussi possible de demander deux sorties à la fonction `deconv` pour calculer aussi le résidu de la division polynomiale. En d'autres termes, la commande `[u, v] = deconv (p, q)` donne deux polynômes `u` et `v` tel que : `p = conv(u, q) + v`.
`polyint(p)`	Retourne l'intégrale du polynôme `p` (en utilisant la valeur 0 comme constante d'intégration). Il est aussi possible de spécifier la constante d'intégration `k` en la passant comme deuxième argument de la fonction : (`polyint(p, k)`).
`polyder(p)`	Retourner le polynôme dérivé de `p`.

Tableau 4 : Les fonctions les plus utilisées sur les polynômes

Exemple

Nous montrons dans cet exemple quelques commandes effectuant des manipulations sur les polynômes

```
>> P = [1 -1];       % P(x) = x - 1.
>> Q = [-1 2];       % Q(x) = x - 1.
>> W = conv(P, Q)    % W(x) = P(x) * Q(x).

W =

    -1     3    -2

>> r = roots(W)      % Calculer les racines de W

r =

     2
     1

>> V = poly(r)       % créer un polynôme à partir de ces racines

V =

     1    -3     2
```

```
>> U = deconv(W, P) % U(x) = W(x) / P(x)

U =

    -1     2

>> A = polyder(V)   % Calculer le polynôme A(x) = V'(x)

A =

     2    -3

>> polyint(A)       % Calculer l'intégral de A(x)

ans =

     1    -3     0

>> polyint(A, 2)    % Calculer l'intégral de A(x) avec la constante d'intégration
2

ans =

     1    -3     2
```

3. Calcul matriciel

Matlab est spécialement conçu pour manipuler des matrices. Il reconnaît et manipule les variables matricielles à coefficients réels ou complexes, denses ou creuses. Nous avons déjà montré dans le premier chapitre comment initialiser, effectuer des calculs et appliquer des fonctions sur des matrices. Nous montrons dans cette section comment effectuer des calculs matriciels plus avancés en utilisant Matlab.

3.1. Les matrices creuses

On appelle matrice creuse une matrice comportant une forte proportion de coefficients nuls. L'intérêt de telles matrices résulte principalement de la réduction de la place mémoire (on ne stocke pas les zéros) et aussi de la réduction du nombre d'opérations (on n'effectuera pas les opérations portant sur les zéros). Par défaut dans MATLAB une matrice est considérée comme pleine, c'est-à-dire que tous ses coefficients sont mémorisés. Si `M` est une matrice, la commande `sparse(M)` permet d'obtenir la même matrice mais stockée sous la forme creuse. Si l'on a une matrice stockée sous la forme creuse, on peut obtenir la même matrice, stockée sous la forme ordinaire par la commande `full`.

Exemple : Avantage des matrices creuses

Dans cet exemple nous allons utiliser la commande `whos` pour avoir des informations sur l'espace mémoire occupé par chaque variable, puis nous comparons l'espace mémoire occupé par des matrices creuses stockées sous différents formats.

```
>> A = diag(1:10)

A =

     1     0     0     0     0     0     0     0     0     0
```

```
     0     2     0     0     0     0     0     0     0     0
     0     0     3     0     0     0     0     0     0     0
     0     0     0     4     0     0     0     0     0     0
     0     0     0     0     5     0     0     0     0     0
     0     0     0     0     0     6     0     0     0     0
     0     0     0     0     0     0     7     0     0     0
     0     0     0     0     0     0     0     8     0     0
     0     0     0     0     0     0     0     0     9     0
     0     0     0     0     0     0     0     0     0    10

>> B = sparse(A)

B =

   (1,1)         1
   (2,2)         2
   (3,3)         3
   (4,4)         4
   (5,5)         5
   (6,6)         6
   (7,7)         7
   (8,8)         8
   (9,9)         9
  (10,10)       10

>> infoA = whos('A'); % Récupérer des informations sur la variable A
>> infoA.bytes        % Afficher le nombre de bytes (octets) occupés par A

ans =

   800

>> infoB = whos('B'); % Récupérer des informations sur la variable B
>> infoB.bytes        % Afficher le nombre de bytes (octets) occupés par B

ans =

   164
```

3.2. Valeurs et vecteurs propres d'une matrice

Soit A une matrice carrée d'ordre n. On dit que x est un vecteur propre de la matrice A pour la valeur δ si Ax = x. Pour trouver les valeurs propres de la matrice A, il faut résoudre le système $Ax=\lambda x \iff (A-\lambda I_n)x=0$. Avec I_n une matrice identité de taille *n* (matrice carrée de taille *n* dont tous les éléments de la diagonale valent "1" et tous les autres éléments valent "0").

À l'exclusion de la solution triviale x = 0, le système ci-dessus admet des solutions si le déterminant de $(A-\lambda I_n)$ vaut 0.

Complément : Calculer le vecteur propre associé à chaque valeur propre

Un vecteur x est associé comme vecteur propre à la valeur λ s'il vérifie la relation $(A-\lambda I_n)x=0$.

Complément : Calculer les vecteurs et les valeurs propres sur Matlab

Pour calculer les valeurs propres d'une matrice carrée `A` nous utilisons la fonction `eig(A)` qui retourne un vecteur colonne contenant toutes les valeurs propres de `A`. Il est aussi possible de calculer les valeurs et les vecteurs propres de `A` en demandant deux sorties à la fonction `eig` (`[vec, val] = eig(A)`). la deuxième sortie de la fonction `eig` dans ce cas donne une matrice diagonale contenant les valeurs propre de `A`, alors que la première sortie donne une matrice telle que chaque colonne est un vecteur propres de `A`.

Plus formellement, la commande `[vec, val] = eig(A)` assure que `A * vec = vec * val`.

Exemple : Utilisation de la fonction eig

Nous montrons dans cet exemple comment utiliser la fonction `eig` de Matlab pour calculer les vecteurs et valeurs propres d'une matrice

```
>> A = [5 -3; 6 -4];
>> [vec, val] = eig(A)

vec =

    0.7071    0.4472
    0.7071    0.8944


val =

     2     0
     0    -1

>> A * vec

ans =

    1.4142   -0.4472
    1.4142   -0.8944

>> vec * val

ans =

    1.4142   -0.4472
    1.4142   -0.8944
```

3.3. Méthodes itératives pour la résolution des systèmes linéaires

Un système d'équations linéaires est un ensemble d'équations portant sur les mêmes inconnues. Nous considérons ici que les systèmes contenant autant d'équations que de variables. En général, un système de m équations linéaires à n inconnues peut être écrit sous la forme suivante :

$$\begin{cases} a_{11}x_1 - a_{12}x_2 + ... + a_{1n}x_n & = b_1 \\ a_{21}x_1 - a_{22}x_2 + ... + a_{2n}x_n & = b_2 \\ \cdots & . \\ \cdots & . \\ \cdots & . \\ a_{n1}x_1 - a_{n2}x_2 + ... + a_{nn}x_n & = b_n \end{cases}$$

Où $x_1, x_2 ... x_n$ sont des inconnus.

Un système d'équations linéaires peut aussi s'écrire sous la forme matricielle : $Ax = b$ tel que :

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} et\ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Si la matrice A est inversible, il est possible de calculer $x = A^{-1} * b$. Plusieurs méthodes directes permettant de résoudre un système linéaire existent : Décomposition LU, Gauss (avec ou sans pivot), calculer l'inverse de A ... etc. Dans cette section nous nous intéressons plus aux méthodes itératives.

L'idée derrière les méthodes itératives est de construire une suite de vecteur $x^{(k)}$ qui converge vers le vecteur x (la solution du système $Ax = b$). Plus formellement $\lim_{k\to\infty} x^{(k)} = x$.

L'intérêt principal des méthodes itératives est d'être plus rapide en terme de temps de calcul (particulièrement dans les cas ou le point de départ x^0 est bien choisi). Nous présentons ici deux méthodes itératives pour la résolution des systèmes linéaires : la méthode de Jacobi et la méthode de Gauss-Seidel.

3.3.1. Méthode de Jacobi

Soit un système linéaire $Ax = b$. Si les éléments diagonaux de A sont non nuls (il est possible de permuter les lignes de A pour obtenir un diagonal qui ne comporte pas de zéro), alors le système est équivalent à :

$$x_i = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j \right) \qquad (2)$$

Pour un vecteur $x^{(k)}$ connu, on peut calculer $x^{(k+1)}$ avec la formule suivante. $x^{(0)}$ est appelée "solution devinée", elle peut être donnée par l'utilisateur, initialisée par des valeurs aléatoires ou autre. Généralement, on arrête les itérations lorsque :

- $b - Ax^{(k)} < \varepsilon$, qui signifie que la solution $x^{(k)}$ est assez précise ;
- Ou $|x^{(k+1)} - x^{(k)}| < \varepsilon$, qui signifie que les solutions ne convergent plus.

Le schéma numérique de la méthode de Jacobi est le suivant :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) pour\ k \geq 0 \qquad (3)$$

Implémentation Matlab

Nous montrons ici l'implémentation Matlab d'une fonction permettant de résoudre un système d'équations linéaires donné sous sa forme matricielle. Pour simplifier l'implémentation, nous allons décomposer la matrice A en deux sous-matrices, la matrice diagonale D contenant les éléments diagonaux de A, et R = A - D. Dans ce cas, le système précédent est équivalent à :

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \ pour \ k \geq 0 \qquad (4)$$

```
% Fonction - jacobi.m
function x = jacobi(A, b, X0)
D = diag(diag(A));                % D prend les éléments diagonaux de A
invD = D^(-1);                    % nous calculons D^-1
R = A - D;                        % R contient les éléments non diagonaux de A
Xk = X0;
while norm(b - A * Xk) > eps      % condition d'arrêt
    Xkplus1 = invD * (b - R * Xk); % calculer Xkplus1 en fonction de k
    Xk = Xkplus1;
end;
x = Xk;
```

3.3.2. Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est très similaire à celle de Jacobi, la seule différence est qu'elle propose d'introduire au fur et à mesure dans le calcul, sans attendre l'itération suivante, les composantes de $x^{(k+1)}$ qui viennent d'être calculées. Le schéma numérique de la méthode de Jacobi est le suivant :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \times \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \ pour \ k \geq 0 \qquad (5)$$

L'algorithme de Gauss-Seidel est en général plus rapide que l'algorithme de Jacobi, et donc préférable. Par contre l'algorithme de Jacobi est plus adapté aux environnements parallèles.

4. Fonction d'une variable

Nous montrons dans ce chapitre les méthodes numériques pour le calcul d'intégrale, et pour la résolution des équations non linéaires. On ne considère dans cette section que les fonctions à une seule variable.

4.1. Calcul d'intégrale

Très souvent le calcul explicite de l'intégrale, d'une fonction *f* continue sur un intervalle [a, b], définie par : $I(f) = \int_a^b f(x)\ dx$ peut être très difficile à atteindre. Par conséquent, on fait appel à des méthodes numériques afin de calculer une approximation de *I(f)*. Nous présentons ici deux méthodes numériques pour le calcule d'intégrale : méthode du point milieu et méthode du trapèze.

4.1.1. Méthode du point milieu

Soit *f* une fonction continue sur l'intervalle *[a, b]*. L'idée de cette méthode est de subdiviser l'intervalle *[a, b]* en *n* sous intervalles $I_k = [x_k, x_{k+1}]$ tel que $x_1 = a$ et $x_{n+1} = b$. Le schéma numérique de la méthode du point milieu est le suivant :

$$I(f)=\sum_{k=1}^{n}(x_{k+1}-x_k)\times f(\bar{x}_k) \qquad avec \qquad \bar{x}_k=\frac{x_{k+1}-x_k}{2} \tag{6}$$

En effet, *I(f)* représente l'aire comprise entre la courbe *y = f(x)* et l'axe des abscisses entre les droites *x = a* et *x = b*. Alors que comme le montre la figure ci-dessous, $(x_{k+1}-x_k)\times f(\bar{x}_k)$ calcule l'aire entre la droite $y=f(\bar{x}_k)$ et l'axe des abscisses sur l'intervalle I_k. Plus les intervalles I_k sont réduits (et par conséquent nombreux), plus l'approximation du calcul d'intégrale est précise.

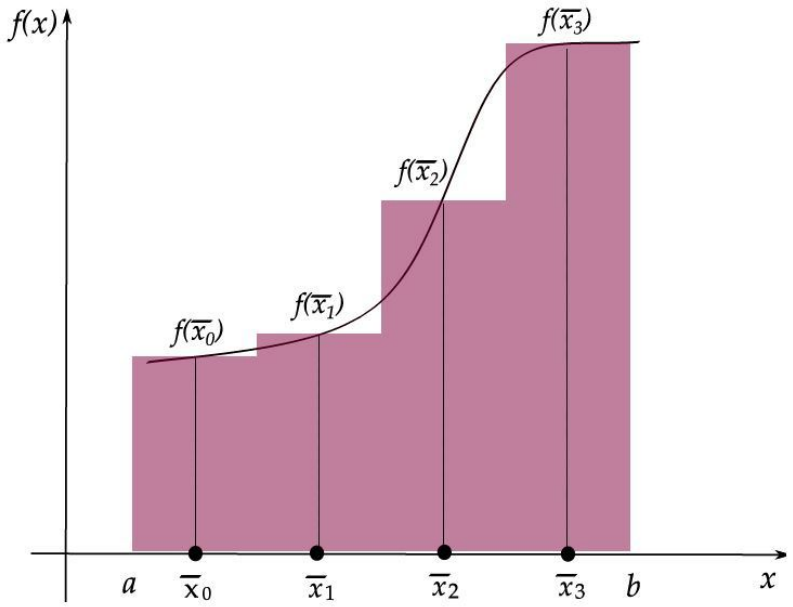

Figure 2 : Représentation de la méthode du point milieu

Si tous les intervalles I_k sont équidistants, le schéma numérique peut être simplifié comme suit :

$$I(f)=\Delta x\times\sum_{k=1}^{n}f(\bar{x}_k) \qquad avec \qquad \Delta x=\frac{b-a}{n} \tag{7}$$

Implémentation Matlab

Nous montrons ici l'implémentation Matlab d'une fonction permettant de calculer l'intégrale d'une autre fonction (passée comme paramètre) en utilisant la méthode du point milieu. Pour que notre calcul d'intégrale soit assez dynamique, il doit prendre en charge toutes les fonctions qui ont un seul paramètre `x`. pour cela, la fonction pour laquelle nous allons calculer l'intégrale est passée aussi comme paramètre, ainsi que les bornes de l'intervalle `a` et `b` (nous divisons l'intervalle *[a, b]* par la suite en 100 sous intervalles équidistants). La fonction est donnée sous forme d'une chaîne de caractères, puis elle est convertie à une fonction en appelant `str2func`.

```
% Fonction - pointMilieu.m
function res = pointMilieu(strFunc, a, b)
integFunc = str2func(strFunc);
Xk = linspace(a, b, 101);            % Générer tous les Xk distribués
uniformement sur l'intervalle [a, b]
dx = (b - a) / 100;                  % Calculer Δx
mp = (Xk(2:end) + Xk(1:end-1)) ./ 2; % Calculer les centres des sous intervalles
res = sum(integFunc(mp)) * dx;       % Calculer l'integrale
```

4.1.2. Méthode du trapèze

La méthode du trapèze consiste aussi à subdiviser l'intervalle *[a, b]* en *n* sous intervalles I_k. Cette méthode est basée sur l'interpolation linéaire de chaque intervalle. En d'autres termes, sur chaque intervalle $[x_k, x_{k+1}]$ est substitué par la droite joignant les points $(x_k, f(x_k))$ et $(x_{k+1}, f(x_{k+1}))$. Le schéma numérique de la méthode du trapèze est le suivant :

$$I(f)=\sum_{k=1}^{n}\left(\frac{f(x_k)+f(x_{k+1})}{2}\times(x_{k+1}-x_k)\right) \qquad (8)$$

La figure ci-dessous montre que $((f(x_{k+1})+f(x_k))/2)\times(x_{k+1}-x_k)$ calcule la surface de la trapézoïdale (d'où le nom trapèze) délimitée par la droite $((x_k,f(x_k)),(x_{k+1},f(x_{k+1})))$ et l'axe des abscisses sur l'intervalle I_k. Il est aussi possible d'augmenter la précision de la méthode du trapèze en réduisant la taille des intervalles I_k.

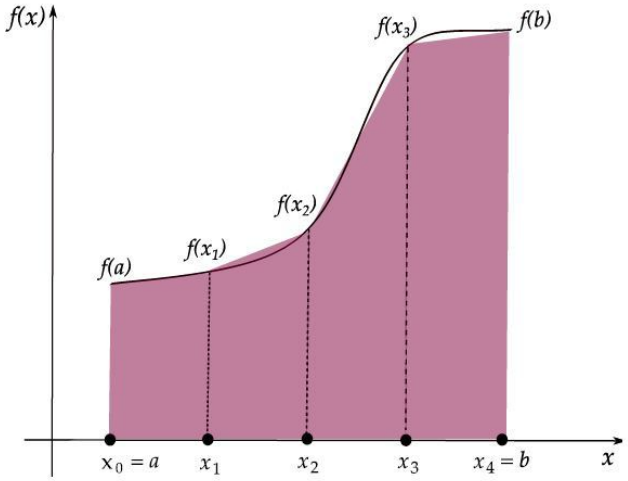

Figure 3 : Représentation de la méthode du trapèze

Si tous les intervalles I_k sont équidistants, le schéma numérique peut être simplifié comme suit :

$$\begin{aligned}
I(f) &= \Delta x\times\sum_{k=1}^{n}\frac{f(x_k)+f(x_{k+1})}{2} \qquad avec \qquad \Delta x=\frac{b-a}{n}\\
&= \frac{\Delta x}{2}\times\sum_{k=1}^{n}(f(x_k)+f(x_{k+1}))\\
&= \frac{\Delta x}{2}\times\left(\sum_{k=1}^{n}f(x_k)+\sum_{k=1}^{n}f(x_{k+1})\right)\\
&= \frac{\Delta x}{2}\times\left(\sum_{k=1}^{n}f(x_k)+\sum_{k=2}^{n+1}f(x_k)\right)\\
&= \frac{\Delta x}{2}\times\left(f(x_1)+\sum_{k=2}^{n}f(x_k)+\sum_{k=2}^{n}f(x_k)+f(x_{n+1})\right)\\
&= \frac{\Delta x}{2}\times\left(f(x_1)+f(x_{n+1})+2\sum_{k=2}^{n}f(x_k)\right)\\
&= \frac{\Delta x}{2}\times\left(f(a)+f(b)+2\sum_{k=2}^{n}f(x_k)\right)\\
&= \Delta x\times\left(\frac{f(a)+f(b)}{2}+\sum_{k=2}^{n}f(x_k)\right)
\end{aligned} \qquad (9)$$

La méthode du trapèze est prédéfinie dans l'environnement Matlab par la fonction `trapz`. Elle est utilisée selon la syntaxe suivante : `trapz(x, y)`. L'argument `x` de la fonction `trapz` contient les x_k qui définissent les subdivisions de l'intervalle sur lequel on va calculer l'intégral, alors que l'argument `y` contient les $f(x_k)$.

4.2. Résolution des équations non-linéaires

Plusieurs méthodes numériques permettent de calculer une approximation des zéros d'une fonction *f*. Deux méthodes sont illustrées dans cette section : méthode du point fixe et de la dichotomie.

4.2.1. Méthode du point fixe

Le principe de la méthode du point fixe est que de transformer l'équation $f(x) = 0$ à une équation $g(x) = x$ (généralement $g(x) = f(x) + x$). Donc, résoudre l'équation $f(x) = 0$ revient à trouver α tel que $g(\alpha) = \alpha$. Dans le cas ou α existe, il est qualifié de point fixe de la fonction $g(x)$ qui est appelée *fonction d'itération*. Lé schéma numérique de la méthode du point fixe est le suivant :

$$x^{(k+1)}=g(x^{(k)}) \qquad pour \qquad k>0 \tag{10}$$

Notez que $x^{(0)}$ est généralement choisi d'une façon aléatoire sur un intervalle *[a, b]*.

Implémentation Matlab

Nous donnons ici une fonction qui prend les trois paramètres d'entrées suivants :

- *f* : Le nom d'une fonction sous forme d'une chaîne de caractères ;
- *a* et *b* : les bornes de l'intervalle sur lequel on doit chercher la solution.

Et qui retourne comme résultat α tel que $f(\alpha) = \alpha$ en utilisant la méthode du point fixe.

```
% Fonction - pointFixe.m
function x = pointFixe(strFunc, a, b)
f = str2func(strFunc);
g = @(x) f(x) + x;          % Définir la fonction g(x) = f(x) + x
x = rand() * (b - a) + a; % Choisir un point du départ aléatoire
while x ~= g(x)           % Répeter x = g(x) jusqu'à trouver le point fixe
    x = g(x);
end
```

⚠ *Attention : Test d'égalité*

Il est important de noter que les ordinateurs n'utilisent pas explicitement les nombres réels. Ils utilisent plutôt les nombres flottants, qui sont un sous-ensemble discret de l'ensemble des nombres réels. Donc il est possible que α (tel que $g(\alpha) = \alpha$) n'appartient pas à ce sous-ensemble. Pour cela il est recommandé d'éviter les tests explicites d'égalité (ou d'inégalité), et de les remplacer par des tests qui sont assez léger pour être salifiables (et en même temps assez exigeant pour que l'algorithme reste correct).

Dans l'exemple du script précédent, il est recommandé de remplacer la condition `x ~= g(x)` par `(x - g(x)) / x < eps`.

4.2.2. Méthode de dichotomie

Le principe de cette méthode est basé sur le théorème des valeurs intermédiaires.

Rappel : Théorème des valeurs intermédiaires

Ce théorème implique que si une fonction *f* est continue sur un intervalle *[a, b]*, et que *f(a)* et *f(b)* ont différents signes (*f(a) * f(b) < 0*) alors .il existe un nombre α qui appartient à *[a, b]* tel que *f(α) = 0*.

La méthode de dichotomie suit les étapes suivantes :

- Calculer *c = (a + b) / 2*.
- Si *f(c) = 0* retourner *c* comme résultat.
- Sinon, si *signe(c) = signe(a)* (*a* et *c* ont le même signe), alors continuer le travaille récursivement sur l'intervalle *[c, b]*, sinon continuer sur l'intervalle *[a, c]*.

5. Exercices

5.1. Exercice : Évaluation d'un polynôme

Question

[solution n°17 p.77*]*

Écrire une fonction qui prend comme argument un polynôme `p` (sous forme d'un vecteur ligne) et une valeur `x` , et qui retourne comme résultat `p(x)` (vous n'avez pas le droit d'utiliser la fonction `polyval`).

Indice :

Utilisez `polyval` pour tester si les résultats fournis par votre fonction sont justes

5.2. Exercice : Dérivé d'un polynôme

Question

[solution n°18 p.77*]*

Écrire une fonction qui prend comme argument un polynôme `p`, et qui retourne comme résultat `q` le polynôme dérivé de `p` (vous n'avez pas le droit d'utiliser la fonction `polyder`).

Indice :

Utilisez `polyder` pour tester si les résultats fournis par votre fonction sont justes

5.3. Exercice : Méthode de Gauss-Seidel

Question

[solution n°19 p.77*]*

Écrire une fonction qui prend comme argument une matrice `A`, un vecteur `b`, et qui retourne comme résultat un vecteur `x` tel que `Ax = b` en utilisant la méthode de Gauss-Seidel. Il faut aussi passer comme argument la solution devinée `X0`.

5.4. Exercice : Méthode du trapèze

Question

[solution n°20 p.77*]*

Écrire une fonction qui prend les arguments :

- `f` : le nom d'une fonction sous forme d'une chaîne de caractères ;
- `a` et `b` : deux nombres.

Et qui retourne l'intégrale de la fonction *f* sur l'intervalle *[a, b]* en utilisant la méthode du trapèze (sans utiliser la fonction `trapz` de Matlab).

5.5. Exercice : Méthode de dichotomie

Question

[solution n°21 p.77*]*

Écrire une fonction qui prend les arguments :

- `f` : le nom d'une fonction sous forme d'une chaîne de caractères ;
- `a` et `b` : deux nombres tels que *f* est continue sur *[a, b]* et *f(a) * f(b) < 0*.

Et qui retourne un zéro de *f* sur *[a, b]*.

Chapitre 5 - Fenêtres graphiques

V

Matlab est un langage réputé pour la facilité d'utilisation de ses fonctions graphiques. Les vecteurs et les matrices peuvent être visualisé sous forme de courbes en 2D ou bien des courbes et surfaces en 3D ou des histogrammes. Ces graphiques peuvent être légendées et manipulées avec des commandes Matlab soit à partir d'une fenêtre de commandes, d'un script ou même directement à partir de la fenêtre graphique.

1. Les fenêtres graphiques

Pour créer une fenêtre graphique, nous utilisons la fonction Matlab `figure()` qui affiche une fenêtre vide. La valeur retournée par la fonction `figure()` est le numéro de la fenêtre (à utiliser par la suite pour afficher des composants sur cette fenêtre). Si la fonction `figure()` est appelée deux fois, on voit s'afficher deux fenêtres avec deux différents numéros comme le montre la figure 4.

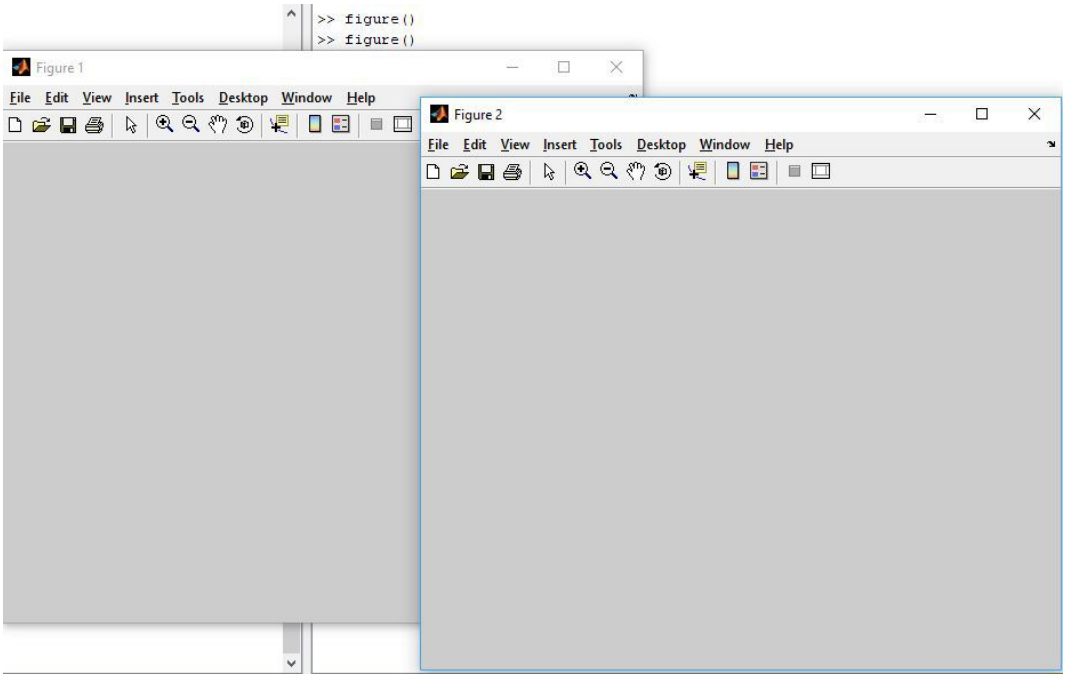

Figure 4 : L'affichage de deux fenêtres graphiques

La dernière fenêtre créée par la fonction `figure()` est celle qui est active. Pour rendre la fenêtre ayant le numéro `x` active, il faut faire appel à la fonction `figure` avec le paramètre `x` (`figure(x)`).

1.1. Courbes du plan

Soit `x` et `y` deux deux vecteurs de même longueur. La fonction `plot(x, y)` trace dans la fenêtre active le graphe de `y` en fonction `x` (si aucune fenêtre n'est active alors une fenêtre est créée automatiquement). En fait le graphe est obtenu en joignant par de petits segments de droite les points de coordonnées $(x_{(k)}, y_{(k)})$ pour (`1 k length(x)`).

Exemple : Afficher une courbe

Dans cet exemple nous allons écrire un script qui affiche la courbe de la fonction $f(x) = x^2 - x + 2$ sur l'intervalle [-2 2].

```
% Script - courbe.m
f = [1 -1 2];
x = linspace(-2, 2, 100);
y = polyval(f, x);
plot(x, y);
```

Le résultat de l'exécution de ce script est l'affichage de la fenêtre suivante :

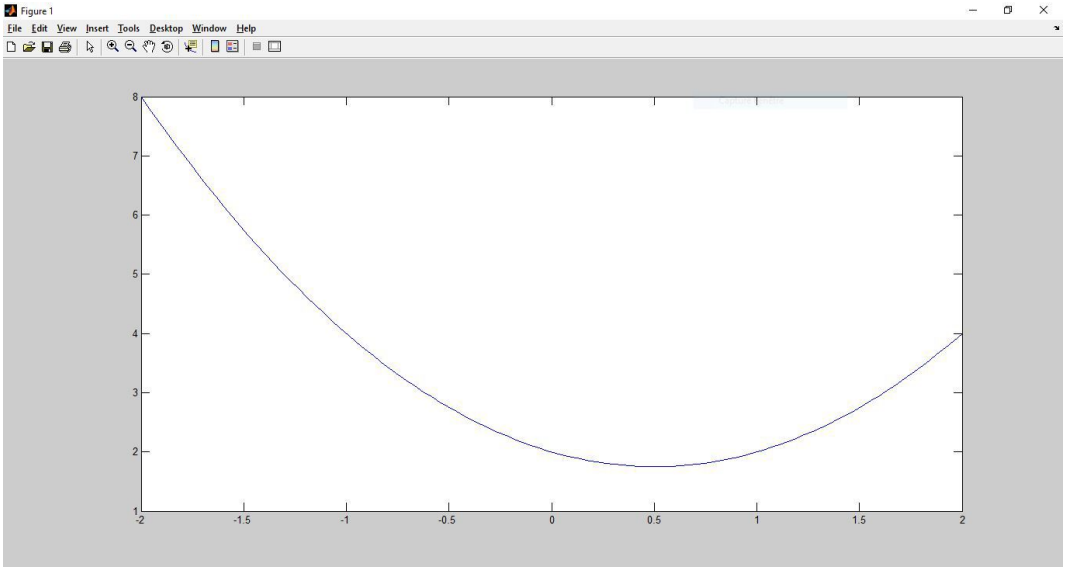

Figure 5 : Affichage de la courbe de la fonction $f(x) = x^2 - x + 2$

Matlab définit automatiquement un système d'axes. Pour personnaliser le système d'axes il faut utiliser la fonction `axis`, pour plus de détails utiliser la commande `help`.

Conseil : Afficher deux tracés

Pour tracer plusieurs courbes, deux possibilités se présentent : soit les tracer tous sur la même figure, ou bien une courbe sur chaque figure. Pour tracer plusieurs courbes sur la même figure, il faut :

1. Tracer la première courbe ;
2. Exécuter la commande `hold on` (pour plus de détails utiliser le `help` de Matlab) ;
3. Tracer les autres courbes.

Pour tracer les courbes sur des figures différentes, il faudra changer la fenêtre active avant de tracer la courbe (en utilisant la fonction `figure`).

Complément : D'autres fonctions utiles

Nous présentons ici quelques fonction utiles pour pour manipuler les courbes, les légendes, les axes ... etc. sur une figure.

Fonction	Opération à effectuer
`xlabel`	Définir une légende pour l'axe des abscisses d'une figure.
`ylabel`	Définir une légende pour l'axe des ordonnées d'une figure.
`title`	Donner un titre à la figure.
`axis`	Définir un système d'axes.
`grid`	Superposer une grille sur la figure.
`legend`	Donner une légende à chaque tracée dans la figure.

Tableau 5 : Des fonctions utiles

Quelques fonctions similaires à la fonction plot sont présentées dans le tableau ci-dessous.

Fonction	Opération à effectuer
`loglog`	Cette fonction est semblable à la fonction `plot` sauf qu'une échelle logarithmique est utilisée respectivement pour les deux axes.
`semilogx`	Fonctions semblables à la fonction `plot` avec une échelle logarithmique pour l'axe des abscisses.
`semilogy`	Semblables à `plot` avec une échelle logarithmique pour l'axe des ordonnées.
`fplot`	La fonction `fplot` prend comme arguments le nom (sous forme d'une chaîne de caractères) de la fonction dont on souhaite tracer le graphe et les valeurs des bornes de l'intervalle d'étude. L'intervalle d'étude est subdivisé par Matlab de façon à donner le tracé le plus précis possible.

Tableau 6 : Des fonctions similaires à `plot`

Exemple

Dans cet exemple nous allons tracer la courbe de la fonction $f(x) = x^2 - x + 2$ sur l'intervalle [-2 2], mais cette fois en utilisant d'autres fonctions pour mettre plus d'informations sur la figure générée.

```
% Script - courbe.m
f = [1 -1 2];
x = linspace(-2, 2, 100);
y = polyval(f, x);
plot(x, y);
title('Courbe de la fonction f(x) = x² - x + 2');
axis([-2 2 1 10]);
xlabel('axe des X');
```

```
ylabel('axe des Y');
legend('f(x) = x^2 - x + 2');
```

Le résultat d'exécution de ce script est montré dans la figure 6

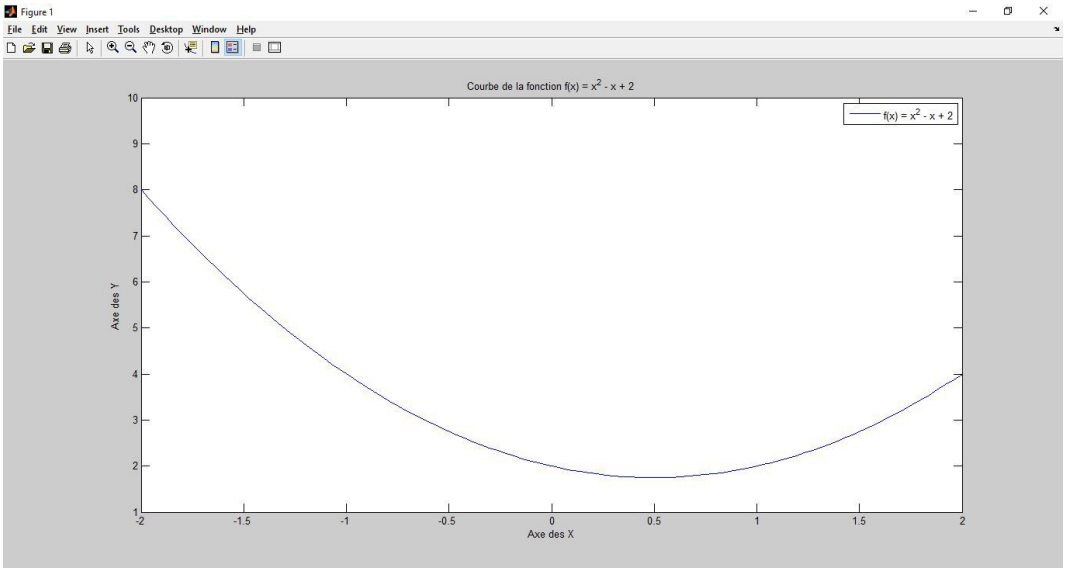

Figure 6 : Affichage de la courbe de la fonction $f(x) = x^2 - x + 2$ avec plus d'options

1.2. Courbes dans l'espace

La fonction `plot3` étend les fonctionnalités de la fonction `plot` aux courbes de l'espace. Les possibilités de personnalisation des axes, ajouter des légendes ou des titres sont les mêmes. Évidemment, pour tracer une courbe dans un espace à trois dimensions, il faut donner trois cordonnés à chaque point. Matlab donne une vue perspective du graphe de la fonction et permet de déplacer la *plotting-box* avec la souris.

Exemple

```
% Script - trace3d
x = linspace(0, 5 * pi, 500);
y = cos(x);
z = sin(x);
plot3(x, y, z);
grid on;
```

Le résultat du script précédent est montré dans la figure 7

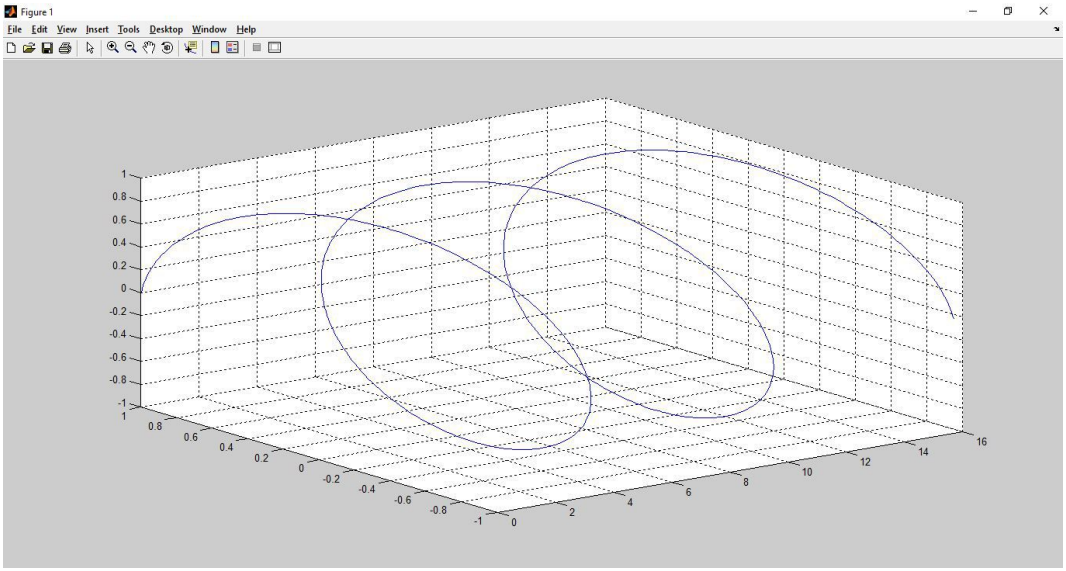

Figure 7 : Une courbe dans un espace 3D

1.3. Surfaces de l'espace

Nous montrons ici comment utiliser Matlab pour représenter des surfaces définies par une relation $z = f(x, y)$ où f est une fonction continue, définie sur un domaine $[x_0, x_1] \times [y_0, y_1]$.

Complément : Modélisation des domaines

La modélisation des deux domaines $[x_0, x_1] \times [y_0, y_1]$ passe par deux étapes :

1. Définition deux subdivisions régulières pour les deux intervalles $[x_0, x_1]$ et $[y_0, y_1]$ (en utilisant la fonction `linspace` ou l'opérateur "`:`" pour créer deux vecteurs `x` et `y`) ;
2. Construction d'une grille modélisant le domaine $[x_0, x_1] \times [y_0, y_1]$ en utilisant la commande `[xx, yy] = meshgrid(x, y)`. La grille est définie par les deux matrices `xx`, `yy` de telle sorte que `(xx(l, k), yy(l, k)) = (x(k), y(l))`.

Puis il est possible d'appliquer la fonction `f` sur le couple `(xx, yy)` pour calculer `z` (`z = f(xx, yy)`).

Complément : Tracer la surface

Une fois le domaine d'étude modélisé par les deux tableaux `xx` et `yy`, et qu'on a évalué les valeurs de la fonction pour obtenir un tableau `(z = f(xx,yy))`. On dessine la surface z = f(x, y) avec la fonction `surf(xx, yy, z)` :

Exemple

Dans cet exemple, nous allons tracer la surface représentant les images de la fonction $f(x, y) = cos(x + y)$ sur le domaine $[0, 5] \times [0, 5]$.

```
% Script - surface.m
x = linspace(0, 5, 20);
y = linspace(0, 5, 20);
f = @(x, y) cos(x + y);
[xx, yy] = meshgrid(x, y);
z = f(xx, yy);
surf(xx, yy, z);
```

Le résultat fourni par le script précédent est montré dans la figure ci-dessous

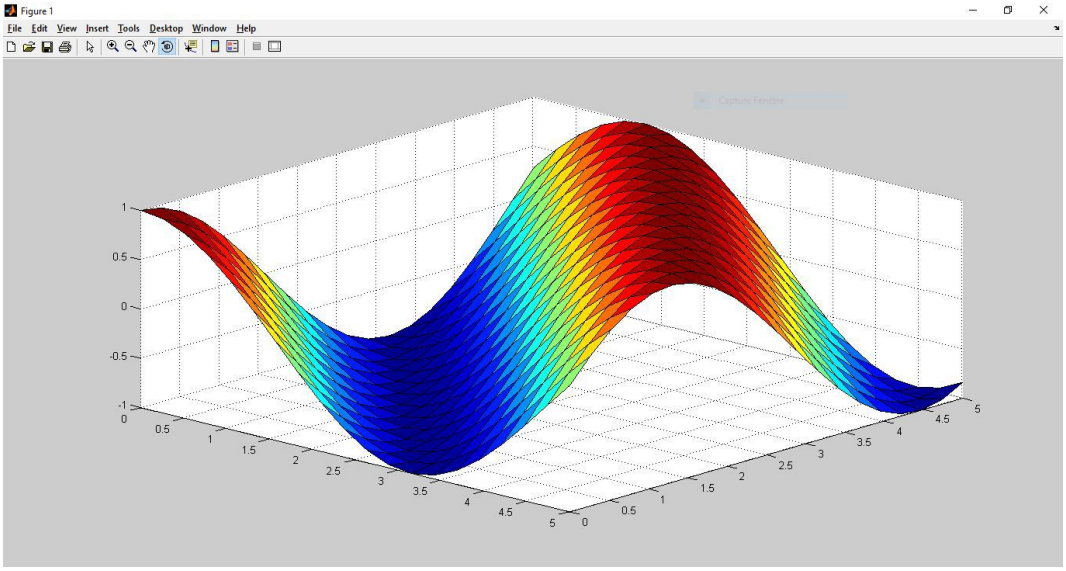

Figure 8 : Dessiner une surface

2. Exercices

2.1. Exercice : Courbe de la fonction sinus (TP)

Question

[solution n°22 p.78*]*

- Générer un vecteur `x` contenant 100 valeurs uniformément distribuées sur l'intervalle [0, π].
- Calculer le sinus de chaque élément du vecteur `x`.
- Afficher la courbe de la fonction sinus.

2.2. Exercice : Définir une fonction et afficher sa courbe (TP)

Question

[solution n°23 p.78*]*

- Définir la fonction `f(x) = sin(x) * cos(3x)`.
- Afficher C_f la courbe de la fonction `f(x)` sur l'intervalle [0, π].

2.3. Exercice : Deux courbes sur la même figure (TP)

Question

[solution n°24 p.78*]*

- Utiliser la commande `help` de Matlab pour avoir une documentation détaillée de la fonction `plot`.
- Tracer les deux courbes des deux fonctions `sin` et `cos` sur la même figure en utilisant une couleur et un style de tracé différents pour chaque courbe (tracer les courbes sur l'intervalle *[0, 2π]*).

Références

VI

[1] Moler, Cleve. MATLAB users' guide. University of New Mexico, 1982.

[2] Fausett, Laurene V., et al. Applied numerical analysis using MATLAB. Upper Saddle River, NJ:: Prentice hall, 1999.

[3] Moler, Cleve B. Numerical computing with MATLAB. Society for Industrial and Applied Mathematics, 2004.

[4] Mathews, John H., and Kurtis D. Fink. Numerical methods using MATLAB. Vol. 4. Upper Saddle River, NJ: Pearson prentice hall, 2004.

[5] Moore, Holly, and Somitra Sanadhya. MATLAB for Engineers. Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[6] Gilat, Amos. Numerical Methods with MATLAB. Wiley Publishing, 2010.

[7] Higham, Desmond J., and Nicholas J. Higham. MATLAB guide. Society for Industrial and Applied Mathematics, 2016.

[8] Kwon, Young W., and Hyochoong Bang. The finite element method using MATLAB. CRC press, 2018.

[9] Yang, Won Y., et al. Applied numerical methods using MATLAB. John Wiley & Sons, 2020.

Solutions des exercices

> **Solution** n°1

Exercice p. 38

```
>> v = [1, 2, 3, 4, 5]; % Solution 1
>> v = 1:5;             % Solution 2
```

> **Solution** n°2

Exercice p. 38

```
>> x = rand(1, 5); % Générer le vecteur x
>> y = rand(5, 1); % Générer le vecteur y
>> z = x * y;      % Calculer le produit scalaire x * y
```

> **Solution** n°3

Exercice p. 38

```
>> A = [2 -2 4; 1 1 0; 4 1 -5];
>> b = [6; 7; 8];
>> x = A^-1 * b
```

> **Solution** n°4

Exercice p. 38

```
>> M = magic(5);
>> sum(M);
>> sum(M, 2);
>> sum(diag(M));
>> sum(diag(rot90(M)));
```

> **Solution** n°5

Exercice p. 38

```
>> x = [pi / 6, pi / 4, pi / 3]; % Générer le vecteur x
>> y1 = sin(x);                  % Calculer sinus(x)
>> y2 = cos(x);                  % Calculer cosinus(x)
>> z = y1 ./ y2;                 % Calculer tan(x) = sin(x) / cos(x)
```

> **Solution** n°6

Exercice p. 39

```
>> x = 1:50;          % Générer le vecteur x
>> y = x(1:5);        % Extraire les 5 premières valeurs de x
>> z = x(end-4:end); % Extraire les 5 dernières valeurs de x
>> n = x(2:2:50);     % Extraire les éléments à indice pair de x
```

> **Solution** n°7

Exercice p. 39

```
>> A = [1 2 3; 4 5 6];
>> B = [A zeros(2, 3); zeros(2, 3) A];
>> B = B(1:end-1, 3:end)
```

> **Solution** n°8

Exercice p. 39

```
>> M = rand(8);
>> L = tril(M);
>> d = prod(diag(L))
```

> **Solution** n°9

Exercice p. 53

```
a = input('donner la valeur de a : ');
b = input('donner la valeur de b : ');
c = input('donner la valeur de c : ');
d = a;
a = b;
b = c;
c = d;
display(a);
display(b);
display(c);
```

> **Solution** n°10

Exercice p. 53

```
note = input('Donner la moyenne : ');
if note >= 10
   disp('admis');
else
   disp('ajourné');
end;
```

> **Solution** n°11

Exercice p. 53

Boucle for

```
depart = input('Donnez un nombre de départ : ');
for k in depart+1:depart+10
   disp(k);
end;
```

Boucle while

```
depart = input('Donnez un nombre de départ : ');
k = depart + 1;
while k <= depart + 10
   disp(k);
```

```
    k = k + 1;
end;
```

> **Solution** n°12

Exercice p. 53

```
function Y = insert(A, x)
n = length(A);
Y = x * ones(1, 2 * n);
Y(1:2:end) = A;
```

> **Solution** n°13

Exercice p. 53

```
function res = factorielle(n);
res = 1;
for i in 2:n
   res = res * i;
end;
```

> **Solution** n°14

Exercice p. 53

```
n = input('saississez un nombre entier positif S.V.P : ');
disp(factorielle(n));
```

> **Solution** n°15

Exercice p. 54

Définir la fonction

```
function c = pgcd(a, b)
while a * b ~= 0
   if a > b
      a = a - b;
   else
      b = b - a;
   end;
end;
c = a + b;
```

Définir le script

```
a = input('donner la valeur de a : ');
b = input('donner la valeur de b : ');
disp(pgcd(a, b));
```

> **Solution** n°16

Exercice p. 54

Solution 1

```
function res = trouver(v, x);
res = 0;
for y in v
```

```
    if y == x
       res = 1;
    end;
end;
```

Solution 2

```
function res = trouver(v, x);
res = (length(v(v == x)) > 0);
```

> **Solution** n°17

Exercice p. 66

```
% Script - evaluer.m
function e = evaluer(p, x)
e = 0;
for a = p
    e = e * x + a; % Nous utilisons ici la méthode de Horner
end
```

> **Solution** n°18

Exercice p. 66

```
% Script - derive.m
function q = derive(p)
for index = 1:length(p) - 1
    q(index) = p(index) * (length(p) - index);
end
```

> **Solution** n°19

Exercice p. 66

```
% Fonction - gaussSeidel.m
function X = gaussSeidel(A, b, X0)
D = diag(diag(A));          % D prend les éléments diagonaux de A
invD = D^(-1);              % nous calculons D^-1
R = A - D;                  % R contient les éléments non diagonaux de A
X = X0;
while norm(b - A * X) > eps % condition d'arrêt
    for index = 1:length(X)
        X(index) = invD(index, index) * (b(index) - R(index,:) * X);
    end;
end;
```

> **Solution** n°20

Exercice p. 66

```
% Fonction - trapeze.m
function res = trapeze(strFunc, a, b)
integFunc = str2func(strFunc);
Xk = linspace(a, b, 101);           % Générer tous les Xk distribués
uniformement sur l'intervalle [a, b]
fx = integFunc(Xk);                 % Calculer les images des Xk
dx = (b - a) / 100;                 % Calculer Δx
res = (sum(fx(2:end)) + sum(fx(1:end-1))) * dx / 2;
```

> **Solution** n°21

Exercice p. 66

```
% Fonction - dichotomie.m
function x = dichotomie(strFunc, a, b)
f = str2func(strFunc);
c = (a + b) / 2;
if f(c) == 0 | c == a | c == b
    x = c;
elif f(c) * f(a) < 0
    dichotomie(strFunc, a, c);
else
    dichotomie(strFunc, c, b);
end;
```

> **Solution** n°22

Exercice p. 72

```
% Script - courbeSinus.m
x = linspace(0, pi, 100);
y = sin(x);
plot(x, y);
```

> **Solution** n°23

Exercice p. 72

Définir la fonction

```
function y = f(x)
y = sin(x) .* cos(3*x);
```

Afficher la courbe

```
>> x = 0:0.1:pi;
>> plot(x, f(x));
```

> **Solution** n°24

Exercice p. 72

La commande help

```
>> help plot
 PLOT   Linear plot.
    PLOT(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, disconnected
    line objects are created and plotted as discrete points vertically at
    X.

    PLOT(Y) plots the columns of Y versus their index.
    If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
    In all other uses of PLOT, the imaginary part is ignored.

    Various line types, plot symbols and colors may be obtained with
```

```
    PLOT(X,Y,S) where S is a character string made from one element
    from any or all the following 3 columns:

           b     blue          .     point              -     solid
           g     green         o     circle             :     dotted
           r     red           x     x-mark             -.    dashdot
           c     cyan          +     plus               --    dashed
           m     magenta       *     star             (none)  no line
           y     yellow        s     square
           k     black         d     diamond
           w     white         v     triangle (down)
                               ^     triangle (up)
                               <     triangle (left)
                               >     triangle (right)
                               p     pentagram
                               h     hexagram

    For example, PLOT(X,Y,'c+:') plots a cyan dotted line with a plus
    at each data point; PLOT(X,Y,'bd') plots blue diamond at each data
    point but does not draw any line.
...
...
...
...
```

Le script

```
% Script - courbeSinCos.m
x = linspace(0, 2 * pi, 100);
y = sin(x);
z = cos(x);
plot(x, y, 'r--');
hold on;
plot(x, z, 'b-.');
```