MEC6212 : Génération de maillages

RAPPEL : *LA PROGRAMMATION AVEC MATLAB*

RICARDO CAMARERO
Département de génie mécanique
École Polytechnique de Montréal

Janvier 2016

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

Table des matières

Chapitre 1

Généralités

1.1 Présentation de Matlab

MATLAB© est un ensemble d'outils informatiques pour le développement de programmes qui intègrent le calcul numérique, la visualisation et les interfaces graphiques. Le système comprend :

- Un environnement de développement ;
- Une programmathèque de fonctions mathématiques ;
- Un langage de programmation ;
- Un langage graphique ;
- Un protocole d'interfaces externes.

L'environnement se présente sous la forme d'une interface graphique, montré à la Fig.1.1, avec plusieurs fenêtres pour :

- la saisie et l'exécution des commandes ;
- la présentation de l'historique de la session de travail ;
- un accès aux variables en mémoire ;
- la navigation et la gestion des fichiers et répertoires de l'usager.

En plus, un éditeur adapté au langage Matlab facilite la rédaction de programmes et de fonctions, le tout complété par un système d'assistance à la programmation (détection d'erreurs, tracé du déroulement de l'exécution d'un programme....)

Sur le plan de la programmation, le langage Matlab est basé sur une structure de donnée où l'élément de base est un tableau avec un jeu opérations adapté à cette entité. La syntaxe pour le traitement de variables qui se représentent sous la forme de vecteurs ou matrices se trouve simplifiée. En plus, une programmathèque permet des opérations variées et riches en fonctionnalités.

L'intégration du langage avec l'interpréteur, l'éditeur et la documentation en fait un outil bien adapté au développement d'applications scientifiques.

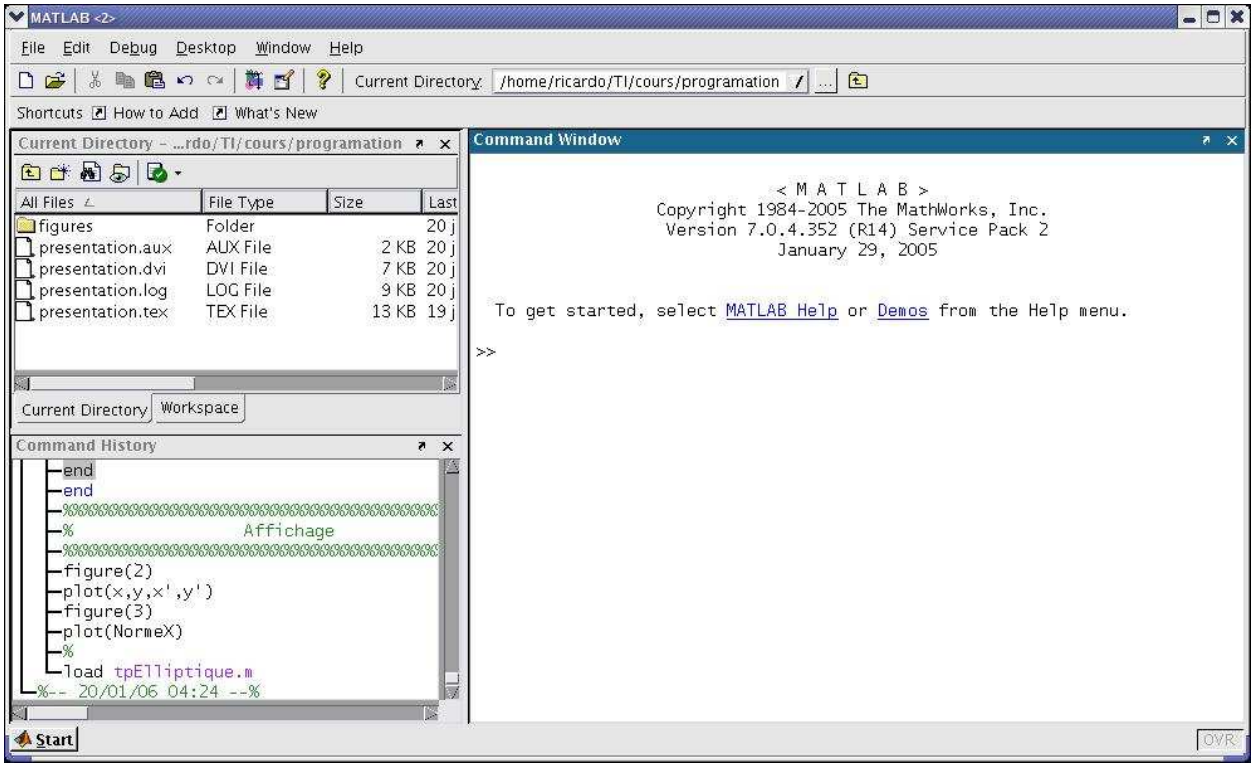


FIGURE 1.1 – Environnement de développement Matlab

Quelques commandes permettent la gestion d'une session de travail :

demo : démonstration de Matlab avec des exemples couvrant un large éventail d'applications ;

cd : change de répertoire *(change directory)* ;

pwd : donne le répertoire courant *(print working directory)* ;

ls ou dir : donne la liste de fichiers dans le répertoire courant ;

delete : supprime un fichier ;

... pour écrire une instruction sur la ligne suivante ;

↑ : recherche à la commande précédente dans la mémoire tampon du système. Cette fonctionnalité permet de reprendre une commande sans avoir à la retaper ;

↓ : même fonction dans le sens inverse ;

On notera la similitude avec les commandes du système UNIX.

1.2 Aide et documentation

Dans une session Matlab, il est possible d'obtenir une aide en ligne sur une commande en tapant :

help nomCommande

On peut aussi utiliser la commande **doc** qui donne accès à la documentation en ligne par l'intermédiaire d'une fenêtre MOSAIC.

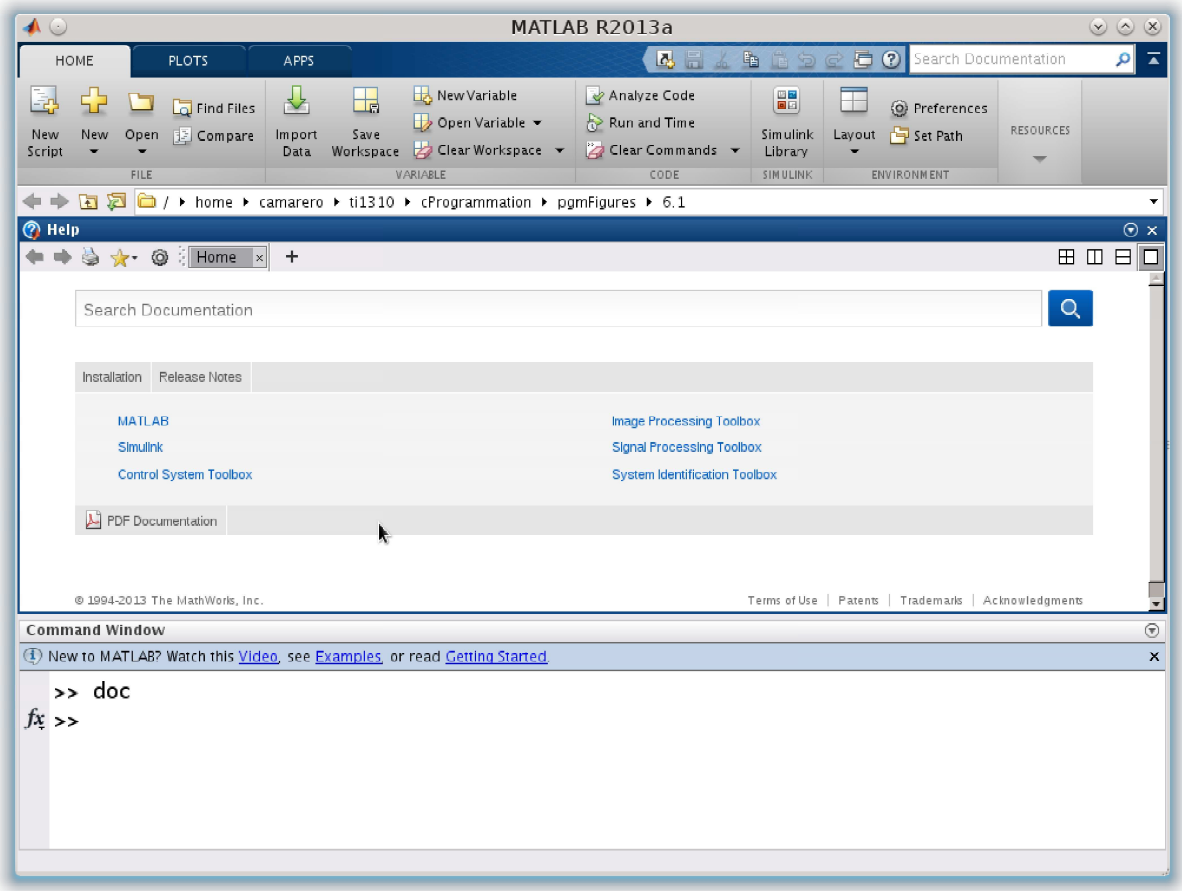


FIGURE 1.2 – Fonction d'aide sous Matlab

Chapitre 2

Types de données

Un ordinateur permet de travailler avec des données : les créer, les manipuler, les traiter et les archiver. Au niveau le plus bas, ces données sont des quantités binaires et incompréhensibles par un humain. Grâce à des programmes, ces données primitives sont enrichies par un langage de programmation qui leur donne un sens dans le contexte d'une application. Ainsi, le type de données que permet un langage dépend du domaine ; scientifique, financier, organisationnel.....Dans le domaine scientifique, on traite de quantités physiques qui se représentent principalement par des nombres, auxquels s'ajoutent les quantités logiques et parfois les quantités de type caractère.

Les opérations de manipulations sont celles de l'arithmétique et de l'algèbre ; addition, soustraction, multiplication et division. Pour des calculs plus sophistiqués, on ajoute un jeu de fonctions, tels les fonctions trigonométriques, statistiques et du calcul analytique.

2.1 Données numériques

Les données numériques dans les langages de programmation scientifique correspondent aux quantités numériques de l'arithmétique classique :

— les nombres entiers ;

— les nombres réels ;

— les nombres complexes ;

Matlab permet la représentation de ces types, positifs et négatifs, avec divers niveaux de précision.

2.1.1 Les entiers

Lorsque l'on travaille avec des entiers de petite et moyenne tailles, il est avantageux de les déclarer comme tels avec les fonctions *intxx* pour des valeurs négatives et positives,

ou *uintxx* pour des valeurs positives. La partie *xx* indique le nombre de bits utilisés. Par exemple, on déclare un entier avec 16 bits à l'aide de la commande,

$$x = int16(32501)$$

Comme un bit est utilisé pour le signe, si on manipule des entiers positifs (> 0), alors la déclaration [1],

$$x = uint16(312)$$

permet une plus grande plage pour ces nombres. Selon la plage des nombres entiers à représenter, on utilisera la déclaration appropriée, parmi celles montrées au Tableau 2.1, ce qui permet une économie de mémoire et de traitement.

	-ve < 0 < +ve				> 0			
8-bit	int8	-2^7	à	2^7-1	uint8	0	à	2^8-1
16-bit	int16	-2^{15}	à	$2^{15}-1$	uint16	0	à	$2^{16}-1$
32-bit	int32	-2^{31}	à	$2^{31}-1$	uint32	0	à	$2^{32}-1$
64-bit	int64	-2^{63}	à	$2^{63}-1$	uint64	0	à	$2^{64}-1$

TABLE 2.1 – Représentation de nombres de type *entier*

2.1.2 Les réels

Théoriquement, un réel nécessite une précision infinie pour le représenter exactement. Avec un nombre fini de bits, soit 16, 32, 64 ou même 128 on n'obtient qu'une approximation. Un nombre est représenté dans la forme,

$$\mathbf{nombre} = \pm\mathbf{mantisse} \times \mathbf{10}^{\mathbf{exposant}}$$

Par exemple, le nombre -357.87 sera représenté sous la forme -3.5787×10^2, et le nombre 0.000469 sous la forme $+4.69 \times 10^{-4}$.

En pratique avec 64 bits, on peut représenter des nombres sur les plages suivantes :

Nombres négatifs : $-1.79769e+308 \Rightarrow -2.22507e-308$
Nombres positifs : $+2.22507e-308 \Rightarrow +1.79769e+308$

Cette représentation à 64 bits nécessite 52 bits pour la mantisse, 11 pour l'exposant et 1 bit pour le signe, avec 23, 8 et 1 bit, respectivement pour une représentation de 32 bits.

Les nombres plus grands que $+1.79769e+308$, ou plus petits que $-1.79769e+308$, ne peuvent être représentés avec seulement 64 bits, et on leur attribue la valeur ***Inf*** $= \pm 1.79769e+308$. Similairement, le plus petit nombre que l'on peut représenter est $2.22507e-308$; c'est-à-dire qu'il n'y a pas, strictement parlant, de zéro !

1. *int* pour *integer* et *uint* pour *unsigned interger* en anglais

Par défaut, MATLAB déclare tous les nombres avec 64 bits, et que l'on appelle des *doubles*. Pour toutes fins pratiques, ces derniers couvrent la plupart des besoins en calcul scientifique. Dans de nombreuses applications, des calculs en simple précision, *single*, suffiront. On utilise alors 32 bits, permettant de représenter des nombres sur les plages suivantes :

$$\begin{array}{lcl} \text{Nombres négatifs} & : & -3.40282e+038 \Rightarrow -1.17549e-038 \\ \text{Nombres positifs} & : & +1.17549e-038 \Rightarrow +3.40282e+038 \end{array}$$

Dans ce mode, les nombres plus grands que $3.40282e+038$ en valeur absolue seront représentés par $\pm$***Inf***, c'est-à-dire infinité. Comme la représentation par défaut est le ***double***, on obtient un ***single*** par la déclaration,

x = single(567.89)

Ce qui représente une économie d'espace et de traitement lorsque la double précision n'est pas nécessaire.

2.1.3 Nombres complexes

Les nombres complexes sont composés d'une partie réelle et d'une partie imaginaire, qui est un nombre réel multiplié par $i = \sqrt{-1}$. Dans Matlab, ils sont déclarés de deux façons, soit explicitement,

```
>> z = 4+5i
z =   4.0000 + 5.0000i
>>
```

ou, avec la commande ***complex*** :

```
>> z = complex(4,5)
z =   4.0000 + 5.0000i
>>
```

On peut extraire les composantes réelle et imaginaire d'un nombre complexe par :

```
>> zreel = real(z)
zreel =     4
>> zimag = imag(z)
zimag =     5
>>
```


2.1.4 Représentations particulières

Certains calculs ne sont formellement pas possibles, par exemple la division par zéro. Matlab prévoit dans ces cas un mécanisme alternatif. Le résultat de toute opération qui donne lieu à un nombre plus grand que la représentation permise par 64 bits (double) sera représenté par ***Inf***. Par exemple :

```
>> x = 1/0
Warning: Divide by zero.

x =   Inf
>>
```

```
>> x = log(0)
Warning: Log of zero.

x =  -Inf
>>
```

Certaines opérations n'ont pas de sens sur le plan mathématique. Alors, le résultat sera représenté par l'entité ***NaN***, (Not a Number). Par exemple :

```
>> z =0/0
Warning: Divide by zero.

z =   NaN

>> z =i/0
Warning: Divide by zero.

z =      NaN +    Infi
>>
```

2.2 Données logiques

Une donnée logique représente un état, "vrai" ou "faux", d'une variable ou d'une expression, par les nombres 1 et 0, respectivement.

```
>> etatV = true
etatV =     1
>> etatF = false
etatF =     0
>>
```

Les données logiques peuvent être manipulées et combinées par des opérations logiques :

Les opérateurs relationnels permettent d'évaluer si une expression est "vrai" ou "faux", et lui assignent "1" ou "0", respectivement.

Opérations logiques		Opérateurs relationnels	
&	**et** (and)	==	égal
|	**ou** (or)	~=	pas égal
~	**pas** (not)	<	plus petit
		>	plus grand

TABLE 2.2 – Opérations logiques

```
>> 10 + sqrt(25) < 0
ans =     0
>> sin(45) == 1
ans =     0
>>
```

Les opérations logiques permettent de combiner plusieurs expressions et d'en évaluer l'état :

```
>> sin(pi/2) <= 1 & sqrt(sin(pi)) == 0
ans =     0
>>
```

Ces manipulations sont utiles pour le contrôle de calculs et permettent de les aiguiller selon certaines conditions. Par exemple, le déroulement d'un calcul pourrait être modifié selon la valeur ou le signe d'une quantité, ou bien un procédé industriel corrigé selon la valeur d'un paramètre.

2.3 Données caractères

Certaines applications utilisent des données alpha-numériques : le nom d'une personne, son adresse, etc. L'ordinateur manipule alors des chaînes de caractères et chiffres : l'alphabet mais aussi d'autres caractères.

```
>> nom = 'Ecole Polytechnique'
nom =   Ecole Polytechnique
>>
```

Les opérateurs logiques du Tableau 2.2 peuvent être utilisés pour faire des comparaisons entre deux chaînes de caractères :

```
>> nomA = 'Poly'
nomA =      Poly
>> nomB = 'poly'
nomB =      poly
>> nomA == nomB
ans =
     0     1     1     1
>>
```

L'opération "==" compare le contenu de deux chaînes, caractère par caractère. Le résultat indique que les chaînes sont identiques sauf pour le premier caractère, "P" et "p".

2.4 Données calendrier

2.5 Opérations et fonctions

Les opérations classiques de l'arithmétique sont supportées par le langage Matlab :

Symbole	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
^	Puissance
'	Transposée complexe

TABLE 2.3 – Opérations arithmétiques

Un certain nombre de fonctions trigonométriques et algébriques sont disponibles : **sin, cos, ..., log, exp...**

Les différents types de nombres, de variables, d'opérations et de fonctions peuvent être combinés pour créer des expressions :

$T = sin(45)^2 + cos(45)^2$

```
>> T = sin(45)^2 + cos(45)^2
T =     1
```

$Z = \sqrt{-1}$

```
>> Z = sqrt(-1)
Z =         0 + 1.0000i
```

$a = \frac{(log(10) - 1)}{(5^2 + 1)}$

```
>> a = (log(10) -1)/(5^2 + 1)
a =    0.0501
>>
```

Chapitre 3

Programmation

3.1 Structure d'un programme

Un programme est une suite structurée d'instructions qui permet de manipuler et traiter des données pour obtenir un résultat. Il est composé d'énoncés qui comprennent des déclarations, des expressions, des entrées-sorties appliquées à des données. Les composantes d'un programme sont les variables et les opérateurs à partir desquels on construit des expressions. Avec des expressions, on forme des énoncés ou commandes. Ces commandes sont soumises à un interpréteur (ou un compilateur) qui les décompose en instructions-machine et sont ensuite exécutées par l'ordinateur.

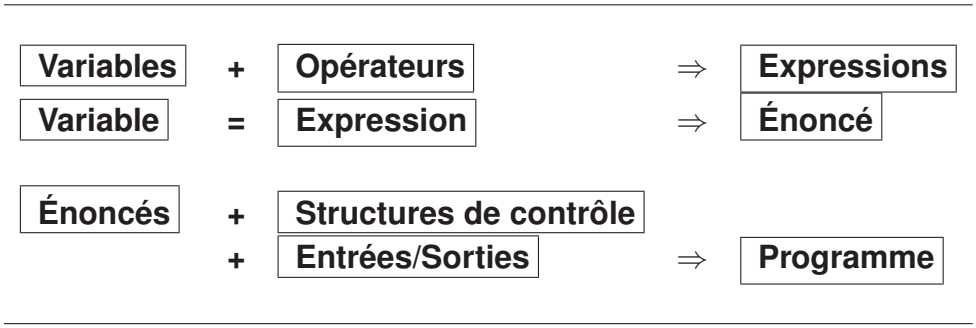


FIGURE 3.1 – Structure d'un programme

Un programme comprend un programme principal et éventuellement des fonctions ou sous-programmes, appelés par le premier.

3.2 Les variables

Les entités de base traitées par un ordinateur sont des données et des instructions machines. Les variables sont des étiquettes ou des noms pour identifier les données qui sont en mémoire. Ces noms permettent de trouver la valeur d'une donnée, de la traiter et

d'en faire une sauvegarde pour utilisation ultérieure. Les variables locales sont connues à l'intérieur du programme (sous-programme) où elles ont été déclarées, tandis que les variables globales sont connues par plusieurs sous-programmes.

Le nom d'une variable doit obligatoirement commencer par une lettre, suivie par une suite de lettres et de nombres, et ne doit pas contenir des espaces ni de caractères spéciaux. La casse est importante et bien qu'il n'y a pas de limite au nombre de caractères, cependant, seuls les premiers ***namelengthmax*** caractères sont reconnus. Il faut que chaque variable d'un programme soit unique et non ambigue.

MATLAB réserve un certain nombre de mots clés (des opérateurs) qui ne peuvent pas être utilisés comme variables dans un programme :

break	end	persistent
case	for	return
catch	function	switch
continue	global	try
else	if	while
elseif	otherwise	

TABLE 3.1 – Noms réservés

Également, certaines variables ont des valeurs prédéfinies dans un contexte scientifique et généralement utilisées dans des calculs. Par exemple, la variable **pi** est équivalente à la constante π :

```
>> x = 2 * pi
x =   6.2832
>>
```

Les variables **intmax** et **realmin** donnent les plus grands entier et réel, respectivement, pouvant être représentés :

```
>> intmax
ans =  2147483647
>> realmin
ans =  2.2251e-308
>>
```

Ces valeurs dépendent du matériel ansi que du système d'exploitation.

Le Tableau 3.2 donne une liste complète de ces variables.

3.3 Les expressions et énoncés

Une expression est une combinaison de variables et d'opérations (arithmétiques etou logiques qui sont la "traduction" informatique des expressions mathématiques que l'on

ans	Valeur du dernier calcul fait par MATLAB
eps	Précision des calculs en virgule flottante
intmax	Le plus grand nombre entier en 8-, 16-, 32-, ou 64-bit que l'ordinateur peut représenter
intmin	Le plus petit nombre entier en 8-, 16-, 32-, ou 64-bit que l'ordinateur peut représenter
realmax	Le plus grand nombre réel que l'ordinateur peut représenter
realmin	Le plus petit nombre réel que l'ordinateur peut représenter
pi	3.1415926535897...
i, j	$\sqrt{-1}$
inf	«Infinité» c'est-à-dire le plus grand nombre pouvant être représenté.
NaN	*Not a Number*, valeur numérique non valide

TABLE 3.2 – Variables réservées

veut calculer. Une expression sera évaluée par Matlab qui effectue les opérations indiquées et le résultat sera affecté à une variable par l'opérateur **"="**. Cet opérateur n'a pas le même sens que le signe **"="** dans le domaine mathématique. L'opérateur informatique signifie que le résultat de l'expression à droite de **"="** sera "déposé" dans un espace mémoire correspondant à la variable à gauche de l'énoncé (voir Fig. 3.1).

Dans les langages de programmation scientifique, il existe plusieurs types d'énoncés :

Déclaration : Avec ce type d'énoncé, on déclare ou on définit une variable en spécifiant son type et sa valeur comme illustré à la Fig. 3.2.

```
%-------------------------------------------------------------
%  Ceci est un commentaire et ne sera pas execute
%-------------------------------------------------------------
global ALPHA BETA% Variables partagees avec d'autres programmes
ALPHA = 0.01;
BETA  = 0.02;
a = uint16(123);%--------------------- Entier positif a 16 bits
b = 120;%-------------------- Reel a double precision (64 bits)
c = single(10);%------------  Reel a simple precision (32 bits)
d = 'abcd';%------------------------------- Variable caractere
```

FIGURE 3.2 – Énoncés de déclaration

Dans ce script, on déclare les variables `ALPHA` et `BETA` comme étant de type **`global`**, c'est-à-dire partagées par plusieurs programmes ou fonctions. Chaque programme ou fonction dispose d'un espace mémoire en propre. Les variables dé-

clarées **global** sont stockées dans un espace mémoire distinct et deviennent accessibles à partir de tous les programmes où elles sont déclarées ainsi.

La variable **a** est déclarée comme un entier, positif à 16 bits. Par défaut, la variable **b** est déclarée comme un réel à 64 bits, tandis que la variable **c** est un réel à simple précision (32 bits). La valeur d'une variable de type caractère, comme **d** est indiquée entre **'...'**. À moins d'autres indications, les variables sont locales et représentées en double précision.

Expression : Ensemble qui combine des variables à l'aide d'opérateurs ou de fonctions pour produire un nouveau résultat qui est affecté à une variable. Le script à la Fig. 3.3 ci-dessous comprend quatre énoncés pour déclarer le type et affecter les valeurs des variables **R**, **n**, **T** et **V**, suivis d'une expression, **n*R*T/V**, pour le calcul de la variable **P**.

```
%-------------------------------------------------------------
%   Calcul de la pression d'un gaz parfait
%-------------------------------------------------------------
R = 1.98726;  %------------------constante universelle des gaz
n = 10.0;     %-------------------------------- nombre de moles
T = 298.7;    %----------------------------------- temperature
V = 24.8;     %------------------------------------- volume
P = n*R*T/V   %------------------------------------ pression
%-------------------------------------------------------------
 P =
    239.3526
```

FIGURE 3.3 – Affectation d'une expression

3.4 Ecriture et formats

On note que les énoncés dans les scripts aux Figs. 3.2 et 3.3 se terminent par **";"** : ceci est la norme sous Matlab. Cependant, si on omet ce point virgule, la valeur de l'expression est imprimée [1]. Le dernier énoncé du script de la Fig. 3.3,

```
P = n*R*T/V
```

ne se termine pas par **";"**, alors, le contenu de la variable ***P*** sera imprimé.

Le script à la Fig. 3.4 montre l'utilisation du point-virgule **";"** à la fin d'une ligne pour l'impression ou non du contenu de la variable. Ainsi l'énoncé ***V_x, V_y*** sans **";"** permet d'afficher le contenu de ces variables.

1. Voir Section 4.2 pour plus de détails.

```
%-------------------------------------------------------------
%    Decomposition d'un vecteur a un angle theta
%-------------------------------------------------------------
L = 1.4147;           %-------------------- Longueur du vecteur
theta = pi/6;         %------------------ angle avec l'axe des x
V_x = L*cos(theta);   %------------------------ composante en x
V_y = L*sin(theta);   %------------------------ composante en y
V_x, V_y              %------------ Sans point virgule: ecriture
%-------------------------------------------------------------
V_x =
   1.2252

V_y =
  0.7073
```

FIGURE 3.4 – Impression des variables

Souvent, il est utile d'afficher le résultat de certains énoncés pour faciliter la recherche d'erreurs de programmation. Alors, l'astuce qui consiste à omettre le point virgule devient intéressante.

Par défaut, l'écriture se fait par le format **`format short`** avec quatre décimales. L'énoncé **`format long`** permet d'augmenter le nombre a 15. (Pour plus de détail voir Section 4.2)

3.5 Instructions de contrôle

Ces structures coordonnent et dirigent le déroulement de la séquence des énoncés à exécuter à l'aide de tests sur la valeur de certaines variables ou expressions.

3.5.1 L'instruction conditionnée `if/elseif/else`

Cette structure évalue une expression logique et exécute une suite d'énoncés si la valeur de l'expression est "vrai". La syntaxe de la variante la plus simple :

```
if (expression logique)
   instructions
   .........
   instructions
end
```

comprend l'énoncé **`if`** fermé par **`end`**, et tous les énoncés compris entre ces derniers, seront exécutés seulement si la condition **expression logique** vaut "vrai".

La variante ***`else`*** permet de traiter le cas où la condition n'est pas vérifiée (elle vaut "faux"). Tous les énoncés **instructions 1** seront exécutés si la condition **expression logique** vaut "vrai", sinon, les énoncés **instructions 2** seront exécutés. Finalement, la variante **`elseif`** permet de traiter une deuxième condition.

```
if (expression logique)
       instructions1
       .........
       instructions1
   else
       instructions2
       .........
       instructions2
   end
```

```
if (expression logique1)
       instructions
   elseif (expression logique2)
       instructions
   else
       instructions
   end
```

L'exemple suivant, à la Fig. 3.5, est une extension du programme présenté à la Fig. 3.3 et montre l'utilisation de l'énoncé **`if`** pour aiguiller le calcul selon la valeur d'une variable, **`typeGaz`**.

```
%-------------------------------------------------------------
%   Calcul de la pression d'un gaz
%-------------------------------------------------------------
typeGaz = 'PARFAIT';%------------------------------ Gaz parfait
R = 1.98726;n = 10.0; T = 298.7; V = 24.8;%--------- parametres
if typeGaz == 'PARFAIT'
    p = n*R*T/V    %-------------- pression pour un gaz parfait
else
    p = F(n,T,V,R) % fonction pour la pression pour un gaz reel
end
%-------------------------------------------------------------
p =
  239.3526
```

FIGURE 3.5 – L'instruction conditionnelle **`if-else`**

L'expression logique **`typeGaz == 1`** est évaluée, et si le résultat vaut "vrai", alors le déroulement du programme est dirigé vers l'énoncé **`p = n*R*T/V`** et la valeur de la variable **`p`** est imprimée. Sinon, l'exécution est dirigée vers la branche **`else`** où **`p = F(n,T,V,R)`** fait appel à la fonction[2] **`F`**. qui fera le calcul nécessaire à partir des argu-

2. L'utilisation des fonctions est décrit à la Section 3.6

ments **n**, **R**, **T** et **V**.

3.5.2 L'instruction inconditionnelle `for`

Cette structure exécute un nombre prédéterminé de fois, une suite d'énoncés compris entre **for** et **end** :

```
for compteur = debut:pas:fin
    instruction;
    ..........
    instruction;
 end
```

L'énoncé **for** comprend un compteur avec une valeur **debut**, un **pas** et une valeur **fin**. Initialisée à la valeur **debut**, cette structure exécute tous les énoncés compris entre **for** et **end**, et incrémente la valeur du compteur par la valeur du **pas**. Ceci est répété jusqu'à ce que le compteur atteigne la valeur de **fin**.

Dans l'exemple suivant, on exécute les énoncés t et $x = t^2$ à partir de **t = 1**, en augmentant le compteur **t** de **3** à chaque passage dans la boucle, jusqu'à **t = 11**.

```
for t = 1:3:11
      t
      x = t^2
end
```

```
t =
     1
x =
     1
t =
     4
x =
    16
t =
     7
x =
    49
t =
    10
x =
   100
```

Le compteur est une variable de type *entier* ou *réel*. Le pas (intervalle) vaut 1 par défaut, mais peut prendre n'importe quelle valeur, même négative. Dans ce dernier cas, il faut gérer correctement la valeur en fin de boucle.

On peut imbriquer plusieurs boucles **for**, comme illustré par le programme de la Fig. 3.6 qui calcule le factoriel, $n!$, des nombres entiers $n = 1, 2, ...5$. Ce script comprend une boucle externe pour calculer les valeurs du factoriel de **n = 1, 2, ...** jusqu'à **n = 5**. A l'intérieur de cette boucle, on initialise la valeur de factoriel à 1 avec l'énoncé **F = uint16(1);** qui déclare **F** comme un entier à 16 bits. Avec la deuxième boucle, on multiplie successivement par 2, 3, ... jusqu'à n.

Pour illustrer une autre utilisation de la boucle **for**, on montre le calcul de la fonction exponentielle e^x à l'aide de son développement de Taylor dans le voisinage de $x = 0$,

$$\begin{aligned} e^x &= 1 + x/1! + x^2/2! + x^3/3! + ... \\ &= \sum_{m=1}^{mMax} \frac{x^{m-1}}{(m-1)!} \end{aligned}$$

```
%----------------------------------------
% Calcul de factoriel d'un nombre entier
%      n! = 1*2*3.... *n
%----------------------------------------
for n = 1:5
    F = uint16(1);
    for j = 1:n
        F=F*j;
    end
    n,F
end
```

```
  n =     1
  F =     1
  n =     2
  F =     2
  n =     3
  F =     6
  n =     4
  F =    24
  n =     5
  F =   120
>>
```

FIGURE 3.6 – Calcul de $n!$

Dans le script à la Fig. 3.7, on évalue une approximation de $e^{1.0}$ en utilisant les sept premiers termes de ce développement, et on compare avec la valeur "exacte" qui est donnée par la librairie de Matlab en appelant la fonction **`exp(x)`**.

```
%----------------------------------------
% Evaluation de la fonction y = e^x
% utilisant un developpement de Taylor,
% et en specifiant le nombre de termes
%----------------------------------------
clear all;clc;
nbTermes = 7;
x        = 1.;
yExact   = exp(x);
y        = 0.;
for m=1:1:nbTermes% Sommation des termes
    y = y + x^(m - 1)/factorial(m - 1)
    erreur = abs(y - yExact)
end
```

```
y =  1
erreur =  1.7183
y =  2
erreur =  0.71828
y =  2.5000
erreur =  0.21828
y =  2.6667
erreur =  0.051615
y =  2.7083
erreur =  0.0099485
y =  2.7167
erreur =  0.0016
y =  2.7181
erreur = 2.2627e-04
```

FIGURE 3.7 – Évaluation de e^x avec une boucle **for**

À partir d'une valeur égale à 0, on accumule successivement dans la variable y, chaque terme du développement $\frac{x^{m-1}}{(m-1)!}$. Le résultat est une suite de valeurs y qui s'approchent de la valeur e^x avec une erreur décroissante.

3.5.3 Boucle `while`

Cette structure exécute un nombre indéterminé de fois, une suite d'énoncés compris entre **`while`** et **`end`**. La boucle est répétée tant qu'une condition est vérifiée. Cette condition est exprimée sous la forme d'une expression logique, et les instructions dans le bloc entre **`while`** et **`end`** seront exécutées tant que la valeur de **expression** sera "vrai".

Dans l'exemple suivant, la variable **`compteur`** est incrémentée tant quelle est inférieure à 5.

```
compteur =0;
while compteur < 5
    compteur=compteur+1
end
```

```
compteur =
     1
compteur =
     2
compteur =
     3
compteur =
     4
compteur =
     5
```

On montre une utilisation de l'énoncé **`while`**, avec l'algorithme de calcul de la fonction e^x illustré à la section précédente. Dans le script de la Fig. 3.7, on calcule la valeur de e^x en spécifiant le nombre de termes, **`nbTermes=7`**, utilisés dans le développement. Une approche plus intéressante consiste à spécifier la précision souhaitée plutôt que le nombre de termes. Alors, le calcul se fait en ajoutant successivement autant de termes au développement que nécessaire pour atteindre l'erreur voulue. Dans le script à la Fig. 3.8, on remplace la structure inconditionnelle **`for`** par la structure conditionnée **`while`**. L'erreur est évaluée comme l'écart entre la valeur calculée et la valeur exacte en supposant la valeur exacte connue, calculée par la fonction Matlab **`exp(x)`**.

Évidemment, ces deux approches sont donnent des résultats identiques.

3.5.4 Sélection `switch/case/otherwise`

Cette structure sélectionne un bloc d'énoncés entre **`switch`** et **`end`** parmi un ensemble prédéterminé de choix. Chacun des choix représente un **`case`** correspondant à une valeur de **`variable`**. Si la valeur de **`variable`** ne correspond à aucun des choix **`variable1`**, **`variable2`**, ... alors le bloc ou **`case otherwise`** sera exécuté.

```
switch variable
   case valeur1
      instructions
   case valeur2
      instructions
   otherwise
      instructions
end
```

```
%---------------------------------------
% Evaluation de la fonction y = e^x
% utilisant un developpement de Taylor,
% et specifiant une erreur
%---------------------------------------
clear all;clc;
x         = 1.;
yExact    = exp(x);
y         = 0.;
erreur= 100;% valeur initiale tres grande
m=0;
while erreur > .001  % Test sur l'erreur
    m=m+1;
    y = y + x^(m - 1)/factorial(m - 1)
    erreur = abs(y - yExact)
end
```

```
y =  1
erreur =  1.7183
y =  2
erreur =  0.71828
y =  2.5000
erreur =  0.21828
y =  2.6667
erreur =  0.051615
y =  2.7083
erreur =  0.0099485
y =  2.7167
erreur =  0.0016
y =  2.7181
erreur = 2.2627e-04
```

FIGURE 3.8 – Évaluation de e^x avec une boucle **while**

L'exemple ci-dessous (Fig. 3.9) illustre un contexte particulier de l'utilisation de la structure **switch/case/otherwise**. A l'invite **selectionner un gaz:** qui apparaît dans la fenêtre de commande, la fonction **input** lit la chaîne de caractères entrée au clavier [3]. La fonction **switch** compare avec chacun des choix proposés par les différents **case**, et, selon la valeur, affiche le message **Gaz choisi:**.

```
gaz=input('selectionner un gaz: ');
switch gaz
    case 'helium'
        disp('Gaz choisi: helium');
    case 'azote'
        disp('Gaz choisi: azote');
    case 'methane'
        disp('Gaz choisi: methane');
    otherwise
        disp('un autre gaz');
end
```

```
selectionner un gaz:'methane'
Gaz choisi: methane
```

FIGURE 3.9 – La structure **switch-case**

3. Taper entre apostrophes : ' '. Voir le Chapitre 4 pour plus de détails

Cette structure de contrôle s'apparente à la structure **`if-elseif-else`**[4]. Les programmes aux Fig. 3.10 et 3.11 comparent ces deux structures. Il s'agit de déterminer le nombre et le type de racines d'un polynôme de degré trois.

$$P_3 = \sum_{i=0}^{3} a_i x^i$$

A partir des coefficients a_i, on évalue le discriminant, et selon les trois possibilités,

discriminant > 0

discriminant < 0

discriminant = 0

un message distinct sera affiché.

Avec la structure **`if-elseif-else`**, on teste les différentes possibilités avec les expressions **`D > 0`** ou **`D < 0`**.

```
%-----------------------------------------------------------------
% Illustration de la difference entre if/elseif/else et switch
%-----------------------------------------------------------------
% Calcul des racines d'une equation du troisieme degre
%          x^3 + a*x^2 + b*x + c =0
% Le discriminant D = q^3 - r^2  ou   q = b/3 - a^2/9  et
% r = (b*a - 3c)/6 - a^3/27
% determine le nombre et le type de racines.
%-----------------------------------------------------------------
a= 0.0;b=-1.0;  c= 0.0;%------------ Declaration des coefficients
q = b/3 - a^2/9 ;%----------------------- Calcul du discriminant
r = (b*a - 3*c)/6 - a^3/27;
D = q^3 - r^2; signeD= D/ abs(D);
if D > 0%---------------------- Premiere methode: if elseif else
message ='Une racine rÃl'elle, et deux racines complexes';
elseif D < 0
message ='Trois racines rÃl'elles distinctes';
 else
message ='Trois racines rÃl'elles, dont deux identiques';
end
disp(message);
```

FIGURE 3.10 – Comparaison des structures **`if-elseif-else`** et **`switch-case`**

Tandis qu'avec la structure **`switch-case`**, on forme dabord une variable **`signeD= D/abs(D)`** qui vaut =1, -1 ou 0, ce qui donne trois **`case`** correspondants.

4. Voir Section 3.5.1

```
switch signeD%------------------- Deuxieme methode: switch case
    case -1
        message ='Trois racines rÃl'elles distinctes';
    case 1
        message ='Une racine rÃl'elle, et deux racines complexes';
    case 0
        message ='Trois racines rÃl'elles, dont deux identiques';
    otherwise
        disp('aucune valeur');
end
disp(message);
```

FIGURE 3.11 – Comparaison des structures **`if-elseif-else`** et **`switch-case`**

3.5.5 Sélection `break`

3.6 Les scripts et fonctions

Dans l'environnement MATLAB, les énoncés peuvent être entrés directement dans la fenêtre de commandes, ou bien sous la forme d'un programme ou ***script***. Un programme est une suite d'instructions contenues dans un fichier créé par un éditeur et ensuite soumis pour exécution dans la fenêtre de commandes. Il y a deux types de programmes ; les scripts et les fonctions. Ces deux types de fichiers portent l'extension *******.m***.

On appelle un script ***programme principal***, et il ne comporte pas d'arguments en entrée ou sortie. Il sert à construire les données et variables dans l'espace mémoire principal de MATLAB. Il gère le déroulement des calculs et peut faire appel à des fonctions, aussi appelées sous-routines ou sous-programmes.

Parmi les fonctions, on trouve celles prédéfinies dans MATLAB tel que les fonctions trigonométriques, algébriques... ou bien des fonctions définies par le programmeur. On crée une fonction dans un fichier à l'aide d'un éditeur. Le fichier porte le nom de la fonction et prend la forme de *******.m***. Le premier énoncé dans le fichier est l'énoncé **`function`**[5] qui sert à déclarer la fonction.

```
function variableSortie = nomFonction(arg1,arg2,.....)
```

Ces fonctions ont généralement des arguments en entrée et sortie ; ce qui permet la communication de données entre le programme appelant et le programme appelé. Suite à l'énoncé **`function`**, suivent des commentaires, des instructions etc.... tel qu'illustré à la Fig. 3.12.

Lorsqu'appelée par le programme principal, la fonction **`nomFonction`** reçoit les valeurs de **`arg1`**, **`arg2`** .... et retourne la variable **`variableSortie`**. Toutes les variables

5. Attention à l'orthographe qui est anglaise !

```
function variableSortie = nomFonction(arg1,arg2,.....)
% ----------------------------------------+
% La fonction nomFonction calcule ..........
% Programmeur: nom prenom
% Date : 21 septembre 2014
% ----------------------------------------+
enonce;
enonce;
enonce;
variableSortie =........
```

FIGURE 3.12 – Déclaration d'une fonction usager

déclarées dans la fonction se trouvent dans l'espace mémoire de celle-ci et ne sont pas accessibles par le programme appelant. Réciproquement, les variables et données déclarées dans le programme principal ne sont pas connues des fonctions, car elles résident dans une mémoire distincte. Les seules variables partagées sont les arguments communiqués entre le script et la fonction.

Dans certaines circonstances, pour permettre l'accès à des variables entre le programme principal et les sous-programmes, il peut être utile de déclarer certaines variables comme des variables globales (voir la Section 3.2). Une variable déclarée ***global*** est alors partagée entre le programme principal et plusieurs fonctions sans qu'il soit nécessaire de la spécifier parmi les variables d'entrée/sortie ou comme argument des différentes fonctions. On déclare une variable globale avec l'énoncé ***global***, qui doit apparaître dans tous les programmes et sous-programmes qui partagent la variable.

La Fig. 3.13 montre un exemple d'un programme principal (script) qui utilise la fonction (sous-programme) **`T`** pour le calcul de la température à partir des arguments **`P`**, **`V`** et **`n`**.

```
%-------------------------------------------------------------
% Calcul de la temperature d'un gaz utilisant la loi
% des gaz parfait:    PV = nRT
%-------------------------------------------------------------
global R
R = 1.98;          %----- Constante universelle des gaz parfaits
P = 100;           %-----------------------------------pression
V = 10;            %-----------------------------------volume
n = 1;             %----------------------------nombre de moles
temperature = T(P,V,n)%----------------  Appel a la fonction T
```

FIGURE 3.13 – Protocole d'appel d'une fonction à l'intérieur d'un script

Cette fonction contient les énoncés qui réalisent le calcul, et retourne le résultat dans l'argument ou variable de sortie **`T`**. Le code source (Fig. 3.14) se trouve dans un fichier ap-

pelé **T.m**[6]. Dans cet exemple, l'appel à la fonction **T** aurait pu être remplacé par l'énoncé **temperature = P*V/R/n;**, mais dans des situations plus complexes ou lorsque cette fonction est appellée de plusieurs endroits, l'encapsulation est très avantageuse.

On note que la variable R est de type ***global***. Cette variable a été déclarée dans le programme principal, et accessible dans la fonction (sous-programme) **T** et toute autre fonction où apparaît l'énoncé, global R .

```
function temperature = T(P,V,n)
%-------------------------------------------------------------------
% Fonction qui calcule la temperature d'un gaz parfait
% En entree: P, V, n
% En sortie: temperature
% Global   : R
%-------------------------------------------------------------------
global R
temperature = P*V/R/n;
```

FIGURE 3.14 – Code source de la fonction **T**

Dans l'exemple suivant à la Fig. 3.15, le programme principal (script) calcule les racines d'un polynôme du 2ème degré

$$ax^2 + bx + c = 0$$

à l'aide d'une fonction appelée **racinesP2** :

```
%-------------------------------------
% Calcul des racines du polynome
%    ax^2 + bx^1 + c
%-------------------------------------
clc;clear all;close all
%-------------------- racines reelles
a = 1; b = -5; c = 6;
[racine1, racine2]=racinesP2(a,b,c)
%------------------- racines complexes
a = 2; b = -3; c = 16;
[racine1, racine2]=racinesP2(a,b,c)
```

```
racine1 =
     3

racine2 =
     2

racine1 =
  0.7500 + 2.7272i

racine2 =
  0.7500 - 2.7272i
```

FIGURE 3.15 – Programme principal pour le calcul des racines d'un polynôme de degré 2

Le rôle du script est de poser les valeurs des coefficients a, b et c, et de les passer en arguments à la sous-routine (fonction) **racinesP2**. Les calculs sont effectués avec les

6. La fonction et le fichier qui contient le code source doivent porter le même nom

coefficients `a`, `b` et `c` à l'interieur de la fonction comme montré à la Fig. 3.16 qui retourne la valeur des deux racines. À l'aide de la structure ***if else***, on dirige le calcul vers les expressions pour des racines réelles ou complexes, selon la valeur du discriminant.

On note que le programme principal et la fonction appelée sont chacun dans des fichiers distincts.

```
function [r1,r2]=racinesP2(a,b,c)
%-------------------------------------------------------------
% Calcul des racines du polynome:   ax^2 + bx^1 + c
% En entree: a, b, c En sortie: r1,r2
%-------------------------------------------------------------
discr = b*b -4*a*c;
if discr < 0%------------------------------ 2 racines compexes
    D= sqrt(abs(discr));
    r1=complex(-b,D)/2/a;  r2=complex(-b,-D)/2/a;
else%--------------------------------------- 2 racines reelles
    D=sqrt(discr);
    r1=(-b + D)/2/a;       r2= -b - D)/2/a;
end
```

FIGURE 3.16 – Fonction pour le calcul des racines d'un polynôme de degré 2

Chapitre 4

Les entrées/sorties

4.1 Entrées interactives

Les programmes montrés dans les chapitres précédents utilisent des données ou paramètres pour faire les calculs. Les valeurs de ces données ont été spécifiées par des énoncés d'affectation. Par exemple, à la Section 3.6, le programme de la Fig. 3.15 calcule les racines d'un polynôme du 2ème degré $ax^2 + bx + c = 0$ où les coefficients a, b et c sont affectés avec les énoncés suivants,

```
a = 2; b = -3; c = 16;
```

Souvent, pour des programmes génériques, on voudrait changer facilement ces valeurs sans qu'il soit nécessaire de modifier ces énoncés dans le fichier source du programme. Pour ce faire, Matlab propose plusieurs fonctions pour la saisie interactive de telles données.

4.1.1 `input`

La fonction **`input`** saisit directement au clavier une valeur, et ensuite l'affecte à la variable correspondante.

input(message,'s')

où **`message`** est un appel qui apparaît dans la fenêtre de commande, et sollicite une saisie. C'est une chaîne de caractères par laquelle le programmeur informe l'usager de l'action à prendre. La fonction **`input`** redirige l'entrée standard vers la fenêtre de commande de Matlab, comme montré à la Fig. 4.1.

L'argument **`'s'`** indique que la réponse attendue est une chaîne de caractères et sera retournée tel quel par **`input`**. Si cet argument est absent, alors la réponse de l'usager est interprétée comme une expression et sera évaluée. Ce comportement est illustré par les exemples de la Fig. 4.2. On notera en particulier les exemples 1) et 3), où la chaîne **`3+(7+29)/3`** est évaluée ou pas, selon que l'argument **`'s'`** est présent ou pas.

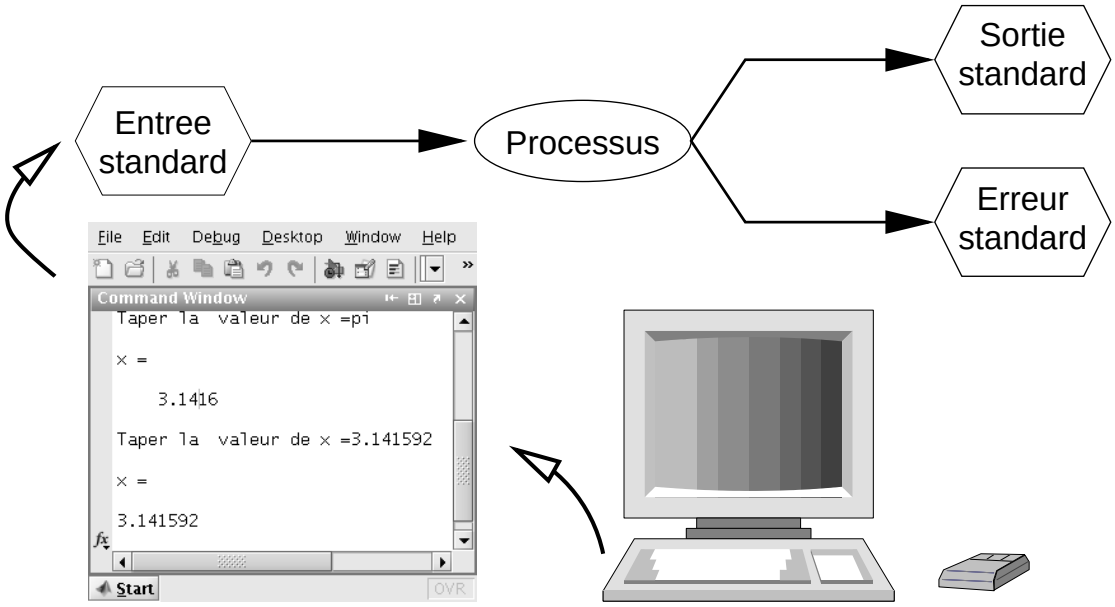


FIGURE 4.1 – Entrée clavier dirigée vers l'entrée standard (fenêtre de commande)

```
%---------------------------------------
%    Arguments de la fonction input
%---------------------------------------
clc;clear all
% 1) Expression
x=input('1) Taper la valeur de x =');
x
% 2) nombre pi
x=input('2) Taper la valeur de x =');
x
% 3) Chaine
x=input('3) Taper la valeur de x =','s');
x
% 4) chaine
x=input('4) Taper la valeur de x =','s');
x
```

```
1)Taper la valeur de
   x =
3+(7+29)/3
x =    15
2)Taper la valeur de
   x =
pi
x =    3.1416
3)Taper la valeur de
   x =
3+(7+29)/3
x =3+(7+29)/3
4)Taper la valeur de
   x =
pi
x =pi
```

FIGURE 4.2 – Utilisation des arguments de la fonction **input**

Un autre exemple est illustré à la Fig. 4.3 où les valeurs de trois variables sont saisies avec la fonction **input**, qui sont ensuite, passées en arguments à la fonction **racines(a,b,c)** (Voir le fichier source à la Fig. 3.16). On note l'affichage d'un titre avec la fonction **disp** qui sera présentée à la Section 4.2.

```
%-------------------------------------
% Calcul des racines du polynome
%    ax^2 + bx^1 + c
% Entree interactive des coefficients
%-------------------------------------
clc;clear all;
%a = 1; b = -5; c = 6;
titre='Calcul␣des␣racines␣de:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ax^2␣+␣bx^1␣+␣c';
disp(titre)
a=input('La␣valeur␣de␣a␣=␣');
b=input('La␣valeur␣de␣b␣=␣');
c=input('La␣valeur␣de␣c␣=␣');
[racine1, racine2]=racines(a,b,c)
```

```
Calcul des racines de:
   ax^2 + bx^1 + c

La valeur de a =
                    2
La valeur de b =
                   -3
La valeur de c =
                   16
racine1 =
  -0.7500 + 2.7272i

racine2 =
  -0.7500 - 2.7272i
```

FIGURE 4.3 – Entrée interactive de plusieurs données avec **`input`**

4.1.2 `inputdlg`

Lorsqu'il y a plusieurs valeurs à saisir, comme dans l'exemple précédent, on utilisera la fonction **`inputdlg`**, plutôt que d'appeler **`input`** plusieurs fois.

inputdlg(message,titre,nbLignes,valDefaut,options))

où les arguements sont définis ci-dessous :

message	un appel qui apparaît dans la fenêtre, et sollicite une saisie pour chaque valeur. C'est une chaîne de caractères par laquelle le programmeur informe l'usager des variables à saisir. Associée à chaque valeur, il y a une case dans laquelle sera tapée la valeur correspondante ;
titre	un titre qui sera affiché sur la fenêtre ;
nbLignes	nombre de lignes pour chaque variable (utile si la variable est un vecteur) ;
valDefaut	valeurs par défaut des variables qui seront affichées dans les cases. Ces valeurs sont choisies par le programmeur pour faciliter la saisie.

Les options permettent de contrôler les propriétés de la fenêtre :

options.Resize ='on'	permet le redimensionnement de la fenêtre ;
options.WindowStyle ='normal	place le programme en attente d'une réponse ;
options.Interpreter ='tex'	Utilise Latex pour formater le message (utile pour des symboles spéciaux pour les variables, comme des formules utilisant l'alphabet grec) ;

La fonction **inputdlg**, contrairement à **input**, redirige l'entrée standard vers une nouvelle fenêtre qui dispose une grille de cases pour les valeurs à saisir. Cette boîte de dialogue est montrée à la Fig. 4.4. On note que les données entrées par l'usager dans chacune des cases sont des chaînes de caractères qui sont converties en **double** par la fonction **str2double** et affectées aux variables a, b et c qui sont ensuite passées en arguments à la fonction **racines(a,b,c)** qui effectue le calcul des racines (Voir le fichier source à la Fig. 3.16).

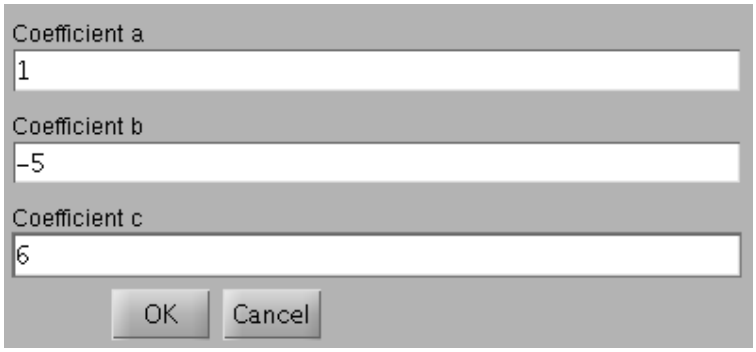


FIGURE 4.4 – Boîte de dialogue pour la saisie interactive de plusieurs valeurs

```
%---------------------- Saisie des coefficients
message={'Coefficient a','Coefficient b',
                          'Coefficient c'};
titre='Calcul des racines de: ax^2 + bx^1 + c';
nbLignes=1;
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';
valDefaut={'0','0','0'};
N=inputdlg(message,titre,nbLignes,valDefaut,
                                      options);

a=str2double(N{1});
b=str2double(N{2});
c=str2double(N{3});
%------------------------- Calcul des racines
[racine1, racine2]=racines(a,b,c)
```

```
racine1 =
          -2


racine2 =
          -3
```

FIGURE 4.5 – Entrée interactive des données avec fenêtre **inputdlg**

4.1.3 `ginput`

Un dispositif de saisie des plus intéressant est le mode graphique par le biais d'un curseur à l'écran. On ouvre une fenêtre graphique qui devient l'entrée standard, et la fonction **`ginput`** présente le curseur que l'usager positionne avec la souris sur le point de l'écran à saisir, comme montré á la Fig. 4.6.

```
[x,y,bouton] = ginput(n)
```

où n indique le nombre de points à saisir, et retourne leurs coordonnées dans les vecteurs colonne x et y.

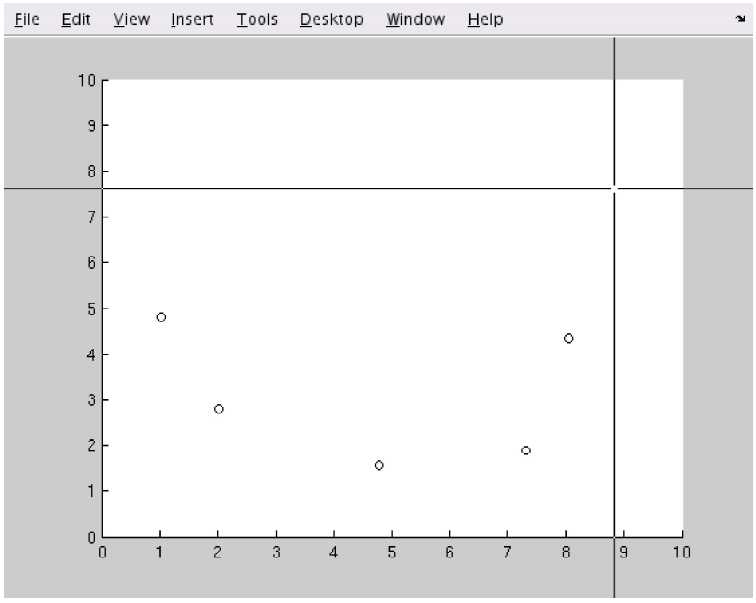


FIGURE 4.6 – Saisie des coordonnées de points á l'écran : **`ginput`**

L'option **`"bouton"`** sert à indiquer comment se termine l'action.

Sans l'option "bouton" : l'action se termine après que n points sont indiqués, mais, avant de faire la saisie, on appuie sur la touche "Entrée" pour indiquer que seulement n points seront saisis.

Avec l'option "bouton" : après chaque clique, la fonction retourne la valeur du bouton de la souris qui a été cliqué, avec la convention suivante :

1. pour le bouton gauche,
2. pour le bouton du centre,
3. pour le bouton de droite.

Remarques :

1. Le programmeur utilisera cette information pour déterminer la course á prendre.

2. Les valeurs des coordonnées saisies sont relatives aux axes respectifs de la fenêtre où l'action a lieu.

Dans l'exemple donné à la Fig. 4.7, on lance la fonction **ginput** qui fait apparaître une fenêtre graphique et un curseur pour la saisie interactive d'une succession de points. L'usager positionne le curseur avec la souris, et les coordonnées du point indiqué par un clic du bouton gauche sont saisies successivement jusqu'à ce que le bouton de droite soit appuyé, terminant ainsi l'action. Pour un bonne rétroaction, aprés chaque saisie, un symbole est affiché pour indiquer à l'usager la position du point.

```
clc; clear all; close all;
figure(1)
axis([0 10 0 10])%------------------- ouvre la fenetre graphique
hold on
but= 1;
while but== 1
    [xi,yi,but]=ginput(1);%--- lance le curseur et saisi le point
    plot(xi,yi,'ko');%------------------------- affiche le point
end
hold off
pause
figure(2)
nbPLN =0;
but= 1;
axis([0 10 0 10])%------------------- ouvre la fenetre graphique
hold on
while but== 1
    nbPLN = nbPLN+1;%--------------------------------PLN suivant
    [xi,yi,but]=ginput(1);%--- lance le curseur et saisi le point
    PLN(nbPLN,1:2) = [xi yi];%--------------- sauvegarde le point
    plot(xi,yi,'ko');%------------------------- affiche le point
end
hold off
```

FIGURE 4.7 – Programme pour la saisie des coordonnées de plusieurs points á l'écran: **ginput**

Ce programme comprend deux parties. Dans la première, les points sont saisis et dessinés au fur et à mesure. Ce qui n'est pas très utile car les coordonnées n'étant pas sauvegardées sont perdues. Dans la deuxième partie, les coordonnées sont affectées dans un tableau et sont alors disponibles pour un traitement subséquent. On note également que seul les points sont affichés, alors le sens du tracé est perdu.

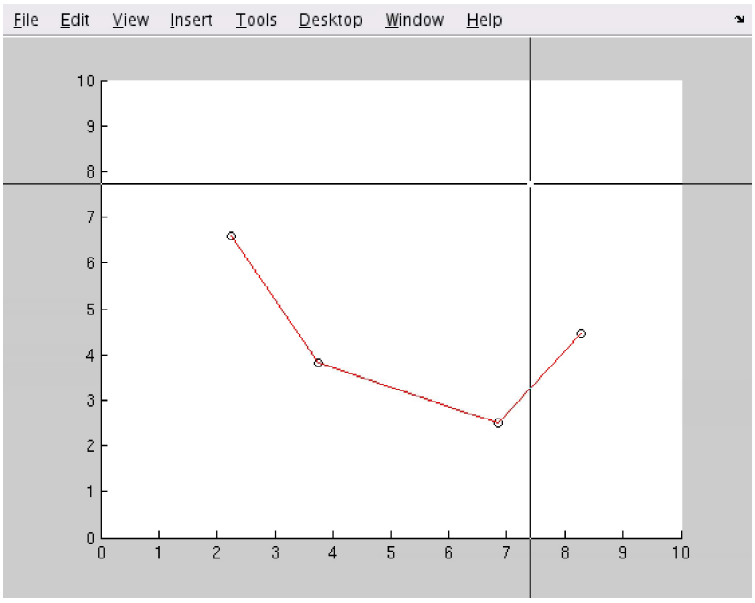


FIGURE 4.8 – **`ginput`** : tracé du polyligne au fur et à mesure de la saisie

Fig. 4.8 montre comment la saisie est accompagnée du tracé de la polyligne au fur et á mesure de la saisie des points. Le programme donné à la Fig. 4.9 montre le déroulement de la séquence des actions. On note la différence du tracé avec le cas précédent de la Fig. 4.6, où le polygone n'est pas dessiné au fur et à mesure ; ce qui donne une moins bonne rétro-action. On note également la différence dans la structure avec le programme de la Fig. 4.7. Il faut être en mesure de distinguer entre l'entrée du premier point (tracer un seul point) et l'entrée des autres points (tracer un segment de droite entre ce point et le point précédent). Comme les deux structures de contrôle (while et if) sont toutes deux nécessaires, est-il préférable de positionner le **`if`** à l'intérieur ou à l'extérieur du **`while`** ? A cette question, il y a pas de réponse générale. Si le **`if`** est à l'intérieur du **`while`**, alors le **`if`** devra être évalué à chaque répétition du **`while`**, ce qui est inefficace ! Cependant, ici le **`while`** se termine par un input usager, alors de le temps d'exécution de chaque répétition du **`while`** dépend majoritairement du temps que l'usager met à cliquer ses points. Dans ce cas, le **`if`** peut être positionné à l'intérieur du **`while`** sans impact sur l'efficacité globale du programme.

4.1.4 `questdlg`

Une dernière fonction d'entrée interactive permet de faire un choix parmi plusieurs options.

```
choix = questdlg(message, titre, choix1, choix2, defaut)
```

La fonction **`questdlg`** affiche une fenêtre avec une message et plusieurs boutons correspondants à différents choix proposés à l'usager.

```
clc; clear all;close all;
figure(1)
axis([0 10 0 10])%------------------- ouvre la fenetre graphique
hold on
nbPLN =0;
[xi,yi,but] = ginput(1);%- lance le curseur et saisi le 1er point
if but==1
    nbPLN=nbPLN+1;%-----------------------------------Premier PLN
    PLN(nbPLN,1:2) = [xi yi];
    plot(xi,yi,'ko');
    but2=1;
    while but2 == 1
        [xi,yi,but2]  = ginput(1);%---- lance le curseur et saisi
        nbPLN = nbPLN+1;%------------------------ le PLN suivant
        PLN(nbPLN,1:2) = [xi yi];
        plot(xi,yi,'ko');%--------------------- affiche PLN et le
        plot(PLN(nbPLN-1:nbPLN,1),PLN(nbPLN-1:nbPLN,2),'-r');%SEG
    end
end
```

FIGURE 4.9 – Saisie des coordonnées de points á l'écran avec tracé de la polyligne

message : texte indiquant á l'usager la liste des choix ;

titre : titre qui sera affiché sur la fenêtre ;

choix1 : libellé du premier choix ;

choix2 : libellé du second choix ;

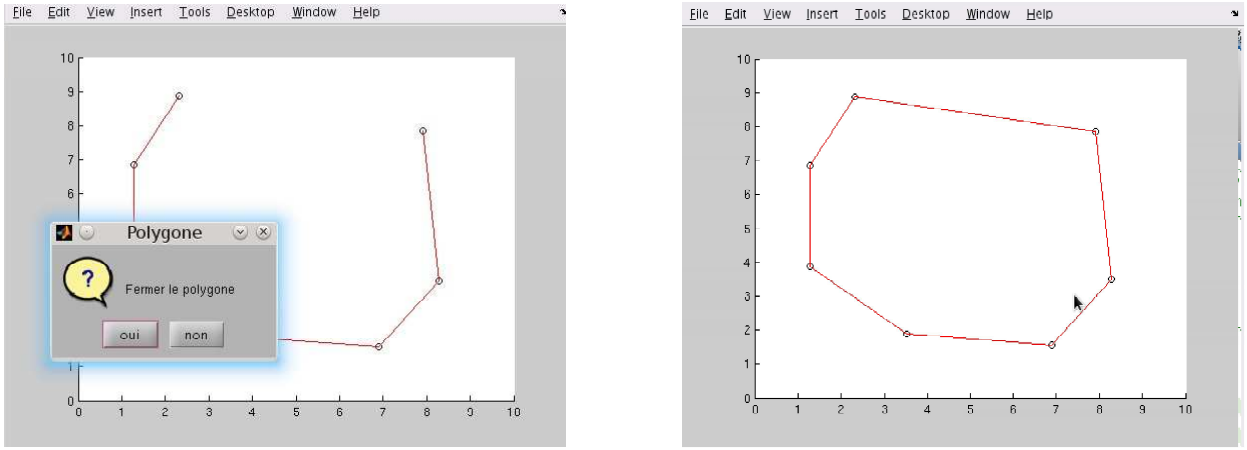
defaut : choix qui sera exercé si la touche **"entrée"** est appuyée.

Le programmeur utilisera cette information pour déterminer la course à prendre pour la suite du programme. Le programme à la Fig. 4.10 montre comment configurer la boîte de dialogue et la présentation des choix.

La première partie de cet exemple (La saisie des points) est identique á l'exemple précédent (Fig. 4.9), et donne une polyligne ouverte. La boite de fonction **questdlg** permet de poser une question á l'usager, soit de fermer ou non la polyligne pour obtenir un polygone. On note que dans ce dernier cas, on créé un nouveau point qui coincide avec le premier. Le résultat du déroulement de ce programme est montré a la Fig. 4.11.

```
% premiere partie du programme identique a l'exemple precedent
%------------------------------ Choix : polygone ou polyligne
message='Fermer le polygone';
titre='Polygone';
choix1='oui'; choix2='non';
defaut=choix1;%-- defaut lorsque la touche "retour" est appuyee
choix = questdlg(message,titre,choix1,choix2,defaut);
switch choix
    case choix1%---------------------------------------- ferme
        nbPLN =nbPLN+1;%---- point qui coincide avec le premier
        PLN(nbPLN,1:2) = PLN(1,1:2);
        figure(2)
        axis([0 10 0 10]);          hold on
        plot(PLN(:,1),PLN(:,2),'-r');plot(PLN(:,1),PLN(:,2),'ok');
        hold off
    case choix2%---------------------------------- ouvert
end
```

FIGURE 4.10 – Saisie de points á l'écran suivi d'un menu par boutons



(a) Choix (b) Résultat

FIGURE 4.11 – **questdlg** : Fenêtre du menu pour les choix

4.2 Ecriture et lecture

Les calculs effectués dans un programme Matlab peuvent être affichés ou acheminés vers la sortie standard qui est la fenêtre de commande, ou bien sauvegardés dans un fichier, comme illustré à la Fig. 4.12.

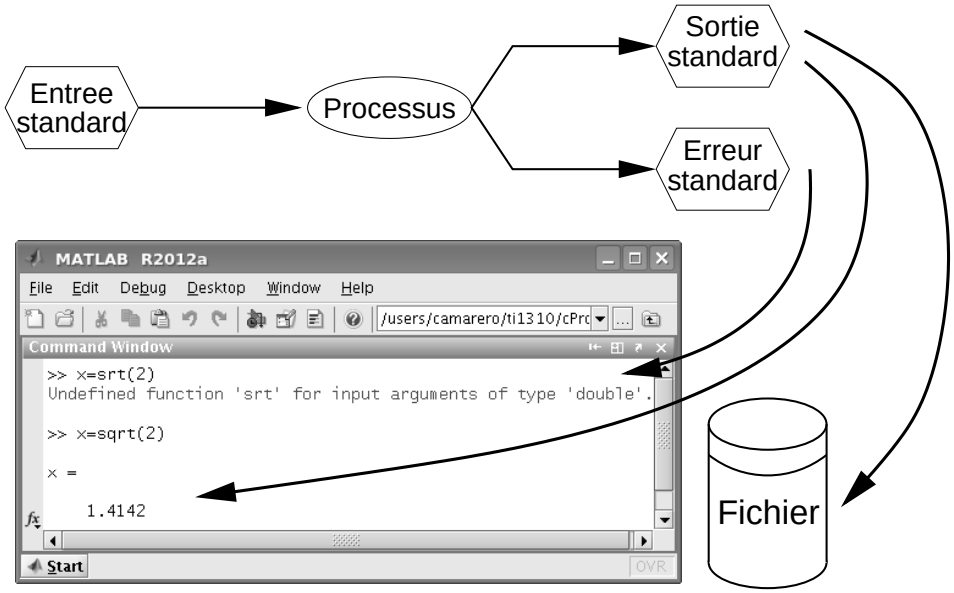


FIGURE 4.12 – Impression dirigée vers la sortie standard (fenêtre de commande) ou redirigée vers un fichier

Cette redirection de l'impression est spécifiée implicitement ou bien explicitement. Lorsque le point virgule est omis après un énoncé, l'expression est évaluée et, par défaut, le résultat est imprimé. Alors, la sortie par défaut est la fenêtre de commande.

Dans l'exemple ci-contre, un énoncé évalue l'expression **pi** sans point virgule " ;", ce qui imprime la valeur avec un titre **"ans ="**. Une autre situation est le calcul de l'aire d'un cercle de rayon **r=2**. L'énoncé affecte cette expression à la variable $aire = pi * r^2$, sans point virgule. Alors, le résultat est imprimé avec le nom de la variable, ici, **aire**.

```
>> pi

ans =
    3.1416

>> r=2;
>> aire=pi*r^2

aire =
   12.5664
>>
```

Dans l'exemple suivant, on montre comment les messages d'erreur (ici, **srt** au lieu de **sqrt**) sont acheminés vers la sortie standard (Fenêtre de commandes) :

```
>> x = srt(2)
Undefined function 'srt' for input arguments of type 'double'.
```

Par défaut, Matlab utilise le format court à 5 chiffres. D' autres formats sont indiqués au Tableau 4.1.

format long à 15 chiffres :

Énoncé	Description
format short	format court à 5 chiffres.
format long	format long à 15 chiffres.
format short e	format court à 5 chiffres avec notation en virgule flottante.
format long e	format long à 15 chiffres avec notation en virgule flottante.

TABLE 4.1 – Divers formats d'écriture de réels

```
>> format long
>> x = sqrt(2)
x =
   1.414213562373095
```

format short e à 5 chiffres avec notation en virgule flottante:

```
>> format short e
>> x = sqrt(2)
x =
   1.4142e+00
```

`Dans cette représentation, la mantisse comprend 5 chiffres.`

format long e à 15 chiffres avec notation en virgule flottante:

```
>> format long e
>> x = sqrt(2)
x =
     1.414213562373095e+00
```

On note que le format d'impression contrôle uniquement l'écriture d'une variable, et n'affecte pas la précision ou la représentation interne qui demeurent inchangées (Voir Section 2.1).

On peut explicitement rediriger les impressions vers la fenêtre de commande ou un fichier par des fonctions telles que **`disp`**, **`fprintf`** et **`sprintf`**, comme illustré à la Fig. 4.12.

4.2.1 `disp`

La fonction **`disp`** affiche dans la fenêtre de commande le contenu de la variable **`valeur`**

disp(valeur)

qui peut être une variable caractère (2.3), un entier (2.1.1) ou un réel (2.1.2).

L'exemple ci-dessous montre la différence entre une impression en omettant le point virgule, et une impression avec la fonction **disp**.

```
>> titre= 'Cours␣MEC1310'

titre =
Cours MEC1310

>> disp(titre)
Cours MEC1310
```

On note que dans le premier cas, le nom de la variable est imprimé, suivie de sa valeur. Tandis que la fonction **disp** imprime la valeur seulement.

Il est souvent utile d'imprimer un résultat avec un descriptif, qui comprend du texte et des valeurs de variables, par exemple,

L'aire d'un cercle de rayon = 2 est égale à 12.5664

On forme une variable caractère **message** par la concaténation de plusieurs chaînes de caractères, soit,

L'aire d'un cercle de rayon

la variable **r**

est égale à

la variable **aire**

On utilise de concaténation **[ ]**, comme montré ci-dssous ;

```
» r=2 ;
» aire=pi*r*r ;
» message=['L"aire d"un cercle de rayon = ',num2str(r),' est égale à',num2str(aire)] ;

» disp(message)

L'aire d'un cercle de rayon = 2 est égale à 12.5664
```

Remarques :

1. ***Les variables `r` et `aire`, qui sont des doubles, sont converties en variables caractères à l'aide de la fonction `num2str` pour que la variable `message` soit entièrement constituée de caractères, et ainsi rendre l'opération compatible.***

2. ***La fonction `disp` n'admet qu'une seule variable caractère en argument, sinon il faut utiliser les fonctions `sprintf` ou `fprintf`.***

4.2.2 `sprintf`

On peut obtenir le même résultat avec la fonction **`sprintf`** qui remplace l'opérateur `[ ]` pour construire la variable caractère **`message`**. On procède en déclarant une chaîne de caractères compris entre `' '`, suivie de la liste des variables à imprimer, **`variable1`**, **`variable2`**, ...

sprintf('message',variable1, variable2, ...)

Reprenant l'exemple précédent, on obtient, avec les variables **`r`** et **`aire`** :

```
» r=2 ;
» aire=pi*r*r ;
» message=sprintf('L"aire d"un cercle de rayon = %f est égale à %f', r, aire) ;

» disp(message) ;

L'aire d'un cercle de rayon = 2.000000 est égale à 12.566371
```

À l'intérieur de la chaîne de caractères **`message`**, on insère des caractères de contrôle `"%"` qui seront remplacés par les valeurs des variables données en arguments. Dans l'exemple précédent, chaque **`%f`** sera substitué par la valeur de la variable correspondante dans la liste. Dans cet exemple, **`%f`** indique que la variable à imprimer est de type réel (2.1.2). D'autres possibilités sont indiquées au Tableau 4.2 :

Caractères de contrôle	Description
%s	format pour une chaîne de caractères
%c	un seul caractère
%d	format pour un entier
%u	format pour un entier positif
%f	format en virgule flottante
%e	format en exponentiel, 0.12345e+01
%g	format en virgule flottante

TABLE 4.2 – Contrôle des formats d'écriture

On spécifie le nombre de chiffres de la façon suivante :

Entiers %12d indique qu'un champ de 12 sera réservé pour imprimer un entier ;

Réels %6.4f indique un champ de 6 chiffres dont 4 après la virgule(point) (le point est compté).

Champ La longueur du champ est un minimum qui sera augmenté automatiquement si la taille du nombre le nécessite.

4.2.3 `fprintf`

Cette fonction est semblable à **`sprintf`** sauf que l'écriture est dirigée dans un fichier, plutôt que vers la fenêtre de commande de Matlab. Par conséquent, on doit spécifier et ouvrir le fichier vers lequel seront dirigées les données. Il existe plusieurs variantes à cette fonction permettant l'écriture sur divers dispositifs et en mode binaire ou ASCII (text). Dans ce document, on retiendra ce dernier car il peut être lu avec un éditeur de texte.

1. On déclare/ouvre un fichier **`nom-de-fichier`** avec la fonction,

   identifiant=fopen('nom-de-fichier','permission')

   qui retourne **`identifiant`**, une variable qui se réfère au fichier. Les permissions sont illustrées à la Fig. 4.3.

Caractère de contrôle	Description
'r'	Ouvre le fichier en mode lecture (défaut)
'w'	Ouvre ou créé un nouveau fichier en mode écriture. (Écrase le contenu d'un fichier existant)
'a'	Ouvre ou créé un nouveau fichier en mode écriture. Ajoute les données à la fin du fichier.
'r+'	Ouvre le fichier en mode lecture et écriture.
'w+'	Ouvre ou créé un nouveau fichier en mode lecture et écriture. (Écrase le contenu d'un fichier existant)

TABLE 4.3 – Contrôle des permissions

2. On dirige l'écriture des variables **`variable1`**, **`variable2`**, ... vers ce fichier, disposées selon le format qui spécifie la mise en forme des données

   fprintf('identifiant','format', variable1, variable2, ...)

Si l'option **`identifiant`** est omise, alors la sortie sera dirigée vers la sortie standard (l'écran).

On illustre avec le programme de la Fig. 4.13 qui calcule le graphe de $y = log(x)$. Dans un premier temps, on calcule les coordonnées de la courbe, et, après l'ouverture du fichier **`nomFichier.txt`**, on écrit le nombre de points **`npts`**,

```
fprintf(idFichier,'%10.0d \n',npts)
```

suivi de l'écriture des coordonées (x_i, y_i), ligne par ligne,

```
for i=1:npts%------------------------ coordonnees de la courbe
    fprintf(idFichier,'%10.4f %10.4f \n',x(i),y(i));
end
```

Dans la spécification du format, il est important de faire le saut de ligne indiqué par

```
\n
```

Ce qui donne le résultat suivant,

```
    10
   1.0000     0.0000
   2.0000     0.6931
   ......     ......
   9.0000     2.1972
  10.0000     2.3026
```

```
%-----------------------------------------------------------------
% Ecriture/lecture vers un fichier
%-----------------------------------------------------------------
clc;clear all;close all
%-------------------------------------- Construction des donnees
xmin = 1; xmax = 10; npts = uint16(10);
dx   = (xmax-xmin)/double(npts-1);
x    = (xmin:dx:xmax);
y    = log(x);
figure(1)
hold on;
title('Donnees␣en␣ecriture'); xlabel('x'); ylabel('log(x)')
plot(x,y,'ob:')
hold off;
nomFichier='nomFichier.txt';%--------- Ecriture dans un fichier
idFichier = fopen(nomFichier,'w+');%------ Ouverture du fichier
fprintf(idFichier,'%10.0d␣\n',npts);%--------- nombre de points
for i=1:npts%------------------------ coordonnees de la courbe
    fprintf(idFichier,'%10.4f␣%10.4f␣\n',x(i),y(i));
end
fclose(idFichier);%---------------------- Fermeture du fichier
```

FIGURE 4.13 – Ecriture dans un fichier; **fprintf**

Dans l'exemple suivant, le programme de la Fig. 4.14, on propose deux modifications :

1. Un test sur l'ouverture du fichier ;
2. Une concaténation des coordonnées dans une même matrice **A = [x;y];**, ce qui permet de remplacer une écriture élément par élément, du programme précédent,

```
for i=1:npts
    fprintf(idFichier,'%10.4f %10.4f \n',x(i),y(i));
end
```

par une écriture sans boucle **for**,

```
fprintf(idFichier,'%10.4f %10.4f \n',A);
```

```
.....................................
x = (xmin:dx:xmax);y    = log(x);
A = [x;y];%- concatenation des coordonnees en une seule matrice
.....................................
nomFichier='nomFichier.txt';%--------- Ecriture dans un fichier
idFichier = fopen(nomFichier,'w+');%------ Ouverture du fichier
if idFichier==-1
    disp('Impossible de creer le fichier'); nomFichier
else
    fprintf(idFichier,'%10.0d \n',npts);%----- nombre de points
    fprintf(idFichier,'%10.4f %10.4f \n',A);
    fclose(idFichier);%------------------ Fermeture du fichier
end
```

FIGURE 4.14 – Ecriture dans un fichier; **fprintf** avec structure matricielle

4.2.4 fscanf

La fonction **fscanf** permet de lire des données à partir d'un fichier ASCII, et de les mettre en mémoire dans des variables d'un programme.

variable = fscanf(idFichier,format,dimensionA)

idFichier : identifiant du fichier en lecture spécifié par la fonction **fopen** ;

format : (Voir le tableau 4.2) ;

variable : les données sont converties dans une matrice **variable** selon les dimensions **dimensionA** ;

où **dimensionA** peut être un entier indiquant le nombre d'éléments à lire par rangée, ou bien sous la forme **[m n]**, **m** le nombre de colonnes (d'éléments par ligne du fichier), et **n** le nombre de rangées (lignes dans le fichier).

On illustre avec deux programmes, Fig. 4.15 et 4.16 qui lisent les données qui ont été écrites par les programmes Fig. 4.13 et 4.13, respectivement. Dans le programme de la Fig. 4.15, après l'ouverture du fichier **nomFichier** et la lecture du nombre de lignes,

```
n=fscanf(idFichier, '%g \n',1);
```

la fonction **fscanf** fait un saut de ligne,

```
\n
```

et lit 2 valeurs qui sont affectées à **CRB(i,1)** et **CRB(i,2)**, ligne par ligne, avec la boucle **for**, suivit d'un saut de ligne,

```
for i=1:n
    CRB(i,1:2)= fscanf(idFichier, '%g %g \n', 2);
end
```

Le programme de la Fig. 4.16 réalise la même action, mais procède différement, en utilisant directement la structure matricielle. En spécifiant la dimension de la matrice **[2 n]**, **fscanf** lit **n** lignes, et **2** valeurs par ligne. On note que les sauts de ligne sont fait automatiquement.

```
.....................................................................
idFichier = fopen(nomFichier);%---------- Ouverture du fichier
n=fscanf(idFichier, '%g␣\n',1);%--- lecture du nombre de lignes
for i=1:n
    CRB(i,1:2)= fscanf(idFichier, '%g␣%g␣\n', 2);
end
fclose(idFichier);
figure(2)
hold on;
title('Donnees␣en␣lecture'); xlabel('x'); ylabel('log(x)')
plot(CRB(:,1),CRB(:,2),'sr:')
hold off;
```

FIGURE 4.15 – Lecture d'un fichier: **fscanf** (Suite de Fig. 4.13)

```
.....................................................................
idFichier = fopen(nomFichier);%---------- ouverture du fichier
n=fscanf(idFichier, '%g',[1 1]);%-- lecture du nombre de lignes
CRB= fscanf(idFichier, '%g␣%g', [2 n]);
CRB=CRB';
fclose(idFichier);
figure(2)
hold on;
title('Donnees␣en␣lecture');
xlabel('x');
ylabel('log(x)')
plot(CRB(:,1),CRB(:,2),'sr:')
hold off;
```

FIGURE 4.16 – Lecture d'un fichier: **fscanf**(Suite de Fig. 4.14)

Dans les deux cas, on utilise le format **%g** ne sachant pas apriori le format d'écriture, et on affiche les coordonnées avec la fonction **plot** pour vérifier que les données ont été lues correctement.

4.3 Graphisme

Matlab dispose d'un ensemble varié et fonctionnel d'outils pour la production de graphiques.

4.3.1 plot(y)

La plus simple des fonctions graphiques affiche le graphe de y_i en fonction de l'indice du tableau ou vecteur y(i),c'est-à-dire i=1,2,3.....

plot(y)

Dans l'exemple suivant, on calcule les valeurs de $log(x)$ sur l'intervalle $.5 \leq x \leq 2$ avec un pas de $\delta x = .3$, et on trace le graphe avec la fonction **plot(y)**. Par défaut, le tracé est un trait plein en bleu.

Avec ce type de graphe, on note que l'abscisse est bien la suite 1,2,3.... et non pas la

```
>> x = .5:.3:2;
>> y = log(x)'

y =
    -0.6931
    -0.2231
     0.0953
     0.3365
     0.5306
     0.6931

>> plot(y)

>>
```

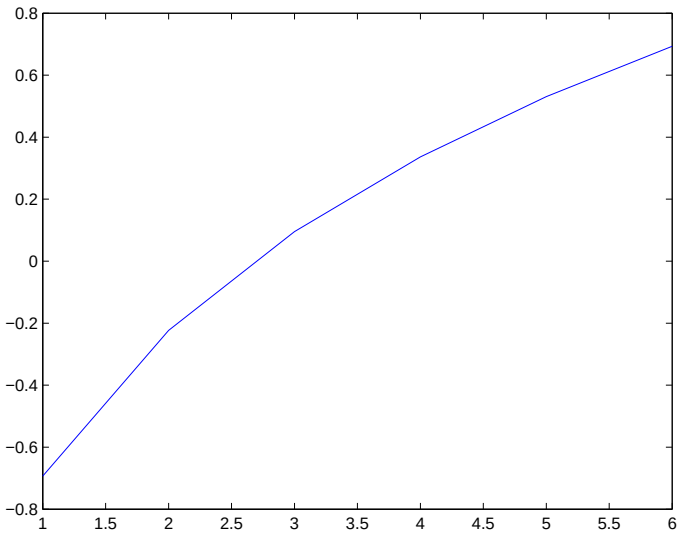


FIGURE 4.17 – Fonction **plot(y)**

variable x. Si y est une variable complexe, alors on affiche la partie imaginaire en fonction de la partie réelle,c'est-à-dire le même résultat que *plot(real(y), imag(y))*.

4.3.2 plot(x,y)

affiche le graphe de y en fonction de x.

Le graphe produit à la Fig. 4.17 de la section précédente est plutôt simple. On peut améliorer la présentation en spécifiant des paramètres pour contrôler le type de tracé et les axes. Les caractéristiques du tracé sont indiquées avec trois arguments, positionnés dans l'ordre suivant ; type de trait, symbole et la couleur qui sont précisées par trois lettres ou symboles `'- - -'`, donnés en arguments dans la fonction **`plot`**

plot(x,y, '- - -')

L'exemple de la Fig. 4.18 montre le programme pour le calcul et le tracé de la fonction $sin(x) - cos(5 * x)$. L'énoncé **`figure(n)`** ouvre un fichier qui contient toutes les instructions et données pour la production du graphe tel qu'illustré à la Fig. 4.19. Le paramètre **`n`** identifie le graphe parmi tous ceux qui peuvent être créés par le programme. Les énoncés **`hold on`** et **`hold off`** délimitent les commandes qui ajoutent des paramètres ou options du graphe. Finalement, les options dans la fonction **`plot`**, indiquent que le tracé sera en trait continu, "-", avec des symboles "o" à chaque point, et en couleur verte, "g".

```
%-----------------------------------------------------------------
% Calcul de la fonction sin(x)-cos(5*x)
% et trace le graphe avec des etiquettes
%-----------------------------------------------------------------
clear all; close all; clc
x=[0:pi/100:pi/2];%---------------------- calcul de la fonction
y=sin(x)-cos(5*x);
figure(1)
hold on%------------------------------- ouverture de la figure
xlabel('x') %------------------------------ titre de l'axe des x
ylabel('sin(x)-cos(5*x)')%---------------- titre de l'axe des y
plot(x,y,'-og')%------- trace en trait plein et cercles en vert
hold off%------------------------------- fermeture de la figure
```

FIGURE 4.18 – Programme pour le calcul et le tracé de la fonction $sin(x) - cos(5 * x)$

Une grande variété de symboles et choix de couleur (voir Tableau 4.4) sont disponibles pour la présentation de résultats. Une valeur par défaut est prise en l'absence de l'un de ces arguments.

Outre le tracé, la présentation d'un graphique utilise plusieurs libellés :

title : une chaîne de caractères qui sera affichée par défaut au-dessus du graphique ;

xlabel : une chaîne de caractères qui sera affichée le long de l'axe de l'absisse ;

ylabel : idem pour l'axe des ordonnées ;

legend :

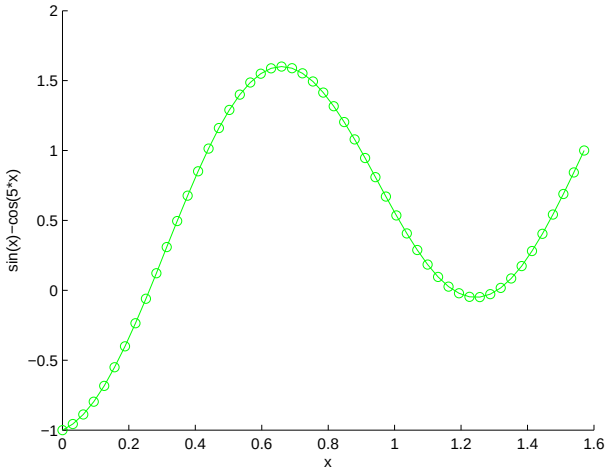


FIGURE 4.19 – Résultat du programme de la Fig. 4.18

Tracé		Symboles		Couleurs	
Trait plein	-	Plus	+	Rouge	r
Pointillé	:	Cercle	o	Vert	g
tirêt	–	Astérique	*	Bleu	b
	-.	Point	.	Cyan	c
		Croix	x	Magenta	m
		Carré	s	Jaune	y
		Losange	d	Noir	k
				Blanc	w

TABLE 4.4 – Éléments de présentation

La taille et le rapport de forme du graphique est automatiquement calculé par la fonction **`plot`**. Cependant, l'usager peut contrôler ces caractéristiques avec l'énoncé **`axis`** :

axis equal : force un rapport de 1 :1 entre les absisses et les ordonnées ;

axis (dimension) : force directement la taille des axes avec les valeurs du vecteur **`dimension= [xmin xmax ymin ymax]`** ;

grid on superpose un quadrillé pour faciliter la lecture des valeurs.

Tous ces paramètres, ainsi que tous les appels à la fonction **`plot`** se trouvent à l'intérieur des balises **`hold on`** et **`hold off`** et s'appliquent au graphique identifié par **`figure(n)`**.

4.3.3 plot3

La fonction **`plot3`** trace des courbes en dimension trois. (Voir Section 6.4.3)

4.3.4 Applications

Ces utilitaires de graphisme permettent, outre la présentation de résultats, d'analyser visuellement des fonctions. On illustre avec l'exemple suivant. Soit deux fonctions,

$$y(x) = 3 * sin(x)$$

et

$$z(x) = x$$

On affiche les graphes de ces fonctions et par inspection visuelle on peut estimer le point d'intersection de la courbe $y = 3 * sin(x)$ avec la droite $y = x$. Pour que cette approche soit efficace, il faut a priori avoir une idée de la plage à l'intérieur de laquelle se trouve l'intersection. Dans l'exemple illustré à la Fig. 4.20 ci-dessous, on estime que ce point se trouve entre $x = 0$ et $x = \pi$. Dans un premier temps, on peut prendre un intervalle assez grand, quitte à le raffiner après une première tentative.

```
%-----------------------------------------------------------------
% Estimation visuelle de l'intersection de deux courbes:
%   y(x)=3*sin(x)   et    z(x)=x
%-----------------------------------------------------------------
clc; clear all; close all
x = [0:pi/30:pi];%------------------------- calcul des fonctions
y = 3*sin(x); z = x;
figure(1)
hold on%---------------------------------- ouverture de la figure
title('Analyse de l''intersection de y(x)=3*sin(x) et z(x)=x')
xlabel('x'); ylabel('y')%---------------------- libelles des axes
grid on%---------------------------------- superpose une grille
plot(x,y)%------ trace par defaut: en trait plein et couleur bleu
plot(x,z)
hold off%--------------------------------- fermeture de la figure
```

FIGURE 4.20 – Estimation de l'intersection de deux courbes

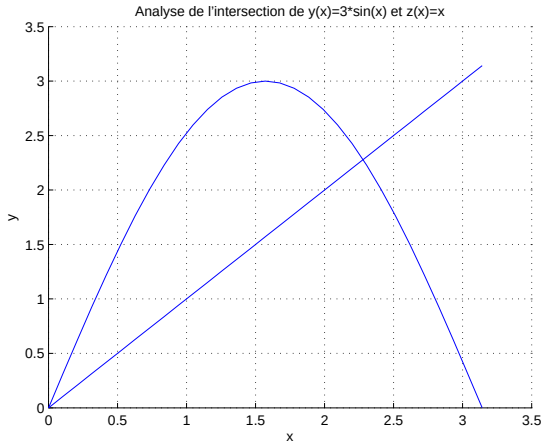
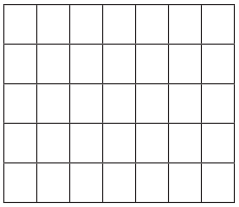


FIGURE 4.21 – Résultat du programme de la Fig. 4.20

Chapitre 5

Structures de données

Une structure de données est une organisation des données élémentaires en entités plus complexes qui enrichit le niveau de traitement de l'information. Dans la hiérarchie des structures de données utilisées dans le calcul, on retrouve les scalaires, les vecteurs et les matrices. Informatiquement, ces structures sont représentées par des tableaux de dimension **`1x1`**, **`nx1`** ou **`1xn`**, **`nxn`** et éventuellement des tableaux à trois dimensions **`nxnxn`**.



Dans MATLAB la structure de données de base est le tableau à deux dimensions que l'on appelle *matrice*. Les données sont les éléments d'une structure rectangulaire qui permet de les stocker et de les récupérer facilement. Ces éléments peuvent être des données de n'importe quel type (entiers, réels, caractères ...voir Chapitre 2).

Ainsi, une quantité scalaire comme une température sera représentée par une matrice **`1x1`**. Tandis que pour une vitesse, on utilisera une matrice **`1xn`** où les n éléments sont les composantes du vecteur dans l'espace à n dimensions. Dans ce chapitre, on montre comment utiliser la structure matrice **`nxn`** pour créer des entités qui reflètent les utilisations rencontrées dans les sciences de l'ingénieur riches et faciliter leur traitement. Toutes les opérations de l'algèbre matriciel font partie du langage Matlab.

5.1 Les matrices

On construit une variable de type matrice avec l'opérateur **`"[ ]"`**. Pour une matrice 1$\times$m, une rangée, on sépare les m éléments par un espace :

```
rangee = [e1 e2 e3.....em]
```

ou, par une virgule :

```
rangee = [e1,e2,e3,e4, .... em]
```

Par exemple, on peut créer un vecteur rangée, **`x = (3,4,5,6,7)`** de plusieurs façons :

1. Directement : en affectant chaque élément **`x(i)`** :

```
x(1) = 3;
x(2) = 4;
x(3) = 5;
x(4) = 6;
\noindentx(5) = 7;
```

ou bien

```
>> x = [-5.0, -2.5, 0.0, 2.5, 5.0, 7.5, 10.0]
>> x =
   -5.0   -2.50   0    2.50   5.00    7.50   10.00
```

2. Avec l'opérateur **"[ ]"** : Pour un vecteur rangée,

```
x = [3 4 5 6 7 8 9];
```

ou bien, pour un vecteur colonne,

```
x = [3;
     4;
     5;
     6;
     7;
     8;
     9];
```

On note que le deuxième exemple aurait pu être obtenu plus simplement par l'opération du transposé indiqué par `"'"`, c-à-d,

```
x = [3 4 5 6 7 8 9]';
```

3. Avec une boucle "for" :

```
for i=1:1:5
   x(i) = 3 + (i-1);
end
```

4. Avec l'opérateur " :" :

```
x = 3:1:5;
```

Ce qui donne le même résultat que l'exemple précédant, mais est plus efficace. On note que dans les deux derniers cas, l'intervalle (ou le pas) entre les éléments a été posé à 1, et en général peut prendre n'importe quelle valeur.

On créé une matrice par une suite de rangées, séparées par un point virgule :

```
matrice = [ rangee1;
            rangee2;
            rangee3;
            ......
            .....
            rangeeN]
```

On peut imaginer la matrice comme une pile de rangées. Avec cette syntaxe, Matlab "comprend" l'objet matrice et l'affiche selon les conventions mathématiques.

```
>> p = [12 23 56 9 11;34 67 89 21 90;45 23 76 43 1]
p =
    12    23    56     9    11
    34    67    89    21    90
    45    23    76    43     1
```

ou, pour une meilleure lisibilité,

```
>> p = [12 23 56  9 11;
        34 67 89 21 90;
        45 23 76 43  1]
p =
    12    23    56     9    11
    34    67    89    21    90
    45    23    76    43     1
>>
```

Une matrice peut contenir des données de type numérique, logique, caractère... Par exemple :

```
>> unePersonne = ['Nom       ';
                  'Prenom    ';
                  'Adresse   ';
                  'Profession']
unePersonne =
Nom
Prenom
Adresse
Profession
>>
```

On peut référencer un élément **(i,j)** d'une matrice **p**, par l'énoncé **p(i,j)** qui désigne l'élément de la ième ligne et de la jème colonne. Par exemple, pour la matrice 3×5 l'élément **p(2,4)** est égal à 21, c'est-à-dire **p(ligne,colonne)**.

			j		
	12	23	56	9	11
i	34	67	89	21	90
	45	23	76	43	1

5.2 L'opérateur " :"

Dans les exemples précédents, un syntaxe particulière, " **:**" a été utilisée pour construire des suites d'éléments compris entre deux valeurs et espacés par un intervalle régulier :

valeur initiale : pas : valeur finale

On la retrouve dans les boucles **for** :

```
>> for i=1:1:7
>> x(i) = -5.0 + (i-1)*2.5;
>> end
>> x
x =
   -5.0   -2.50    0    2.50   5.00    7.50   10.00
```

ou bien avec l'opérateur **"[ ]"** :

```
>> x = [-5.0:2.5:10.0]
x =
  -5.00  -2.50    0    2.50    5.00    7.50   10.00
>>
```

Les bornes et l'intervalle (le pas) peuvent être quelconques, comme montré ci-dessous :

```
>> angle = [pi:pi/6:2*pi]
angle =
3.142   3.665   4.189   4.712   5.236   5.760   6.283
>>
```

On note cependant que le partage de la plage des valeurs est régulier ou constant.

5.3 Concaténation

La concaténation est une opération qui permet de construire une nouvelle matrice à partir d'une ou plusieurs matrices. Soit deux matrices, A (3x5) et B (2x5),

```
A=
    12    23    56     9    11
    34    67    89    21    90
    45    23    76    43     1


B=
    14    63    96     4    22
    64    77    81    52    82
```

On construit une matrice C avec l'opérateur **"[ ]"**, C=[A ;B], ce qui donne,

```
C=
    12    23    56     9    11
    34    67    89    21    90
    45    23    76    43     1
    14    63    96     4    22
    64    77    81    52    82
```

On représente schématiquement cette opération

```
A(3x5)    12    23    56     9    11
          34    67    89    21    90
          45    23    76    43     1
   ;                    ;
B(2x5)    14    63    96     4    22
          64    77    81    52    82
   =                    =
C(5x5)    12    23    56     9    11
          34    67    89    21    90
          45    23    76    43     1
          14    63    96     4    22
          64    77    81    52    82
```

Cette concaténation à la verticale assemble les rangées de A et de B, et par conséquent exige que le nombre de colonnes soit le même.

L'opération de concaténation peut aussi se faire dans le sens horizontal en assemblant les colonnes. Soit deux matrices, M (3x2) et N (3x3)

```
M=
    12    23
    34    67
    45    23
```

```
N=
    14    63    96
    64    77    81
     4    22    82
```

Avec l'opérateur **"[ ]"**, L=[M N], on obtient,

```
L=
    12    23    14    63    96
    34    67    64    77    81
    45    23     4    22    82
```

Ici, c'est le nombre de rangées qui doit être le même. Schématiquement,

```
[   M              N      ]  =          L
 12  23        14   63   96       12  23  14  63  96
 34  67        64   77   81       34  67  64  77  81
 45  23         4   22   82       45  23   4  22  82
  (3x2)          (3x3)                (3x5)
```

5.4 Scalaire vs vecteur

La structure de données de base dans Matlab permet de traiter des quantités complexes comme les vecteurs ou les matrices aussi simplement que les scalaires. Par exemple :

Affectation d'une variable scalaire :

```
x = 3;
y = log(x);
x,y
x =      3
y =    1.0986
>>
```

Affectation des éléments d'une variable vecteur, élément par élément :

```
x(1) = 3;
x(2) = 4;
x(3) = 5;
x(4) = 6;
x(5) = 7;
y(1) = log(x(1));
```



```
y(2) = log(x(2));
y(3) = log(x(3));
y(4) = log(x(4));
y(5) = log(x(5));
x,y
x =
    3     4     5     6     7
y =
    1.0986    1.3863    1.6094    1.7918    1.9459
>>
```

Affectation des éléments d'une variable vecteur, boucle "for" :

```
for i=1:5
   x(i) = 3 + (i-1);
   y(i) = log(x(i));
end
```

Affectation d'une variable vecteur, avec la structure Matlab :

```
x = [3 4 5 6 7 8 9];
y = log(x);
```

Déclaration d'une variable vecteur :

```
x = 3:1:5;
y = log(x);
```

Dans les deux derniers exemples, la variable `x` est déclarée comme un vecteur. Matlab le reconnaît et lors du calcul de la variable `y`, automatiquement celle-ci est construite comme un vecteur de même dimension, où chaque élément `y(i)` est construit comme `y(i)=log(x(i))`. La comparaison entre ces différentes façons illustre bien la puissance et l'élégance de cette structure de données.

5.5 Scalaire vs matrice

Avec la structure de données matricielle, on peut manipuler des entités encore plus complexes que les vecteurs. Par exemple, un objet comme un triangle est décrit par ses trois sommets (x_1, y_1), (x_2, y_2) et (x_3, y_3). Il existe plusieurs structures de données pour représenter un tel objet. Nous allons les illustrer par une suite de représentations différentes, en ordre croissant de simplicité, allant des scalaires, aux vecteurs et pour terminer avec les matrices.

Structure scalaire Dans l'exemple ci-dessous, on illustre une représentation de cet objet,

(x_1, y_1), (x_2, y_2) et (x_3, y_3), avec une structure scalaire. Ceci nécessite l'affectation de six variables, une pour chaque des coordonnées x_i, y_i, et l'affichage nécessite trois appels à **plot**. (La fonction **axis** sert à forcer la dimension du graphe, plutôt que de laisser la fonction faire le choix de la taille des axes).

```
%--------------------------- structure scalaire
x1=1;x2=2;x3=3;
y1=1;y2=3;y3=2;
figure(1)
hold on
xlabel('x')
ylabel('y')
plot(x1,y1,'-ob')
plot(x2,y2,'-ob')
plot(x3,y3,'-ob')
axis([0 5 0 5])
hold off
```

Structure vectorielle On obtient une meilleure fonctionnalité en déclarant une structure vectorielle qui représente un point, et où les éléments correspondent aux coordonnées en x et y. Ainsi, une telle variable contient les deux composantes, et peut se manipuler avec les opérations de l'algèbre vectoriel.

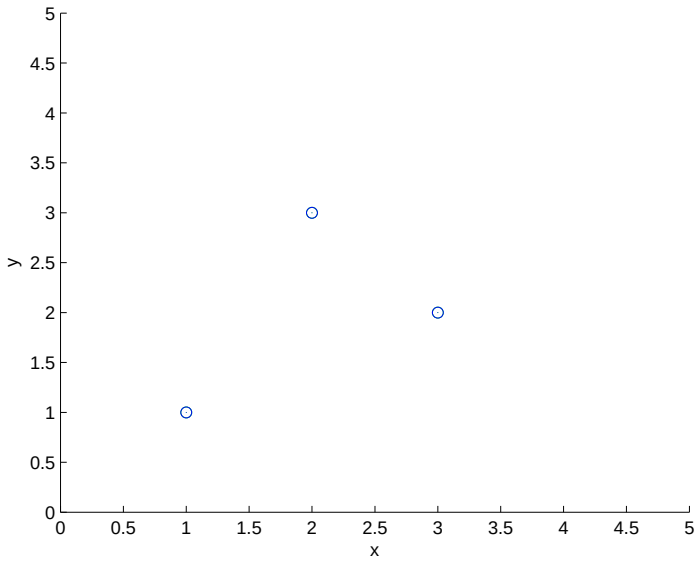
```
p1 = [x1 y1];%-------------------- structure vectorielle
p2 = [x2 y2];
p3 = [x3 y3];
figure(2)
hold on
xlabel('x')
ylabel('y')
plot(p1(1),p1(2),'-sb')
plot(p2(1),p2(2),'-sb')
plot(p3(1),p3(2),'-sb')
axis([0 5 0 5])
hold off
```

Structure matricielle Finalement l'objet au complet peut se représenter comme un empilement de plusieurs points, structure qui se traduit par une matrice :

```
t = [p1;p2;p3];%-------------------- structure matricielle
figure(3)
hold on
xlabel('x')
```

```
ylabel('y')
for point=1:3
    plot(t(point,1),t(point,2),'xk')
end
% plot(t(:,1),t(:,2),'o')
axis([0 5 0 5])
hold off
```

Évidemment, pour ces trois exemples on obtient le même résultat montré ci-dessous.



5.6 Calcul vectoriel

Toutes les opérations de l'algèbre vectorielle classique font partie intégrante du langage Matlab de par ses structures de données de base ainsi que l'extension des opérateurs d'addition/soustraction (**+**/**-**), multiplication/division par un scalaire (***** et /), le produit scalaire et le produit vectoriel. Soit trois vecteurs a, b et c :

```
a = [ 2, pi/6];
b = [ 3, pi/8];
c = [ 1, 2, 3];
```

Les opérations d'addition/soustraction et de multiplication/division par un scalaire se font composante par composante :

```
d = pi*a + b/3
d =
    7.2832    1.7758
```

Il est important de noter que pour ces opérations, la dimension des vecteurs doit être la même. Par exemple,

```
>> a + b
ans =
    5.0000    0.9163

>> a + c
Error using  +
Matrix dimensions must agree.
```

Produit scalaire Le produit scalaire de deux vecteurs, $\vec{a}$ et $\vec{b}$, où $\vec{a} = (a_1, a_2, a_i....a_n)$ et $\vec{a} = (b_1, b_2, b_i....b_n)$ donne un scalaire égal à la somme des produits des composantes :

$$\begin{aligned} \vec{a}\cdot\vec{b} &= \sum_{i=1}^{i=n} a_i b_i \\ &= a(1)*b(1) + a(2)*b(2) + ... \\ &= |a||b|cos(\theta) \end{aligned}$$

où n est la dimension des vecteurs.

Dans Matlab, ces vecteurs sont construits avec les expressions $a = [2, pi/6];$ et $b = [3, pi/8];$ ci-dessus, et le produit scalaire s'écrit :

```
e = a*b'

e =
    6.2056
```

On réitère que la dimension des vecteurs soit la même lors de la multiplication de différents vecteurs.

```
>> e = a*b'
e =
    6.2056

>> format long
>> e
e =
   6.205616758356029

>> a*c'
Error using  *
Inner matrix dimensions must agree.
>>
```

Norme d'un vecteur :

```
>> f=[1 3 7];
>> norme =sqrt(f*f')
norme =
    7.6811
```

Produit vectoriel : Le produit vectoriel de $\vec{a}$ et $\vec{b}$ est un vecteur :

$$\begin{aligned}
\vec{a}\times\vec{b} &= \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a(1) & a(2) & a(3) \\ b(1) & b(2) & b(3) \end{vmatrix} \\
&= \vec{i}\begin{vmatrix} a(2) & a(3) \\ b(2) & b(3) \end{vmatrix} - \vec{j}\begin{vmatrix} a(1) & a(3) \\ b(1) & b(3) \end{vmatrix} + \vec{k}\begin{vmatrix} a(1) & a(2) \\ b(1) & b(2) \end{vmatrix} \\
&= |a||b|sin(\theta)
\end{aligned}$$

où $\vec{i}$, $\vec{j}$ et $\vec{k}$ sont les vecteurs unitaires dans les directions x, y et z, respectivement, et où θ est l'angle entre les deux vecteurs $\vec{a}$ et $\vec{b}$. En dimension 2, ce vecteur est perpendiculaire au plan xy,

$$\begin{aligned}
a\times b &= \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a(1) & a(2) & 0 \\ b(1) & b(2) & 0 \end{vmatrix} \\
&= \vec{i}\cdot 0 - \vec{j}\cdot 0 + \vec{k}\begin{vmatrix} a(1) & a(2) \\ b(1) & b(2) \end{vmatrix} \\
&= \vec{k}(a(1)*b(2) - a(2)*b(1)) \\
&= \vec{k}|a||b|sin(\theta)
\end{aligned}$$

Une commande MATLAB permet de réaliser ce calcul directement,

```
cross(a,b)
```

5.7 Calcul matriciel

Par défaut, les variables de Matlab sont des matrices, et le scalaire est un cas particulier, c'est-à-dire une matrice de 1×1. Donc toutes les opérations arithmétiques d'addition/soustraction, multiplication, puissance s'appliquent dans la mesure où elles sont conformes à l'algèbre matriciel.

`Addition` La somme de deux matrices A et B n'a de sens que si elles ont les mêmes dimensions :

```
>> A = [1 2 3;
        4 6 7];
>> B = [5 7;
        9 1];
>> C = A + B
??? Error using ==> plus
Matrix dimensions must agree.
>>
```

L'erreur découle du fait que A est une matrice de 2×3 tandis que B est de dimension 2×2.

`Multiplication` Le produit de deux variables est possible si les dimensions des matrices A et B sont compatibles avec l'opération matricielle. Pour le produit B*A on obtient :

```
>> B*A
ans =
    33    52    64
    13    24    34
>>
```

Par contre, le produit A*B n'est pas permis :

```
>> A*B
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

`Transposée` On obtient la transposée de la matrice (M), avec (M)',

```
>> C = (B*A)'
C =
    33    13
    52    24
    64    34
>>
```

`Opérations élément par élément` En plus des opérations de l'algèbre, il est possible d'effectuer des opérations élément par élément. On indique cette opération en faisant précéder l'opérateur par un point. Par exemple, la multiplation `*` de deux vecteurs, $x^2 = x * x$ nécessite la multiplication de x par lui-même. Lorsque x est un scalaire, l'opération est permise, mais lorsque x est un vecteur, alors cette opération peut poser problème. On illustre ci-dessous la différence entre les opérateurs `*` et `.*` :

x*x Cette opération n'a pas de sens, et n'est pas permise :

```
>> x=[2 5 8 1];
>> x*x Error
   using * Inner matrix dimensions must agree.
```

x*x' Par contre, le produit scalaire x*x' est autorisé, et donne un scalaire :

```
>> x*x'
   ans =
        94
```

x.*x Il existe de nombreuses situations où l'on veut le produit élément par élément des vecteurs, plutôt que vecteur par vecteur :

```
>> x.*x
   ans =
        4    25    64     1
```

De facon semblable, le produit A*B n'est pas permis, tandis que A.*B l'est.

```
>> A = [1 2 3;
        4 6 7];
>> B = [ 33    52    64;
         13    24    34];
>> C = A.*B
   C =
      33   104   192
      52 144 238 >>
```

On note que cette opération s'applique également à

```
^ et .^
```

Déterminant La fonction **det** calcule le déterminant d'une matrice nxn,

```
det(M)
```

ones créé une matrice de nXn éléments de 1,

```
X=ones(n)
```

ou bien une matrice mXn de 1,

```
M=ones(m,n)
```

Pour un vecteur colonne de m éléments de 1,

```
M=ones(m,1)
```

zeros créé une matrice nXn de 0,

```
X=zeros(n)
```

ou bien, créé une matrice mXn de 0,

```
M=zeros(m,n)
```

5.8 Résolution de systèmes d'équations

Il existe un grand nombre de problèmes en ingénierie qui s'expriment sous la forme d'un système d'équations algébriques de la forme,

$$A * X = B$$

où A est une matrice de $n \times n$ coefficients, et où B un vecteur colonne de $n \times 1$ coefficients. Formellement, la solution x peut s'exprimer par,

$$X = A^{-1}B$$

En pratique, il ne faut **jamais** évaluer l'inverse d'une matrice pour résoudre un système d'équations. Il est beaucoup plus efficace de le faire par une décomposition de Gauss, qui sous Matlab se fait par l'opérateur "\" :

$$X = A\backslash B$$

Un exemple simple est de trouver l'intersection de deux droites dont les équations sont

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

Connaissant les coefficients a_{ij} et b_i, le point $X = (x, y)$ se trouve par la résolution du système sous forme matricielle,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

$$A * X = B$$

Appliquant cette méthode au problème illustré à la Fig. 5.1, on construit la matrice A et le vecteur B dans Matlab à partir de,

$$\begin{pmatrix} .5 & -1 \\ .5 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -2 \\ 7 \end{pmatrix}$$

La solution $X = (x, y)$ s'obtient de $X = A\backslash B$.

```
>>
>> A=[.5  -1;
      .5   1];
>> B=[-2  7]';
>> X = A\B

X =
    5.0
    4.5
```

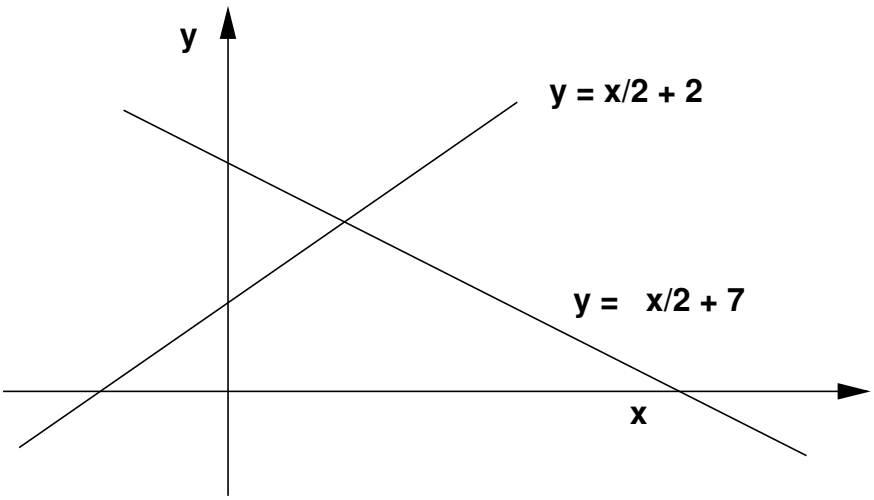


FIGURE 5.1 – Intersection de deux droites par la résolution d'un système d'équations 2x2

Chapitre 6

Applications

6.1 Représentation des courbes

Une courbe est un ensemble de points dont les coordonnées vérifient une relation de la forme,

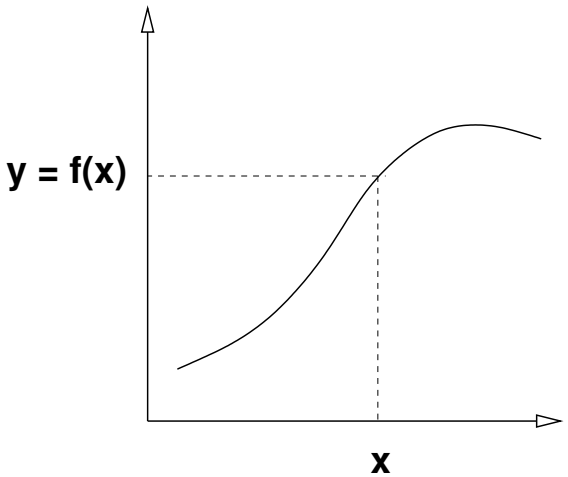
$$F(\vec{x}) = 0$$

où $\vec{x}$ représente une position (x, y) en dimension 2, ou (x, y, z) en dimension 3.

Il existe plusieurs types de représentation de courbes :

Représentation explicite Pour un point $\vec{P} = (x, y, z)$ sur la courbe, une des coordonnées est exprimée explicitement en fonction des autres coordonnées :

$$\begin{aligned} y &= f(x) \\ \text{ou bien} \quad z &= g(x, y) \end{aligned}$$



Pour une représentation explicite, en variant x, on engendre y avec $y = f(x)$, et ainsi l'ensemble des points constituant la courbe.

Cette représentation est bien adaptée. à la production de graphes, mais présente un nombre d'inconvénients pour certains types de courbes ; courbes fermées, avec des points de rebroussement ou bien avec une pente infinie.

Représentation implicite La forme implicite décrit une courbe comme l'ensemble des points dont les coordonnées vérifient une relation de la forme suivante :

$$\begin{aligned} F(x, y) &= 0 \\ \text{ou bien} \quad G(x, y, z) &= 0 \end{aligned}$$

Représentation paramétrique La forme paramétrique représente une courbe par un ensemble de relations (une par coordonnée) qui donnent les valeurs de celles-ci explicitement par une fonction d'un paramètre. La position d'un point $\vec{P} = (x, y, z)$ est donnée par ses coordonnées x, y et z où,

$$\begin{aligned} x &= f(u) \\ y &= g(u) \\ z &= h(u) \end{aligned}$$

sont des fonctions explicites du paramètre u qui varie sur l'intervalle de définition que l'on prend, sans perte de généralité :

$$0 \leq u \leq 1$$

En parcourant la plage du paramètre u, on associe trois coordonnées, x, y et z à chaque valeur de u ; ce qui engendre l'ensemble des points de la courbe.

6.2 Représentation explicite

6.2.1 Fonctions polynômiales

Les polynômes sont une illustration de la représentation explicite. Comme exemple, l'évaluation du polynôme,

$$y = x^3 - x$$

est illustrée par le programme à la Fig. 6.1 qui calcule les coordonnées y en fonction des ordonnées x. Le calcul de x porté à une puissance pose problème lorsque x est un vecteur (Voir Section5.7). On montre deux façons de faire ce type de calcul,

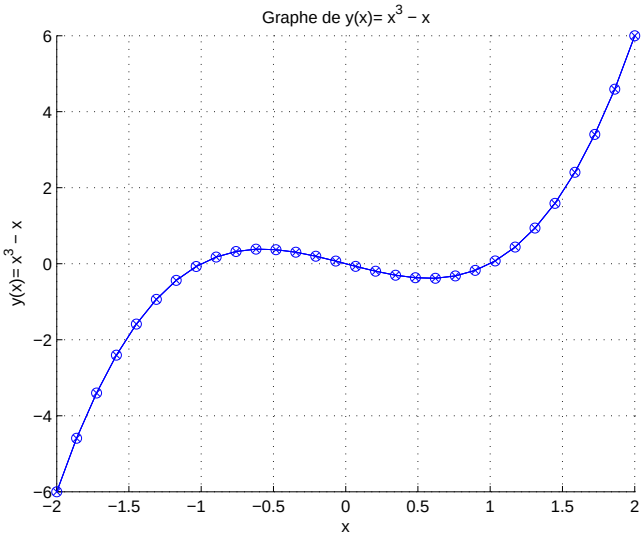
1. avec une boucle **for**

```
for i=1:npts
    y(i)=x(i)^3-x(i)
end
```

2. en utilisant les calculs élément par élément avec "." précédant l'opérateur.

```
y=x.^3-x;
```

Le résultat est montré à la figure ci-contre.



```
%-------------------------------------------------------------------------
% Representation explicite de la fonction y = x^3 -x
%-------------------------------------------------------------------------
clc; clear all; close all
%-------------------------------------------------------------------------
xIni = -2;%------------------------------------- position initiale
xFin = 2;%-------------------------------------- position finale
npts = 30;%------------------------------------- nombre de points
dx   = (xFin - xIni)/(npts-1);%---------- calcul de l'intervalle en x
x    = [xIni:dx:xFin];%------------------------------ calcul des x
%----------------------------- calcul des y Methode 1: boucle "for"
for i=1:npts
    y(i) = x(i)^3 - x(i) ;
end
% calcul des yy Methode 2: structure matricielle avec l'operateur "."
yy=x.^3-x;
figure(1)%----------------- Affiche y en trait plein et yy avec des o
hold on
grid on
title('Graphe de y(x)= x^3 - x')
xlabel('x')
ylabel('y(x)= x^3 - x')
plot(x,y,'-');
plot(x,yy,'o');
hold off
```

FIGURE 6.1 – Programme pour calculer et dessiner le graphe de $y(x) = x^3 - x$

La forme générale d'un polynôme de degré n se présente comme la somme de puissances de la variable x :

$$P_n(x) = A_1 x^n + A_2 x^{n-1} + \ldots + A_{n-1} x^2 + A_n x^1 + A_{n+1}$$

Bien que pratique du point de vue mathématique, sur le plan informatique cette écriture présente quelques difficultés. Elle nécessite n additions, $n+1$ multiplications et l'évaluation de $n+1$ puissances de x, équivalentes à $n^2/2$ multiplications.

On remarque qu'il n'est pas nécessaire d'évaluer les termes x^i séparément, car lors de la sommation, chacun peut être obtenu par une simple multiplication du terme précédent,

$$x^i = (x^{i-1})x$$

Après cette constatation, on peut réécrire le polynôme sous la forme,

$$\begin{aligned} P_n(x) &= A_1 x^n + A_2 x^{n-1} + \ldots + A_{n-1} x^2 + A_n x^1 + A_{n+1} \\ &= ((((A_1 x + A_2)x + A_3)x + A_4)x \ldots + A_{n+1} \end{aligned}$$

Cette sommation imbriquée s'appelle l'algorithme d'Horner et consiste à obtenir une suite de polynômes avec la formule de récursion suivante,

$$Q_i = Q_{i-1}x + A_i$$

en partant avec $Q_1 = A_1$, et en appliquant successivement pour $i = 1, 2, \ldots, n$ donne $P_n(x)$. Ce qui revient à n multiplications et n additions, une réduction considérable du nombre d'opérations arithmétiques.

La mise en oeuvre de cet algorithme est illustrée à la Fig. 6.2.

```
function Q = horner(degre,coeff,x)
% Pn(x)= An+1x^0 + A1x^1 + A2x^2 +  .... + A1x^n
%      = (((((((((A1x + A2)x + A3)x + A4)x ..... +An+1
n=degre+1;
Q=coeff(1);
for k=2:n
    Q= Q*x + coeff(k);
end
```

FIGURE 6.2 – Fonction pour calculer un polynôme, $P_n(x)$, de degré n avec l'algorithme de Horner.

Matlab propose une fonction, **`polyval(A,x)`** pour évaluer un polynôme $P(x)$, qui est une variante optimisée de la formule de Horner.

polyval(A,x)

où A représente les coefficients du polynôme, en ordre décroissant de la puissance de x, et x le vecteur des positions.

$$P_n(x) \quad = \quad A_1x^n + A_2x^{n-1} + \ldots + A_2x^2 + A_nx^1 + A_{n+1}$$

Le programme suivant, à la Fig. 6.4, évalue le polynôme $y = x^3 - x$ de l'exemple montré à la Fig. 6.1, avec la formule de Horner (tracé en pointillé) et la fonction **`polyval`** (tracé en cercles). Les graphes sont identiques et sont montrés à la Fig. 6.3.

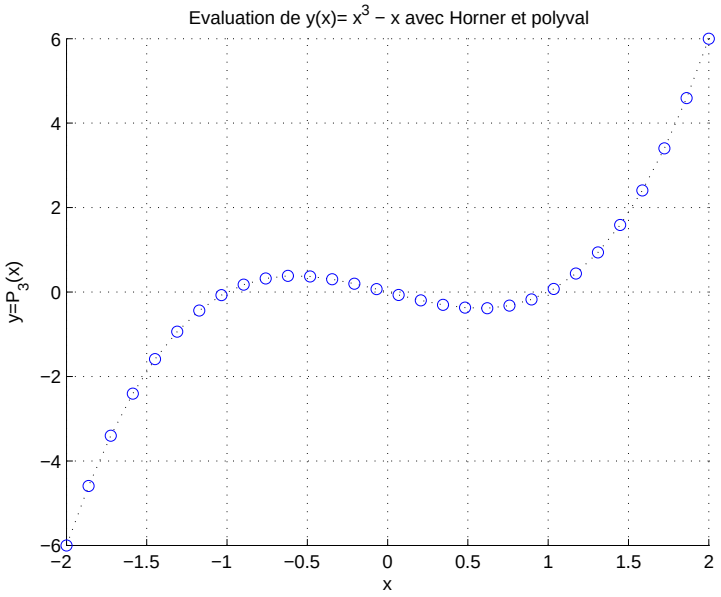


FIGURE 6.3 – Comparaison des méthodes de Horner et de la fonction **polyval**

```
%-----------------------------------------------------------------
% Evaluation du polynome y= Pn(x) par une representation
%   Pn(x)= A1x^n + .......          + A2x^2 + A1x^1 + An+1
%-----------------------------------------------------------------
clc; clear all; close all
% Calcul des x ------------------------------------------------
xIni = -2; xFin = 2; npts = 30;
dx=(xFin - xIni)/(npts-1); x = [xIni:dx:xFin];
degre=3; A=[1 0 -1 0];%-------------- specificaton coefficients
for i=1:npts%------------------------------- Methode de Horner
    y(i)=horner(degre,A,x(i)) ;
end
yy=polyval(A,x);%------------------- fonction polyval de matlab
figure(1)%----  Affiche  y: en pointille  et yy avec des cercle
hold on; grid on
title('Evaluation de y(x)= x^3 - x avec Horner et polyval')
xlabel('x'); ylabel('y=P_3(x)')
plot(x,y,':k'); plot(x,yy,'ob');
hold off
```

FIGURE 6.4 – Programme pour évaluer un polynôme de degré n avec Horner et **polyval**.

6.2.2 Méthode de Vandermond

Cette manière de procéder suppose que les coefficients A_i sont connus, ce qui permet d'utiliser la forme explicite de la représentation polynômiale. Une autre approche, plus intuitive et que l'on retrouve dans la pratique, est de construire le polynôme à partir de points (x_i, y_i) qui se trouvent sur la courbe. Tel qu'illustré à la Fig. 6.5, avec $n+1$ points, (x_j, y_j) où $j = 0, 1, 2, \ldots n$, on peut définir de façon unique un polynôme de degré n.

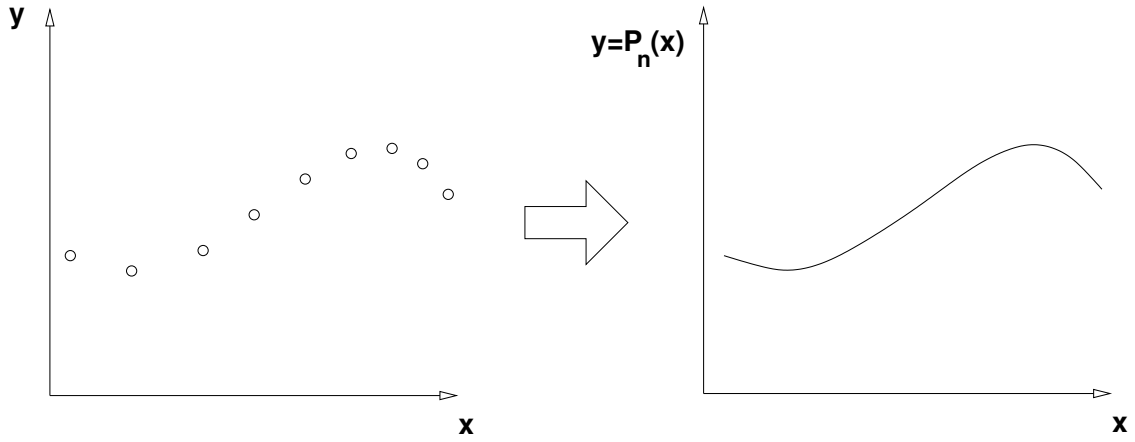


FIGURE 6.5 – Construction d'un polynôme de degré n à partir de $n+1$ points

On illustre avec l'exemple suivant, où on utilise les 3 points de collocation

x_i	1	4	9
y_i	1	2	3

Avec les coordonnées (x_i, y_i) de chacun de ces 3 points, on peut écrire une équation de la forme,

$$P_2(x) = A_1 x^2 + A_2 x^1 + A_3$$

Ce qui donne les trois équations suivantes,

$$\begin{aligned} y_1 &= A_1 x_1^2 + A_2 x_1 + A_3 \\ y_2 &= A_1 x_2^2 + A_2 x_2 + A_3 \\ y_3 &= A_1 x_3^2 + A_2 x_3 + A_3 \end{aligned}$$

qui s'écrivent, sous forme matricielle,

$$M \cdot A = B$$

ou, en explicitant les temes M, A et B,

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Les coefficients A_i sont obtenus par la résolution de ce système, utilisant une décomposition de Gauss, qui sous Matlab se fait par l'opérateur "\", $A = M \backslash B$, ou bien,

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \backslash \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Ces calculs sont réalisés par le programme donné à la Fig. 6.6 qui en montre le détail. La première étape est l'évaluation de la matrice M en substituant les (x_i, y_i) par leurs valeurs numériques,

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 16 & 4 & 1 \\ 81 & 9 & 1 \end{bmatrix} \backslash \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

On montre deux facons :

1. élément par élément

```
xPoint=[1 4 9];%---------------------- points de collocation
yPoint=[1; 2; 3];
M = [ xPoint(1)^2 xPoint(1) 1;%--- construction de la matrice
      xPoint(2)^2 xPoint(2) 1;
      xPoint(3)^2 xPoint(3) 1];
A= M\yPoint;%------------ calcul des coefficients du polynome
```

2. structure vectorielle suivie de la transposée

```
xPoint=[1 4 9];%---------------------- points de collocation
yPoint=[1 2 3]';
M = [ xPoint.*xPoint;%------------ construction de la matrice
              xPoint;
               1 1 1]';
A= M\yPoint;%------------ calcul des coefficients du polynome
```

Une analyse de ces fragments de programme montre que la deuxième méthode est la plus efficace

La Fig. 6.7 illustre graphiquement le résultat de cette démarche pour un polynôme de degré deux, obtenu avec les trois points ci-dessus.

```
%-------------------------------------------------------------
% Construction du polynome P(x) par la methode de Vandermond
% a partir de (n+1) points (xi,yi). Les coefficients Ai sont
% obtenus par la resolution du systeme A= X\Y
% Le polynome est evalue par la fonction polyval ou Horner
%-------------------------------------------------------------
clear all; clc; close all
%-------------------------------------------------------------
xPoint = [1 4 9];%---- points de collocation
yPoint = [1 2 3]';
M = [ xPoint.*xPoint;%-------------- construction de la matrice
              xPoint;
               1 1 1]';
A= M\yPoint;%-------------- calcul des coefficients du polynome
npts = uint16(10);%---------------- constructuction du polynome
xIni = 0.0; xFin = 10.0;
dx = (xFin-xIni)/double(npts-1);
xPoly = (xIni:dx:xFin);
yPoly=polyval(A,xPoly);%-------- y = A(1)*x.*x + A(2)*x + A(3);
figure(1)%---------------------------------- trace les Points
hold on
title('Points␣pour␣la␣construction␣du␣polynome')
xlabel('Xi');
ylabel('Yi');
axis fill;
axis equal;
axis([0 10 0 4])
plot(xPoint,yPoint,'sr')
hold off
figure(2)%--------------------------------- trace le polynome
hold on
title('Polynome␣calcule␣par␣la␣methode␣de␣Vandermond')
xlabel('x');
ylabel('y=P(x)');
axis fill;
axis equal;
axis([0 10 0 4])
plot(xPoly,yPoly); plot(xPoint,yPoint,'sr')
hold off
```

FIGURE 6.6 – Construction d'un polynôme avec la méthode de Vandermond

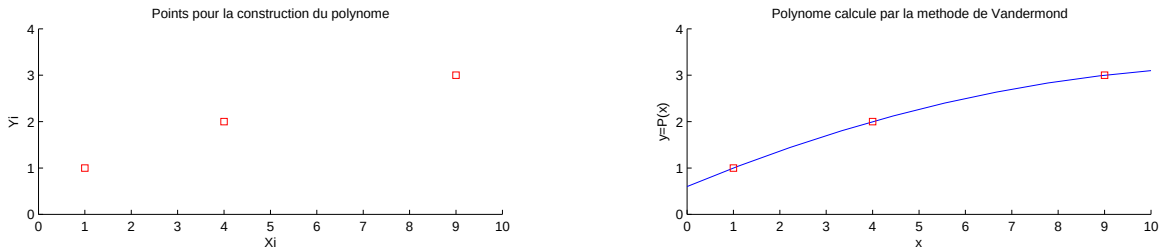


FIGURE 6.7 – Polynôme construit avec la méthode de Vandermond

On généralise avec les coordonnées (x_i, y_i) de chacun de ces $n+1$ points. On peut écrire une équation de la forme,

$$\begin{aligned} P_n(x) &= A_1 x^n + A_2 x^{n-1} + \ldots + A_2 x^2 + A_n x^1 + A_{n+1} \\ &= \sum_{i=n}^{i=0} A_{n-i+1} x^i \end{aligned}$$

qui donne le système de $(n+1)$ équations suivant,

$$\begin{aligned} y_0 &= A_1 x_0^n + A_2 x_0^{n-1} + \ldots + A_2 x_0^2 + A_n x_0^1 + A_{n+1} \\ y_1 &= A_1 x_1^n + A_2 x_1^{n-1} + \ldots + A_2 x_1^2 + A_n x_1^1 + A_{n+1} \\ y_2 &= A_1 x_2^n + A_2 x_2^{n-1} + \ldots + A_2 x_2^2 + A_n x_2^1 + A_{n+1} \\ \ldots &= \ldots \\ \ldots &= \ldots \\ y_n &= A_1 x_n^n + A_2 x_n^{n-1} + \ldots + A_2 x_n^2 + A_n x_n^1 + A_{n+1} \end{aligned}$$

On assemble sous la forme matricielle de Vandermond et s'écrit

$$\begin{bmatrix} x_0^n & x_0^{n-1} & \ldots & x_0^2 & x_0^1 & 1 \\ x_1^n & x_1^{n-1} & \ldots & x_1^2 & x_1^1 & 1 \\ x_2^n & x_2^{n-1} & \ldots & x_2^2 & x_2^1 & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \\ x_n^n & x_n^{n-1} & \ldots & x_n^2 & x_n^1 & 1 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

ou bien,

$$X * A \;=\; Y$$

où X est la matrice formée par les coordonnées x_i et le vecteur Y, par les coordonnées y_i. Les coefficients A_i du polynôme $P_n(x)$ sont obtenus par la résolution de ce système en utilisant une décomposition de Gauss, qui sous Matlab se fait par l'opérateur "\" (Voir Section 5.8).

$$A \;= X\backslash Y$$

Alors, le polynôme peut être calculé à l'aide de la fonction **`polyval`** ou la méthode de Horner décrites ci-dessus.

La construction d'un polynôme peut également se faire en utilisant d'autres informations telles que la pente ou diverses dérivées. L'exemple suivant (Fig. 6.9) montre comment construire un polynôme $p_2(x)$ de degré deux, avec deux points de collocation et une pente,

$$\begin{aligned} p_2(0) &= 1 \\ p_2(3) &= 7 \\ p_2'(1) &= 1. \end{aligned}$$

En substituant dans l'équation générale d'un polynôme de degré 2, donnée par $p_2(x) = A_1x^2 + A_2x^1 + A_3$, on obtient, à partir de ces trois conditions, le système d'équations suivant,

$$\begin{aligned} 1 &= A_3 \\ 7 &= A_1 \cdot 9 + A_2 \cdot 3 + A_3 \\ 1 &= 2A_1 \cdot 1 + A_2 \cdot 1 \end{aligned}$$

Ce qui donne sous forme matricielle, $Xa = Y$, le système de Vandermond,

$$\begin{bmatrix} 0 & 0 & 1 \\ 9 & 3 & 1 \\ 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 7 \\ 1 \end{bmatrix}$$

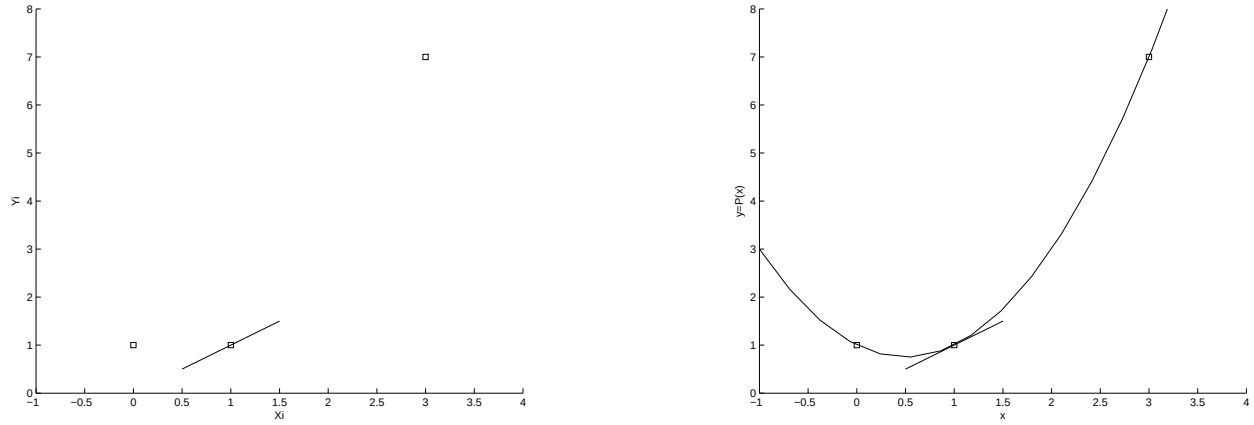
La solution du système, $XA = Y$, donne les coefficients $A_i = X\backslash Y$, à partir desquels on évalue le polynôme,

$$\begin{aligned} p_2(x) &= A_1x^2 + A_2x^1 + A_3 \\ &= x^2 - x + 1 \end{aligned}$$

Le détail de cette démarche est donné au programme de la Fig. 6.8.

```
%-----------------------------------------------------------------
% Construction du polynome P(x) par la methode de Vandermond
% a partir de 2 points (xi,yi) et la derivee (xi',yi').
% Les coefficients Ai sont obtenus par la resolution du systeme
%        A= X\Y
% Le polynome est evalue par la fonction polyval ou Horner
%-----------------------------------------------------------------
clear all; clc; close all
%-----------------------------------------------------------------
xPoint = [0 3 1];yPoint = [1 7 1]';
M = [ 0 0 1;%------------------------ construction de la matrice
      9 3 1;
      2 1 0;];
A= M\yPoint;%---------------- calcul des coefficients du polynome
npts = uint16(30);%------------------ constructuction du polynome
xini = -1.0;xfin = 8.0;
dx = (xfin-xini)/double(npts-1);
xPoly = (xini:dx:xfin);
yPoly=polyval(A,xPoly);
%-------- yPoly = A(3) + A(2)*x + A(1)*x.*x;
figure(1)%------- trace les points de collocation et de la pente
hold on
%title('Conditions pour la construction du polynome')
xlabel('Xi'); ylabel('Yi');
axis([-1 4 0 8])
plot(xPoint(1:3),yPoint(1:3),'sr')
xp = [.5 1.5];yp = [.5 1.5];
plot(xp,yp,'-r')
hold off
figure(2)%----------------------------------- trace le polynome
hold on
%title('Polynome calcule par la fonction polyval ou Horner')
xlabel('x'); ylabel('y=P(x)');
axis([-1 4 0 8])
plot(xPoly,yPoly);
plot(xPoint(1:3),yPoint(1:3),'sr')
xp = [.5 1.5];yp = [.5 1.5];
plot(xp,yp,'-r')
hold off
```

FIGURE 6.8 – Construction d'un polynôme avec des conditions de collocation et de pente



(a) Conditions (b) Polynôme

FIGURE 6.9 – Polynôme construit avec des conditions de collocation et de pente, 6.8

6.3 Représentation implicite : Cercle et ellipse

On illustre la représentation implicite avec l'exemple d'un cercle :

$$x^2 + y^2 = 1$$

On voit que cette représentation ne permet pas le calcul des coordonnées des points de la courbe.

On doit nécessairement transformer la relation $f(x) = 0$ en une équation explicite où les ordonnées, y, sont exprimées en fonction des abscisses, x,

$$y = \pm\sqrt{1 - x^2}$$

Ce qui donne soit, deux valeurs réelles de y , soit deux valeurs complexes.

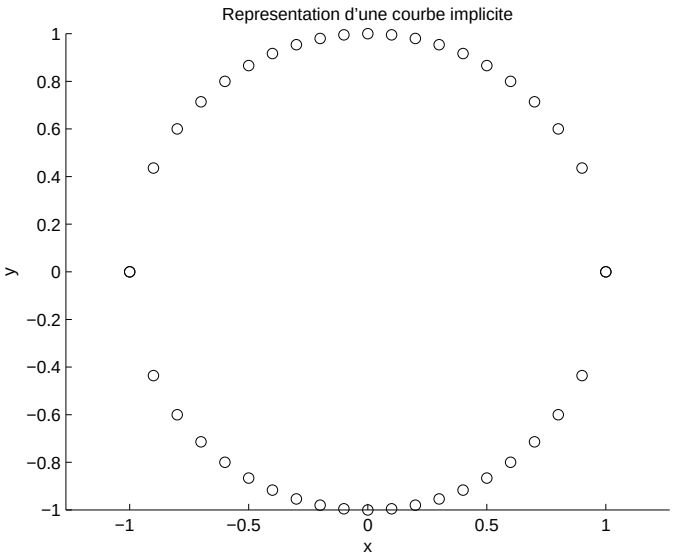


FIGURE 6.10 – Représentation implicite d'un cercle

Le programme de la Fig. 6.11 construit le tracé de cette courbe (Fig. 6.10) en parcourant les valeurs de x sur un intervalle donné (Dans cet exemple $-1.5 \leq x \leq 1.5$) en calculant les valeurs de y correspondantes ; si elles sont réelles, un point est affiché, sinon on incrémente la valeur de x, et on répète le calcul.

```
%-------------------------------------------------------------------
% Trace d'un cercle a partir d'une representation implicite :
%                  x^2 + y^2 = 1
%  que l'on transforme en exprimant  y fonction de x:
%              y = +/- sqrt(1 -x^2)
%-------------------------------------------------------------------
clc; clear all; close all
figure(1)%------------------ les points sont calcules et affiches
hold on
axis equal
title('Representation d''une courbe implicite')
xlabel('x')
ylabel('y')
for x=-1.5:.1:1.5
    discr = 1 - x^2;
    if discr >= 0
        y = sqrt(discr);
        yH = y;    yB = -y;
        plot(x,yH,'ok',x,yB,'ok')
    end
end
hold off
```

FIGURE 6.11 – Programme pour créer et dessiner un cercle à partir d'une représentation implicite

On généralise pour une section conique :

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$$

où a et b représentent les axes majeur et mineur, respectivement, d'une ellipse.

On peut construire le tracé de cette courbe en explicitant les valeurs des ordonnées, y, en fonction des abcisses x :

$$y = \pm b\sqrt{1 - \left(\frac{x}{a}\right)^2}$$

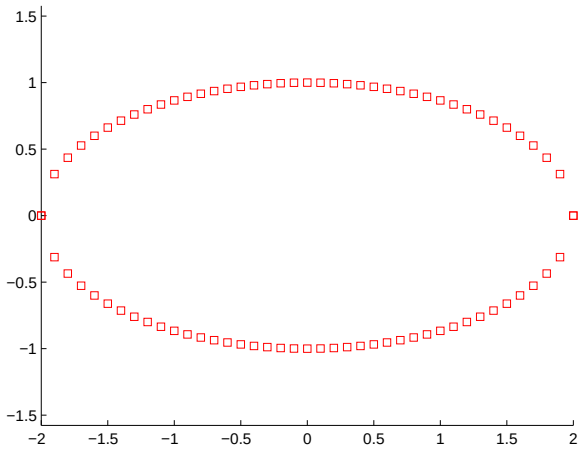


FIGURE 6.12 – Représentation implicite d'une ellipse

En apportant ces modifications au programme de la Fig. 6.11, on obtient le résultat illustré a la Fig. 6.12.

On note que dans le programme de la Fig. 6.11, on a utilisé la commande **`axis equal`**, qui force la même échelle sur les deux axes. Ceci est utile lorsque le rendu ne doit pas déformer le dessin, Sinon, le cercle apparaîtrait comme une ellipse.

La représentation implicite présente plusieurs difficultés comme le montre le mode de calcul. Il faut connaître assez précisement la localisation de la courbe et faire des tests sur les valeurs calculées, ce qui rend difficile la généralisation à des courbes quelconques.

6.4 Formes paramétriques des courbes

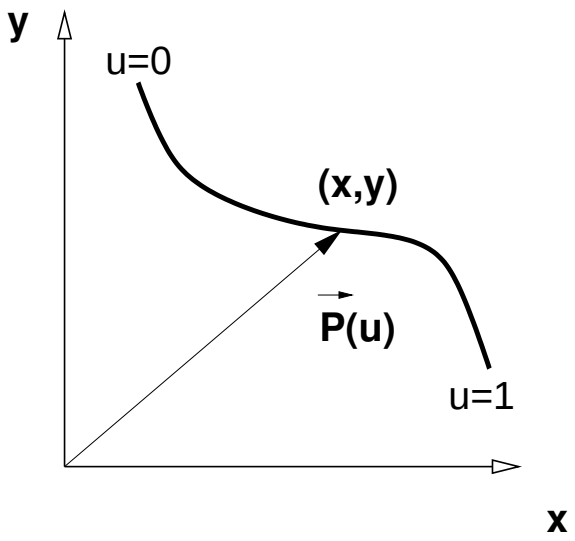
Dans la représentation paramétrique, chaque coordonnée est représentée par une équation explicite, fonction d'un paramètre :

$$\vec{P} = \begin{bmatrix} x = f(u) \\ y = g(u) \\ z = h(u) \end{bmatrix}$$

Le paramètre u varie sur l'intervalle de définition que l'on prend, sans perte de généralité :

$$0 \leq u \leq 1$$

En parcourant la plage du paramètre, on associe trois coordonnées, x, y et z à chaque valeur de u ; ce qui engendre l'ensemble des points de la courbe.

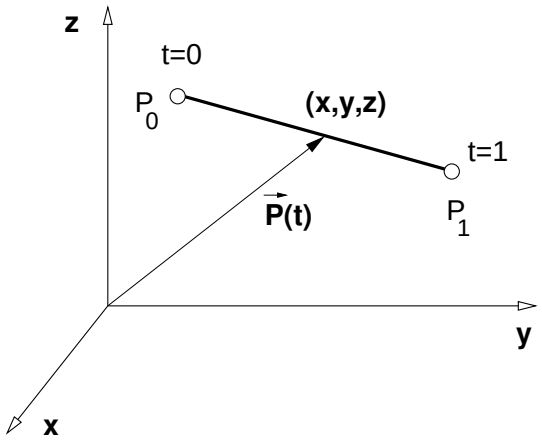


6.4.1 La droite

La représentation paramétrique d'une droite est donnée par :

$$\begin{aligned} & \vec{P} = \vec{P_0} + t(\vec{P_1} - \vec{P_0}) \\ \text{avec} \quad & x = a + lt \\ & y = b + mt \\ & z = c + nt \end{aligned}$$

où $\vec{P_0} = (a, b, c)$ indique le début de la droite, correspondant au paramètre $t = 0$, et $\vec{P_1} = (a + l, b + m, c + n)$ indique l'autre extrémité, correspondant à $t = 1$.



Aux Figs. 6.13, 6.14 et 6.15 on montre trois méthodes pour la représentation d'une droite paramètrique, pour illustrer les différences dans l'utilisation des structures de données vectorielle et matricielle, respectivement.

```
%-------------------------------------------------------------
% Programme pour creer un segment de droite entre deux points
% P0 et P1, en utilisant une representation parametrique:
%               P = P0 + t*(P1-P0)
%-------------------------------------------------------------
clear all; close all; clc
P0=[1.0  6.0]; P1=[7.0 -1.0]; %------- On declare les deux points
npt= uint8(11);%-------- Nombre total de points incluant P0 et P1
dt = .1;%------------------------------ Intervalle du parametre t
%------------ Construction du segment - 1ere methode -----------
t  = 0.0;
P(1,1)=P0(1);  P(1,2)=P0(2);%---------------------- point initial
for i=uint8(2):npt-1
   t = t+dt;
   P(i,1)= P0(1)+t*(P1(1)-P0(1));
   P(i,2)= P0(2)+t*(P1(2)-P0(2));
end
P(npt,1)=P1(1);  P(npt,2)=P1(2);%-------------------- point final
```

FIGURE 6.13 – Programme(1) pour créer et afficher une droite paramétrique

```
%-------------------------------------------------------------
P0=[-2.0 -6.0]; P1=[7.0 8.0];%------ On declare les deux points
npt= uint8(5);%------- Nombre total de points incluant P0 et P1
dt = 1./double(npt-1);%-------------- Intervalle du parametre t
%------------ Construction du segment - 2eme methode ---------
t  = 0.0;
P(1,1:2)   =  P0; %------------------------ point initial: t=0
for i=uint8(2):npt-1
   t = t+dt;
   P(i,1:2) =  P0 + t*(P1-P0);
end
P(npt,1:2) =  P1;%---------------------------- point final: t=1
```

FIGURE 6.14 – Programme(2) pour créer et afficher une droite paramétrique

```
%-----------------------------------------------------------------
P0=[-3.0  9.0];%-------------------- On declare les deux points
P1=[ 4.0 -2.0];
npt= uint8(6);%-------- Nombre total de points incluant P0 et P1
tmin = -.5;%------------------------- Intervalle du parametre t
tmax = 1.5; dt = (tmax-tmin)/(double(npt-1));
t    = [tmin:dt:tmax]';%----------------------- vecteur colonne
%------------ Construction du segment - 3eme methode ---------
Segment = [P0(1) + t*(P1(1)-P0(1)), P0(2) + t*(P1(2)-P0(2))];
figure (1)%------------------------------- Affichage du segment
hold on; grid on
axis ([-4 8  -8 10])
plot(P0(1),P0(2),'ro',P1(1),P1(2),'ro')
plot(P0(1,1),P0(1,2),'ro',P1(1,1),P1(1,2),'ro')
plot(Segment(:,1),Segment(:,2),'g-')
hold off
```

FIGURE 6.15 – Programme(3) pour créer et afficher une droite paramétrique

Ces trois programmes et leur représentations respectives donnent les résultats montrés à la Fig. 6.16.

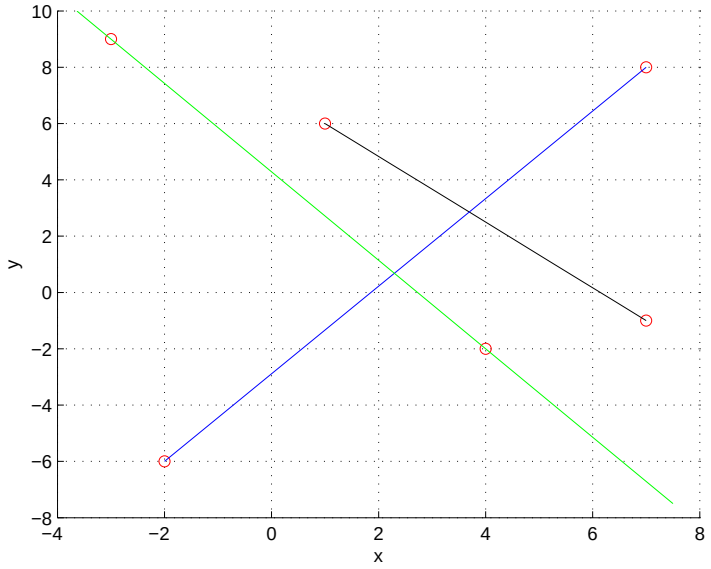


FIGURE 6.16 – Diverses représentations paramètriques d'une droite

6.4.2 Les coniques

La représentation paramétrique d'une ellipse est donnée par les relations,

$$\begin{aligned} x &= a\cos u \\ y &= b\sin u \end{aligned}$$

où a et b sont les axes majeur et mineur, respectivement.

Lorsque $a = b$, on retrouve l'équation du cercle. Également, à partir des identités trigonométrique, on peut retrouver la représentation explicite de la Section 6.3.

Dans l'exemple suivant, on montre comment utiliser la représentation paramétrique de façon intéractive pour construire une ellipse. Le programme donné à la Fig. 6.18 procède en deux étapes : d'abord les valeurs des axes a et b sont saisies interactivement à l'aide de la fonction **`inputdlg`** qui affiche une grille de saisie telle que montrée à la Fig. 6.17. Les données entrées dans les boîtes sont des chaînes de caractères qui doivent être converties en double par la fonction **`str2double`**.

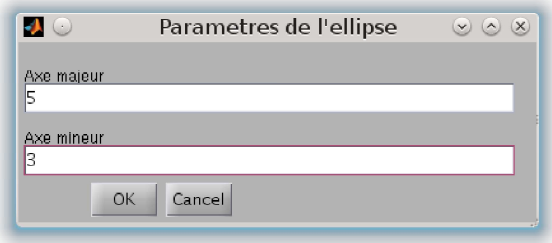


FIGURE 6.17 – Saisie interactive des paramètres pour la construction d'une ellipse

```
% Programme pour creer une ellipse avec la representation
% parametrique:     x = a*cos(t) et    y = b*sin(t)
% a: axe majeur, b: axe mineur
%-------------------------------------------------------------------
clc; clear all; close all
%--------------------------- Saisie des  parametres de l'ellipse
prompt={'Axe majeur','Axe mineur'};
name='Parametres de l''ellipse';
numlines=1;
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';
defaultanswer={'0','0'};
N=inputdlg(prompt,name,numlines,defaultanswer,options);
a=str2double(N{1});
b=str2double(N{2});%------------------------------------ Axes
```

FIGURE 6.18 – Représentation paramétrique d'une conique : saisie interactive

Le paramètre u est arbitraire, et on choisira l'angle $0 \leq t \leq 2\pi$. Ce qui donne le résultat de la Fig. 6.19.

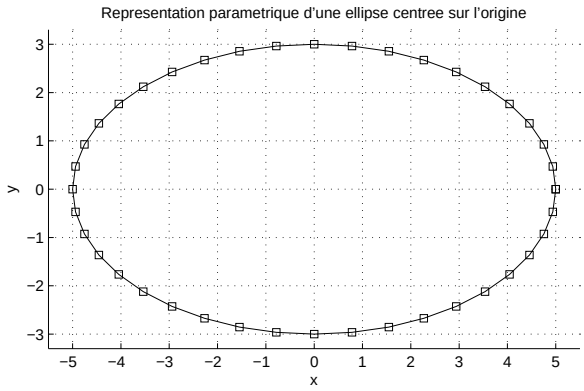


FIGURE 6.19 – Construction d'une ellipse avec représentation paramétrique

```
%------------------   SUITE DU PROGRAMME  -----------------------
tmin  = 0.; tmax  = 2*pi; %---------- discretisation de la courbe
npt = uint8(41);   dt = (tmax-tmin)/double(npt-1);
%----------------------------- Calcul de l'ellipse: 1ere methode
for i=uint8(1):npt
    t=tmin+double(i-1)*dt;%--------- t calcule au fur et a mesure
    P(i,1:2) = [a*cos(t) b*sin(t)];
end
figure (1)
hold on;
title('Representation parametrique d''une ellipse
centree sur l''origine')
grid on; axis equal; axis([-1.1*a 1.1*a -1.1*b 1.1*b])
xlabel('x'); ylabel('y')
plot(P(1:npt,1),P(1:npt,2),'ob-')
hold off;
%----------------------------- Calcul de l'ellipse: 2eme methode
tt=(tmin:dt:tmax)';%------------------- t est calcule et stocke
P(1:npt,1:2) = [a*cos(tt) b*sin(tt)];
figure (2)
......
......
```

FIGURE 6.20 – Suite du programme Fig. 6.18 : Calcul de l'ellipse

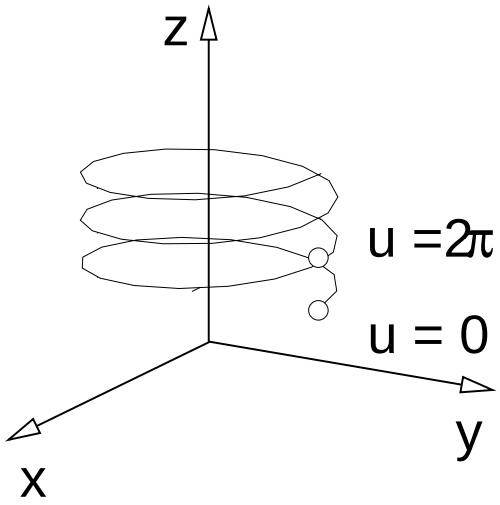
6.4.3 L'hélice et vecteur tangent

La représentation paramétrique d'une spirale circulaire de rayon r est donnée par les relations suivantes :

$$\begin{aligned} x &= r\cos u \\ y &= r\sin u \\ z &= bu \end{aligned}$$

où le paramètre u peut s'interpréter comme l'angle du vecteur position avec l'axe des x et donne un pas égal à b pour :

$$0 \leq u \leq 2\pi$$



Le programme à la Fig. 6.21 montre la construction de cette courbe avec la représentation paramètrique où le paramètre u est égal à l'angle t.

```
%-------------------------------------------------------------------
%  Programme pour creer et tracer une helice de rayon r
%  et un pas de s, avec les equations parametriques:
%   x = r*cos(t),   y = r*sin(t),    z = s*t
%-------------------------------------------------------------------
clear all; close all; clc
%-------------------------------------------------------------------
r = 2.0;s = 0.54;%-------------------- Les parametres de l'helice
tmin  = pi/4;%-------------------- La discretisation de la courbe
tmax  = 5*pi; npt = uint8(50); dt = (tmax-tmin)/double(npt-1);
for i=uint8(1):npt%----------- Calcul des coordonnees de l'helice
t        = tmin + double(i-1)*dt;
P(i,1:3) = [r*cos(t) r*sin(t)  s*t];
end
figure (1)%-------------------------------------------------------
hold on;
plot3(P(1:npt,1),P(1:npt,2), P(1:npt,3),'o-')
xlabel('X'); ylabel('Y'); zlabel('Z')
hold off;
```

FIGURE 6.21 – Programme pour créer et dessiner une hélice

Dans cet exemple, le paramétre t varie entre $\pi/4 \leq t \leq 5\pi$, qui fait en sorte que la courbe fait plusieurs tours, se déplaçant de s, la valeur du pas, dans la direction z.

On note que le dessin se fait avec la fonction **`plot3`** qui est la version 3d de **`plot`**, et qui peut être tournée dans l'espace. Le tracé avec 50 points est illustré à la Fig. 6.22.

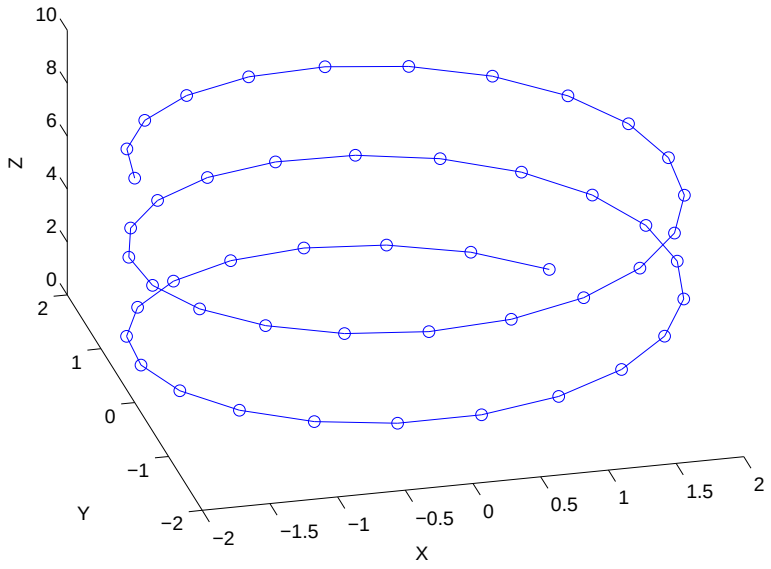


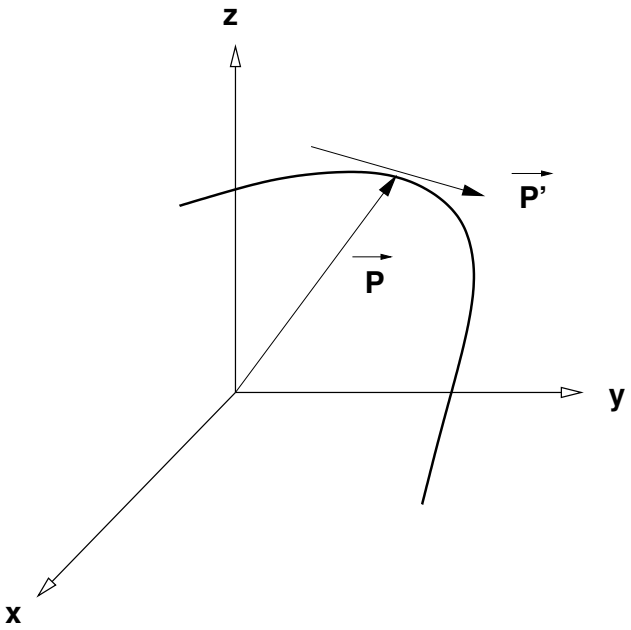
FIGURE 6.22 – Tracé de l'hélice obtenu avec le programme de la Fig. 6.21

En dérivant les équations paramétriques de l'hélice par rapport au paramètre t, on obtient le vecteur tangent à la courbe :

$$\begin{aligned} \vec{P}\,'(t) &= \frac{d\vec{P}(t)}{dt} \\ &= (\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}) \end{aligned}$$

Dans le cas de l'hélice, pour les trois composantes du vecteur, on obtient

$$\begin{aligned} x' &= -r\sin t \\ y' &= r\cos t \\ z' &= b \end{aligned}$$



On peut représenter graphiquement ce vecteur par un segment de droite allant du pied du vecteur, la position du point (x, y, z), et la pointe du vecteur (x', y', x'). Cet ajout au calcul est montré à la Fig. 6.23, et le résultat à la Fig. 6.24.

```
%--------------- suite du programme --------------------------
%------ Calcul du vecteur tangent au point 25 obtenu en derivant
%------------- le vecteur position P par rapport au parametre t:
%-------------------------------- VT= [-r*sint(t)  r*cos(t)  s]
point = uint8(25);
t    = tmin + double(point-1)*dt;
VT(1,:) = P(point,:);%------------------ pied du vecteur tangent
VT(2,:) = [-r*sin(t) r*cos(t) s] + VT(1,:);%-- pointe du vecteur
%-------------------------------------------------------------
figure (2)
hold on;
plot3(P(1:npt,1),P(1:npt,2), P(1:npt,3))
plot3(VT(:,1),VT(:,2),VT(:,3),'-r')
xlabel('X'); ylabel('Y'); zlabel('Z')
hold off;
```

FIGURE 6.23 – Calcul et dessin du vecteur tangent

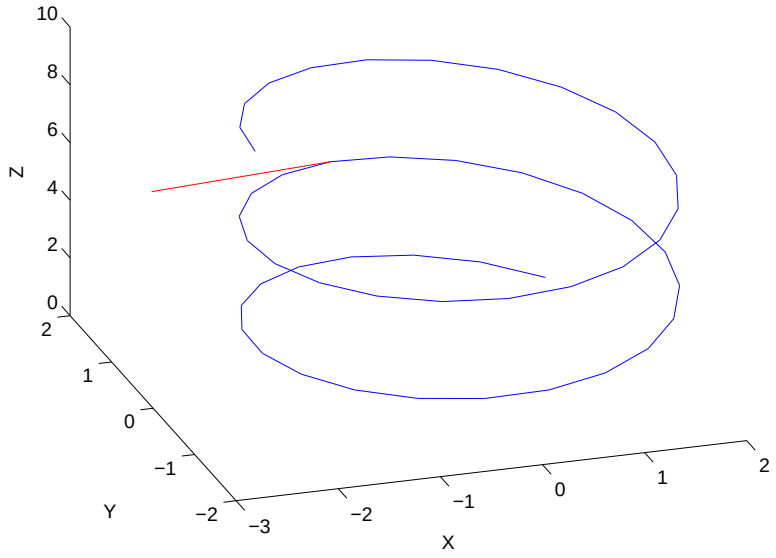


FIGURE 6.24 – Tracé de l'hélice et le vecteur tangent obtenus avec le programme de la Fig. 6.23

6.5 Intégration numérique

6.5.1 Calcul de l'aire sous une courbe

Soit une courbe décrite par la fonction :

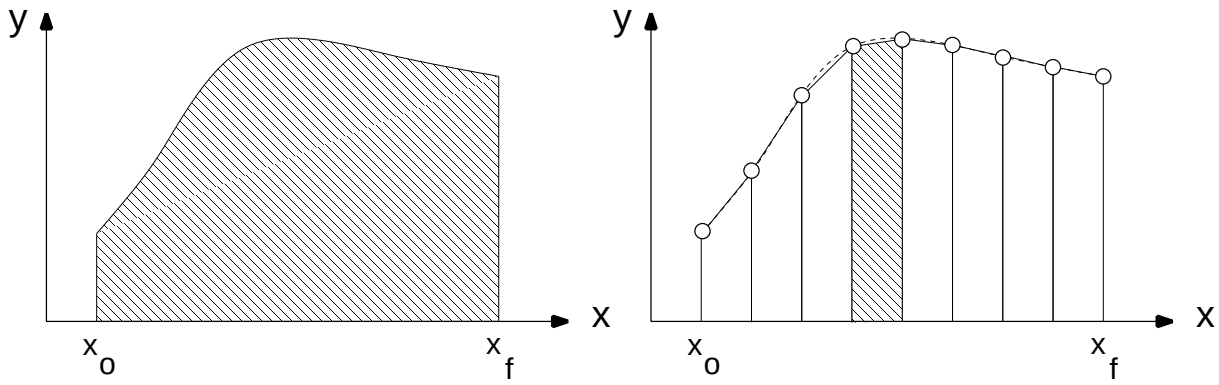
$$y = f(x)$$

On calcule l'aire sous la courbe par l'intégrale :

$$A = \int_{x_0}^{x_f} f(x)dx$$

Une méthodologie pour approximer ce calcul consiste à discrétiser cette courbe en n points, x_i espacés de $\Delta x = x_i - x_{i-1}$. On obtient $(n-1)$ quadrilatères d'aire

$$dA_i = \frac{1}{2}(y_i + y_{i+1})\Delta x$$



(a) Intégration continue (b) Intégration discrète

FIGURE 6.25 – Intégration numérique

La sommation de ces dA_i donne une approximation à l'aire sous la courbe :

$$A = \int_{x_0}^{x_f} f(x)dx \approx \sum_{i=1}^{i=n-1} dA_i$$

On applique au calcul de l'intégrale de la fonction $sin(x)$ entre les limites $x_0 = 0$ et $x_f = \pi$, illustré à la Fig. 6.26

$$A = \int_{x_0}^{x_f} sin(x)dx$$

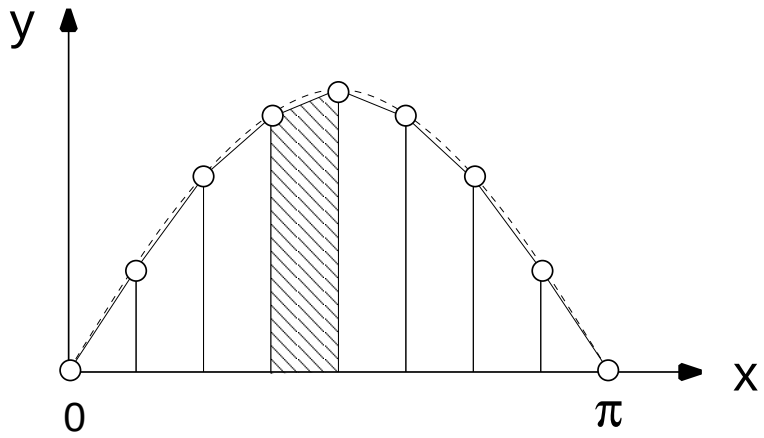


FIGURE 6.26 – Approximation de l'intégrale de $sin(x)$

La programmation comprend deux scripts : un programme principal (Fig. 6.27) où les paramètres du calcul sont initialisés, et une sous-routine ou fonction (Fig. 6.28) qui évalue la sommation des dA_i.

```
%-------------------------------------------------------------------------
% Programme principal pour evaluer l'integrale de f(x)=sin(x)
% avec nbPoints points  de x=limite1 a x=limite2
%-------------------------------------------------------------------------
clear all;clc;
nbPoints= uint16(10); limite1 = 0.; limite2 = pi;
dx = (limite2-limite1)/double(nbPoints-1);
x  = limite1:dx:limite2; y  = sin(x);
integrale    = calculeINT(nbPoints,x,y);
valeurExacte = 2.0;
erreur  = abs(valeurExacte - integrale);
%-------------------------------------------------------------------------
resultat=sprintf('Integration␣numerique␣de␣sin(x)␣sur␣l''intervalle␣\n
x0=␣%10.5f␣,␣x1=␣%10.5f␣avec␣%3d␣points,\n
donne␣%10.5f␣et␣une␣erreur␣de␣%10.5f',
limite1,limite2,nbPoints,integrale,erreur);
disp(resultat)
```

FIGURE 6.27 – Programme pour le calcul de l'intégrale de $f(x) = sin(x)$

La fonction Matlab **trapz(x,y)** réalise le même calcul que la fonction donnée à la-Fig. 6.28.

```
function integrale = calculeINT(nb,x,y)
%-------------------------------------------------------------------------
% Integrale d'une fonction y= f(x) donnee avec nb points (xi,yi)
%-------------------------------------------------------------------------
integrale=0;
for i=2:nb
    integrale = integrale + .5*(x(i)-x(i-1))*(y(i)+y(i-1));
end
```

FIGURE 6.28 – Fonction pour l'évaluation de la sommation des dA_i

Les fonctions **sprintf** et **disp** (Voir Section 4.2) sont utilisées dans le programme de la Fig. 6.27 pour imprimer le résultat sous la forme suivante, .

```
L'integration numerique de sin(x) sur l'intervalle
 x0=    0.00000 , x1=    3.14159 avec  10 points,
 donne    1.97965 et une erreur de    0.02035
```

Pour évaluer l'intégrale d'autres fonctions, $f(x)$, il suffit de remplacer l'énoncé **y = sin(x);** dans le programme de la Fig. 6.27 par l'expression appropriée.

6.5.2 Calcul de la longueur d'une courbe

6.6 Transformations géométriques

La géométrie d'un objet est constituée par un ensemble de points, où chaque point est représenté par ses coordonnées. Informatiquement, pour la représentation d'un point, on utilise une variable de type vecteur 1X2 ou 1X3, en 2d ou 3d, respectivement. Un objet complexe comprend une suite de nbPTS points dont les coordonnées sont representées, informatiquement, par une matrice nbPTSX2 ou nbPTS X 3, en 2d ou 3d, respectivement. Par convention, le premier indice représente le numéro du point, tandis que le second, représente la coordonnée, 1 pour x, 2 pour y etc .

Dans l'exemple suivant, on construit un triangle en spécifiant les coordonnées de ses trois sommets dans un tableau de dimension 3X2,

```
triangle = [ 1  1;
             8  6;
             2  8];
```

Le script à la Fig. 6.29 illustre la construction du triangle.

```
%-------------------------------------------------------------------
% Construction de l'objet
%-------------------------------------------------------------------
clc; clear all; close all
npts = 3;
triangle = [ 1  1;
             8  6;
             2  8];
figure(1)%---------------------- Triangle defini par les sommets
hold on
grid on
axis equal
axis([0 10 0 10])
title('Sommets d''un triangle')
xlabel('x')
ylabel('y')
plot(triangle(1:3,1),triangle(1:3,2),'s')
```

FIGURE 6.29 – Programme pour la construction d'un triangle par ses sommets

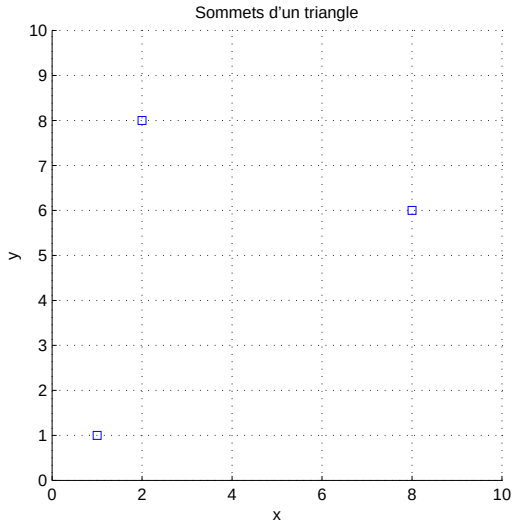


FIGURE 6.30 – triangle et affichage de ses trois sommets

Le résultat est illustré à la Fig. 6.30. Souvent, pour un meilleur rendu sur le plan graphique, on trace le contour ou la frontière de l'objet qui doit donner une ligne continue représentant le bord. Alors pour fermer cette ligne, on doit répéter le premier point, qui se superpose avec le dernier. La comparaison des scripts et des tracés aux Figs. 6.29-6.30 et Figs. 6.31-6.32 illustre cette différence.

```
%-------------------------------------------------------------------
% Construction de l'objet
%-------------------------------------------------------------------
clc; clear all; close all
npts = 4;
triangle = [ 1  1;
             8  6;
             2  8;
             1  1];%le dernier point se superpose avec le premier
figure(2)%------------------------ Triangle defini par les cotes
hold on
grid on
axis equal
axis([0 10 0 10])
title('Cotes d''un triangle')
xlabel('x')
ylabel('y')
plot(triangle(1:4,1),triangle(1:4,2))
```

FIGURE 6.31 – Programme pour la construction d'un triangle par ses cotés

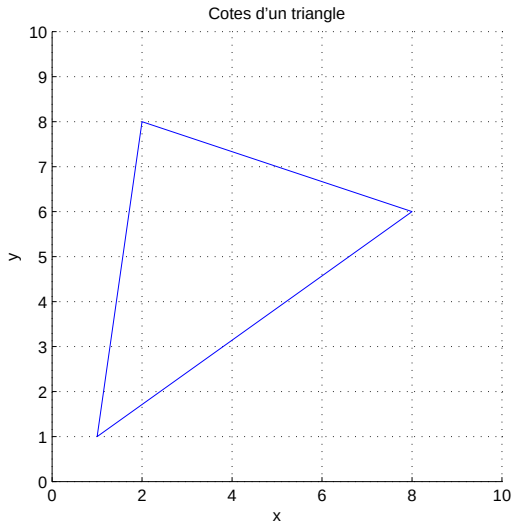


FIGURE 6.32 – triangle et affichage de ses trois cotés

```
figure(3)%------------------------ Triangle defini par la face
hold on
grid on
axis equal
axis([0 10 0 10])
title('Face d''un triangle')
xlabel('x')
ylabel('y')
fill(triangle(1:3,1),triangle(1:3,2),'r')
```

FIGURE 6.33 – Représentation d'un triangle par sa face

Finalement, une dernière représentation est la "face" où l'intérieur du polygone est rempli avec la fonction **`fill`** comme illustré à la Fig. 6.33

6.6.1 Translation

Il existe plusieurs opérations qui transforment une géométrie. Nous en étudirons deux parmi les plus courantes, qui ne déforment pas la géométrie.

La translation consiste á déplacer un objet, c'est-à-dire l'ensemble des points, de façon solidaire, en bloc. Comme un point est représenté par un vecteur, alors, le déplacement d'un point $P(x, y)$ d'une distance $d(a, b)$ s'interprête comme une addition vectorielle.

$$\vec{P}'(x', y') = \vec{P}(x, y) + \vec{d}(a, b)$$

où on ajoute à chaque composante x et y de P, la composante respective, a et b, du vecteur déplacement. Cette opération est illustrée à la Fig. 6.35 et le programme est donné à la Fig. 6.34.

$$\begin{aligned} x' &= x + a \\ y' &= y + b \end{aligned}$$

```
%-------------------------------------------------------------------
% Translation d'un objet
%-------------------------------------------------------------------
clc; clear all;close all
%-------------------------------------------------------------------
npts = 4;%------------------------------ Construction de l'objet
triangle = [ 1  1;
             8  5;
             2  6;
            1  1];% le dernier point se superpose avec le premier
d = [1 2];%---------------------------------------- deplacement
figure(2)
hold on
axis equal; axis([0 10 0 10])
title('Deplacement  d''un triangle'); xlabel('x'); ylabel('y')
plot(triangle(1:4,1),triangle(1:4,2),'sk')%--- position initiale
plot(triangle(1:4,1),triangle(1:4,2),':k')
for i=1:2%------------------------------ Translation de l'objet
    triangle(:,i) =triangle(:,i) +d(i);
end
plot(triangle(1:4,1),triangle(1:4,2),'sr')%----- position finale
plot(triangle(1:4,1),triangle(1:4,2),'-k')
```

FIGURE 6.34 – Programme pour le déplacement par translation d'un triangle.

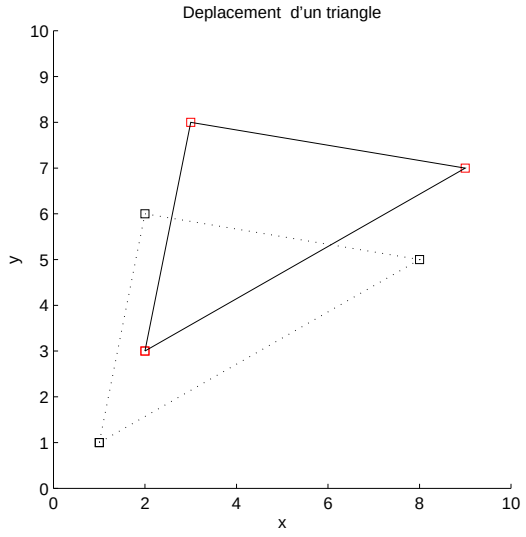


FIGURE 6.35 – translation d'un triangle et tracé des ses trois cotés par des segments

6.6.2 Rotation

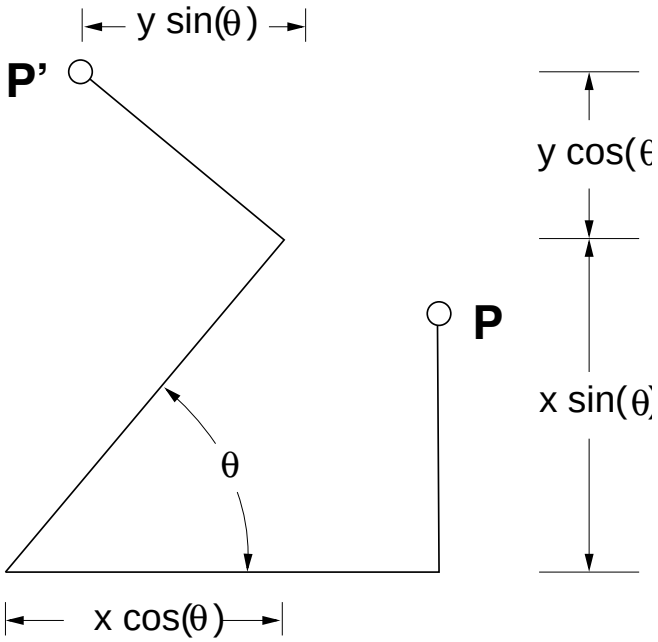
On tourne un point P autour de l'origine, tel qu'illustré à la figure ci-contre.

Les coordonnées du nouveau point P' sont obtenues par :

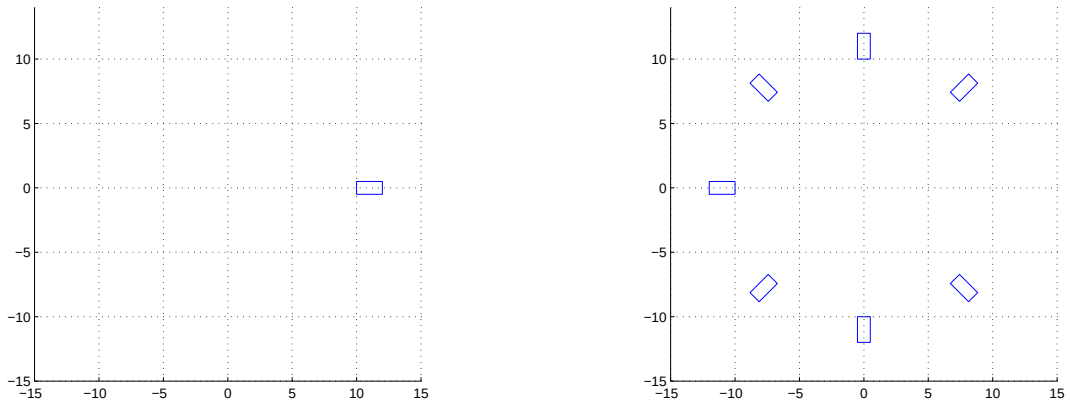
$$\begin{aligned} x' &= xcos(\theta) - ysin(\theta) \\ y' &= xsin(\theta) + ycos(\theta) \end{aligned}$$

qui s'écrit sous forme matricielle suivante,

$$[x' \;\; y'] = [x \;\; y] \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}$$



On illustre avec l'exemple de la Fig. 6.36, où un rectangle est tourné successivement d'un angle θ autour de l'origine, par l'application de la matrice de rotation sur la matrice des coordonnées du rectangle.



(a) Position de départ

(b) Positions successives

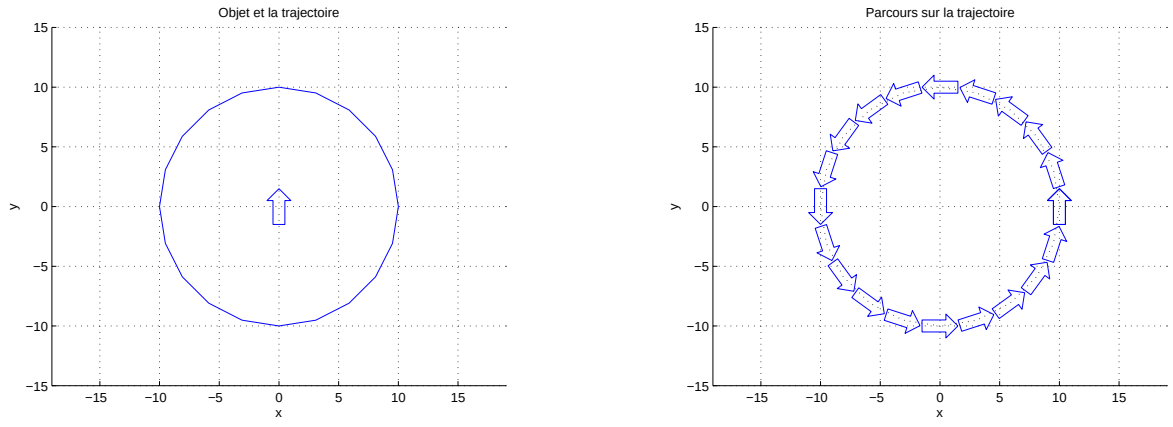
FIGURE 6.36 – Rotation d'un rectangle autour de l'origine

Les calculs sont détaillés dans le programme à la Fig. 6.37.

```
% Rotation d'un objet autour de l'origine
%-------------------------------------------------------------------
clc; clear all; close all
npts = 5;%------------------------------ Construction de l'objet
obj = [ 10 -.5; 12 -.5; 12  .5; 10  .5; 10 -.5];
%------------------ le dernier point se superpose avec le premier
figure(1)
hold on; grid on; axis equal; axis([-15 15 -15 14])
plot(obj(:,1),obj(:,2))
angle = pi/4;%------------ Construction de la matrice de rotation
R = [cos(angle) sin(angle);
    -sin(angle) cos(angle)];
figure(2)
hold on; grid on; axis equal; axis([-15 15 -15 14])
for i=1:7%---------------------------------- Rotation de l'object
    obj = obj*R;
    plot(obj(:,1),obj(:,2))
end
```

FIGURE 6.37 – Programme pour tourner un objet autour de l'origine

Une application plus complexe consiste à aligner un objet tangentiellement avec une courbe. Ici, l'objet est une flêche centrée sur l'origine, et la courbe est un cercle centré sur l'origine. Le but est de faire parcourir la flêche sur la courbe de sorte que celle-ci soit tangente à la courbe, comme montré à la Fig. 6.38.



(a) Objet et trajectoire centrés sur l'origine

(b) Vecteur tangent au cercle

FIGURE 6.38 – Flêche suivant une trajectoire

L'aspect le plus important du fonctionnement de la rotation est que cette transformation s'opère autour de l'origine. Pour l'exemple ci-dessus, la flêche étant centrée sur l'origine, il faut d'abord appliquer la rotation á cet endroit pour aligner la flêche sur la trajectoire, et ensuite déplacer vers la position sur le cercle (6.39). L'inverse donnera un résultat différent, car ces deux transformations ne commutent pas !

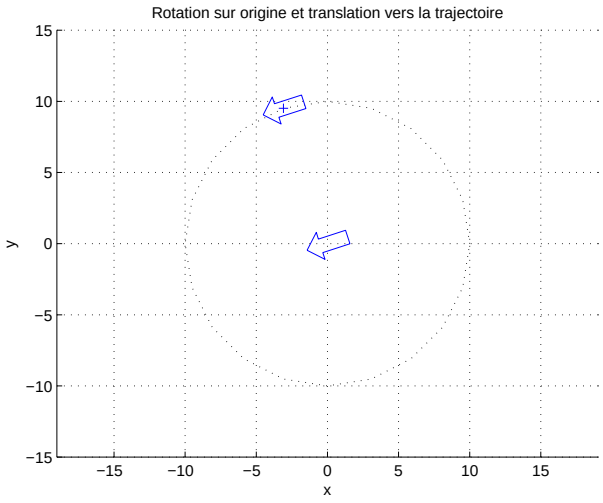


FIGURE 6.39 – Alignement (Rotation) suivie du déplacement vers la position sur la trajectoire

On procède en trois étapes :

Construction de l'objet : Fig. 6.40 ;

```
%-------------------------------------------------------------------
% Deplacement d'un objet le long d'une trajectoire
%-------------------------------------------------------------------
clc; clear all; close all
%------------------------------------ Construction de l'objet
F=[ -0.0    1.5;  -1.0    0.5;  -0.5    0.5;   -0.5   -1.5;
     0.5   -1.5;   0.5    0.5;   1.0    0.5;   -0.0    1.5];
figure (1)%---------------------------------- On dessine l'objet
hold on
grid on; axis([-15 15 -15 15]); axis equal
plot(F(:,1),F(:,2))
hold off
```

FIGURE 6.40 – Déplacer un objet tangentiellement le long d'une courbe :objet

Calcul de la trajectoire : Fig. 6.41

```
%-----------------suite du programme-----------------------------
rayon=10; %------------------------------------ la trajectoire
tINI=0; tFIN=2*pi; nbPTS=uint16(21);
dt= (tFIN-tINI)/double(nbPTS-1); t=tINI:dt:tFIN;
trajet(:,1:2)= [rayon*cos(t') rayon*sin(t')];
figure(2)
hold on; grid on; axis([-15 15 -15 15]); axis equal
plot(trajet(:,1),trajet(:,2))
hold off
```

FIGURE 6.41 – Déplacer un objet tangentiellement le long d'une courbe :trajectoire

Rotation et déplacement : Fig. 6.42

```
%-----------------suite du programme-----------------------------
figure(3)%-------------  On calcule et dessine le vecteur tangent
hold on; grid on
axis([-15 15 -15 15]); axis equal
for iPTS=1:nbPTS
    T=Rotation(F,t(iPTS));%----------------------------- rotation
    T=Translation(T,trajet(iPTS,:));%---------------- translation
    plot(T(:,1),T(:,2))
end
plot(trajet(:,1),trajet(:,2),'+k')
hold off
```

FIGURE 6.42 – Déplacer un objet tangentiellement le long d'une courbe :trajectoire

Une autre considération, d'ordre informatique, est l'encapsulation de ces deux opérations sous forme de fonctions qui prennent en argument la matrice des coordonnées et les paramètres de la transformation (l'angle de rotation ou le vecteur de dplacement), et retourne l'objet transformé sous forme d'une matrice. Cette dernière peut porter le même nom, dans quel cas l'objet original sera écrasé. Sinon, une nouvelle matrice sera créée, et on aura deux objets, représentant l'ancien et le nouveau, respectivement. Le détail du programme se trouve à la Fig. 6.42 où les deux fonctions **`Rotation`** et **`Translation`** réalisent ces transformations.

```
function objet = Rotation(objet,angle)
R = [cos(angle) sin(angle);%----------------- matrice de rotation
    -sin(angle) cos(angle)];
objet = objet*R;%------------------------- Rotation de l'object
```

FIGURE 6.43 – Fonction pour tourner unn objet autour de l'origine

```
function objet = Translation(objet,displ)
for i=1:2
    objet(:,i) =objet(:,i) +displ(i);
end
```

FIGURE 6.44 – Fonction pour déplacer un objet

6.7 Résolution d'équations nonlinéaires

Une équation nonlinéaire

$$y = f(x)$$

est une équation où les coefficients sont fonction de l'inconnue, la variable x. Résoudre ou trouver les racines de f revient à en trouver les zéros,

$$f(x) = 0$$

. Il existe de nombreuses méthodes pour réaliser cette opération. On présente deux algorithmes pour montrer les différences dans l'utilisation des structures de contrôle **`while`**, **`if`** et **`break`**.

On illustre l'utilisation de **`while`** avec la méthode de la bissection qui consiste à itérer sur un intervalle donné qui encadre la racine. C'est-à-dire que l'on connaît deux valeurs, xG et xD, respectivement à gauche et à droite de la racine recherchée. Dans cette situation, une approximation est obtenue par la bissection de cet intervalle,

$$xM = (xG + xD)/2$$

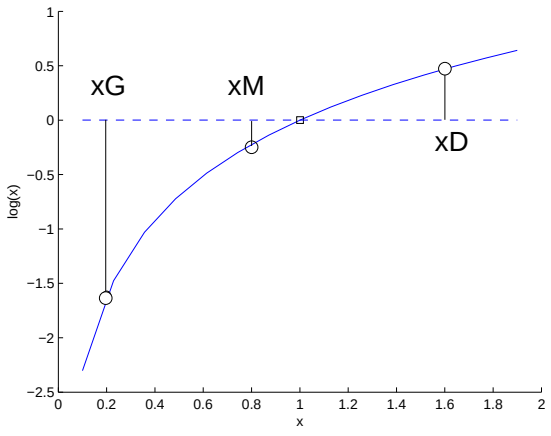


FIGURE 6.45 – La méthode de bissection avec l'énoncé **while**

tel qu'illustré à la Fig. 6.45. Cette nouvelle valeur située à mi-chemin de l'intervalle, se trouve soit à droite ou à gauche de la racine.

Avec un test sur le changement de signe de $f(xM)$ par rapport, soit à $f(xG)$ ou $f(xD)$, on établi les nouvelles valeurs de xG et xD, qui nécessairement encadrent la solution. On répète tant qu'une précision donnée n'est pas atteinte.

On note qu'il faut amorcer ce type de calcul avec un intervalle initial qui encadre au départ le zéro recherché. Donc, il faut une connaissance approximative de la solution. À chaque itération, l'intervalle se raffine, de sorte que la précision est fonction du nombre de fois que la subdivision est appliquée. Le code source pour cet algorithme, appliqué à la fonction $log(x) = 0$, se trouve à la Fig. 6.47.

Ce qui donne le résultat suivant :

Iteration	xG	xD	Erreur
1	0.2000	1.6000	1.4000
2	0.9000	1.6000	0.7000
3	0.9000	1.2500	0.3500
4	0.9000	1.0750	0.1750
5	0.9875	1.0750	0.0875
6	0.9875	1.0312	0.0437
7	0.9875	1.0094	0.0219
8	0.9984	1.0094	0.0109
9	0.9984	1.0039	0.0055
10	0.9984	1.0012	0.0027
11	0.9998	1.0012	0.0014
12	0.9998	1.0005	0.0007

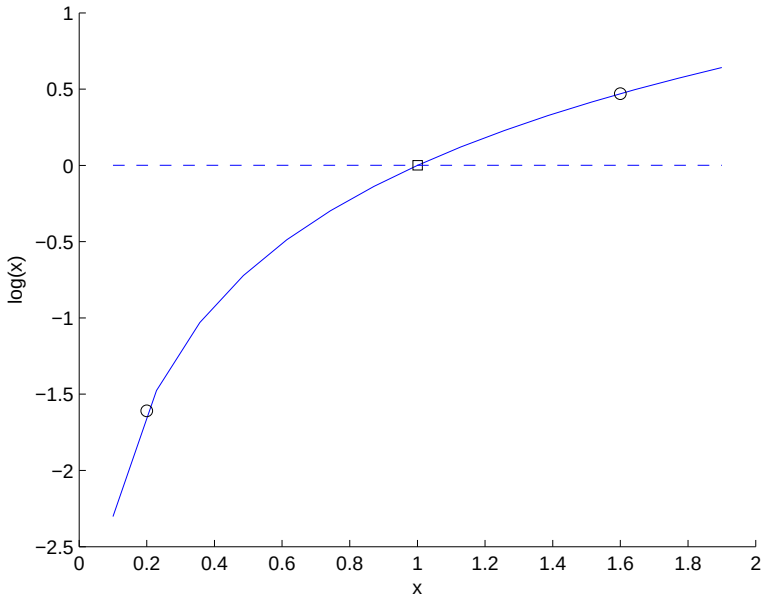


FIGURE 6.46 – Convergence de la méthode de bissection pour le calcul de la racine de $log(x) = 0$.

```
%-------------------------------------------------------------
% On resoud f(x)=0 avec la methode de bissection, qui necessite
% deux valeurs de x, xG et xD, qui encadrent la racine.
%-------------------------------------------------------------
clc; clear all; close all
%-------------------------------------------------------------
xG=.2; xD=1.6;%- Valeur initiale de l'intervalle de la solution
xDebut = xG;
xFinal = xD;
erreur= abs(xD-xG);
iter =1;
precision = .001;%------------------------ precision souhaitee
sousTitre='Iteration␣␣␣␣␣xG␣␣␣␣␣␣␣␣␣␣xD␣␣␣␣␣␣␣␣␣Erreur';
titre    ='␣␣␣␣␣␣␣␣␣␣␣METHODE␣DE␣BISSECTION';
disp(titre)
disp(sousTitre)
resultat=sprintf('%␣5d␣%12.4f␣%12.4f␣%12.4f',iter,xG,xD,erreur);
disp(resultat)
% Tant que la precision n'est pas atteinte, subdivise intervalle
while erreur > precision
    yG = log(xG);
    yD = log(xD);
    xM = (xG + xD)/2;
    yM = log(xM);
    if yM*yG > 0
        xG = xM;
    else
        xD = xM;
    end
    erreur= abs(xD-xG);
    iter=iter+1;
    resultat=sprintf('%5d␣%12.4f␣%12.4f␣%12.4f',iter,xG,xD,erreur);
    disp(resultat)
end
```

FIGURE 6.47 – Calcul de la racine d'une fonction nonlinéaire par la méthode de bissection

Ce type de calcul itératif se prête idéalement pour l'énoncé **`while`**. Une autre approche serait l'utilisation de la structure **`for`**, combinée avec l'énoncé conditionnel **`if`**. On illustre avec un algorithme pour calculer la racine carrée d'un nombre réel. Cette méthode consiste à trouver la racine de A comme le coté d'un rectangle dont l'aire est donnée par,

$$\begin{aligned} aire &= A \\ &= largeur \times hauteur \\ &= L \times H \end{aligned}$$

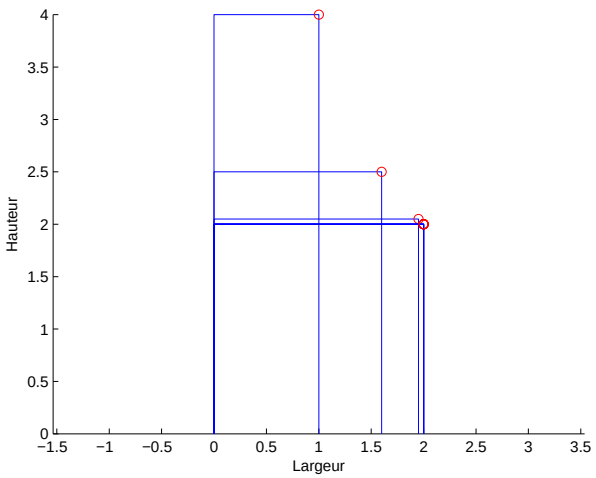
À partir d'un rectangle initial, on modifie successivement, la largeur L et la hauteur H de sorte que l'aire $L \times H$ soit toujours égale à A.

1. On pose A et les valeurs initiales de la largeur L et la hauteur H qui sont choisies de sorte que $A = L \times H$;
2. On calcule la nouvelle valeur de $H = .5 * (L + H)$;
3. On calcule la nouvelle valeur de $L = A/H$;
4. On répète de 2) à 3).

Ceci engendre deux suites de valeurs pour L_i et H_i qui convergent vers une même valeur, c'est-à-dire le coté d'un carré, qui est la racine carrée de A. Le résultat donne la suite d'itérations suivantes pour la largeur et la hauteur qui convergent vers 2.0000.

Le script pour ce calcul est montré à la Fig. 6.48.

```
Calcul de la racine carree
#     Largeur     Hauteur
0     1.00000     4.00000
1     1.60000     2.50000
2     1.95122     2.05000
3     1.99939     2.00061
4     2.00000     2.00000
5     2.00000     2.00000
6     2.00000     2.00000
```



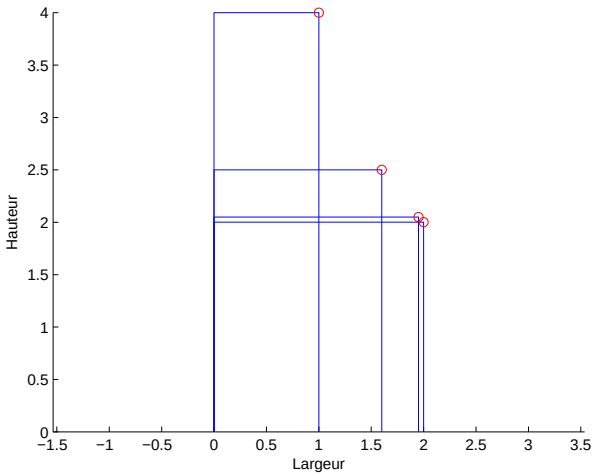
La différence principale entre les deux exemples précédents est que dans le premier cas la structure **`while`** permet d'intégrer un test pour le contrôle du nombre d'itérations. Avec la structure inconditionnelle **`for`**, le nombre d'itérations est fixé quelle que soit la précision atteinte. On peut remédier à cette lacune, en insérant à l'intérieur de la boucle **`for`**, un test sur un critère de convergence avec l'énoncé **`if`**. Si la condition est vérifiée, alors, l'énoncé **`break`** permet de sortir de la boucle **`for`**.

Les modifications sont indiquées à la Fig. 6.49. Son exécution donne la suite d'itérations suivantes pour la largeur et la hauteur qui convergent vers 2.0000 en 3 itérations pour un écart entre L_i et H_i inférieur à 1% des valeurs de départ.

```
%-----------------------------------------------------------------
% Algorithme pour le calcul de la racine carre d'un nombre A
% a partir de 2 estimes H et L
% On calcule une suite de rectangles de hauteur H et largeur L
% dont la surperfie est egale a A=HxL
%-----------------------------------------------------------------
clc; close all; clear all
%-----------------------------------------------------------------
sousTitre='    #    Largeur    Hauteur';
titre='Calcul de la racine carree';
A=4;%------------------------------------------ aire du rectangle
L=1;%------------------------------------------- largeur initiale
H=4;%------------------------------------------- hauteur initiale
disp(titre);
disp(sousTitre);
iter=0;
iteration=sprintf('%5d %10.5f %10.5f',iter,L,H);
disp(iteration)
figure(1)
x=[0 0 L L];%------------------- coordonnees du rectangle initial
y=[0 H H 0];
axis equal
hold on
xlabel('Largeur')
ylabel('Hauteur')
plot(x,y)
plot(L,H,'or')
for iter=1:10%---------------------------- execute 10 iterations
    H=.5*(L+H);%------------------- nouvelle valeur de la hauteur
    L=A/H;%------------------------ nouvelle valeur de la largeur
    iteration=sprintf('%5d %10.5f %10.5f',iter,L,H);
    disp(iteration)
    x=[0 0 L L];%--------------- coordonnees du rectangle courant
    y=[0 H H 0];
    plot(x,y)
    plot(L,H,'or')
end
hold off
```

FIGURE 6.48 – Calcul de la racine carrée d'un nombre réel par une suite de rectangles

```
#     Largeur     Hauteur
0     1.00000     4.00000
1     1.60000     2.50000
2     1.95122     2.05000
3     1.99939     2.00061
```



Les deux approches, soit avec **`while`** ou bien **`for`** et **`break`** nécessitent une solution de départ qui donne la plage de la solution.

```
%-------------------------------------------------------------------
% Modification: un test de convergence sur l'ecart entre L et H
%-------------------------------------------------------------------
for iter=1:10%---------------------------- execute 10 iterations
    H=.5*(L+H);%--------------------nouvelle valeur de la hauteur
    L=A/H;%-------------------------nouvelle valeur de la largeur
    iteration=sprintf ('%5d %10.5f %10.5f',iter,L,H);
    disp(iteration)
    x=[0 0 L L]; y=[0 H H 0];%-- coordonnees du rectangle courant
    plot(x,y);     plot(L,H,'or')
    if abs(L-H)<erreur%--------------------- test de convergence
        break
    end
end
```

FIGURE 6.49 – Calcul de la racine carrée d'un nombre réel avec test de convergence

6.8 Intersection de courbes

Un grand nombres de problèmes en ingénierie donnent lieu à des calculs qui s'interprêtent gémétriquement comme l'intersection de courbes. Lorsque ces courbes sont des polynômes de degré petit, le résultat peut être obtenu analytiquement, sinon une procédure itérative doit être utlisée.

6.8.1 Courbes polynômiales

On illustre la démarche avec l'intersection d'un polynôme de degré n et une droite. Soit,

$$P_n(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$$

et

$$P_1(x) = b_0 + b_1x^1$$

Les points d'intersection sont donnés par,

$$P_n(x) = P_1(x)$$

Ce qui donne, après quelques manipulations, l'équation du n-ième degré suivante,

$$a_nx^n + ... + a_2x^2 + (a_1 - b_1)x + (a_0 - b_0) = 0$$

Trouver l'intersection des courbes $P_n(x)$ et $P_1(x)$ revient donc à trouver les racines d'un polynôme de degré n. Il existe une solution analytique pour des polynômes de degré inférrieur à 4. Par exemple, pour n=2, les racines sont

$$x_{1,2} = \frac{-(a_1 - b_1) \pm \sqrt{(a_1 - b_1)^2 - 4a_2(a_0 - b_0)}}{2a_2}$$

Pour l'intersection d'un polynôme de degré trois et une droite, on obtient l'équation du troisième degré suivante,

$$a_3x^3 + a_2x^2 + (a_1 - b_1)x + (a_0 - b_0) = 0$$

Trouver l'intersection des courbes $P_3(x)$ et $P_1(x)$ revient donc à trouver les racines de

$$P_3(x) - P_1(x) = A_3x^3 + A_2x^2 + A_1x + A_0 = 0$$

Le nombre et type de racines de cette équation dépend de la valeur du discriminant :

D > 0 : Une racine réelle, et deux racines complexes ;

D < 0 : Trois racines réelles distinctes ;

D = 0 : Trois racines réelles, dont deux identiques.

Le discriminant est défini par,

$$D = q^3 - r^2,$$

où

$$q = B/3 - A^2/9$$

et

$$r = (B * A - 3 * C)/6 - A^3/27$$

Dans l'exemple suivant, le polynôme $P_3(x)$ est défini pas ses coefficients a, b, c et d, et est affiché à la Fig. 6.50. Sur cette fenêtre, la fonction **`ginput`** [1] lance un curseur graphique

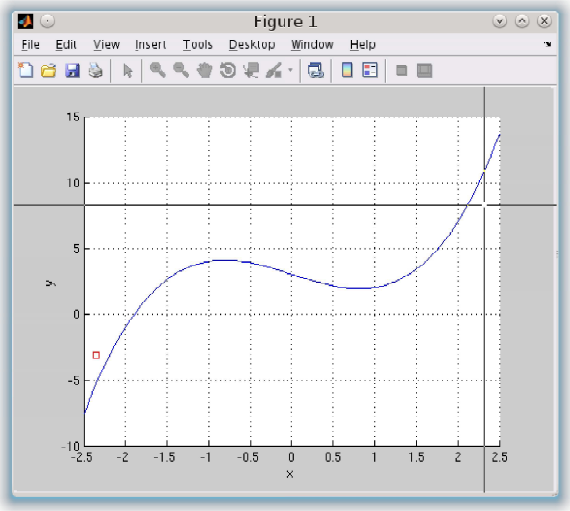
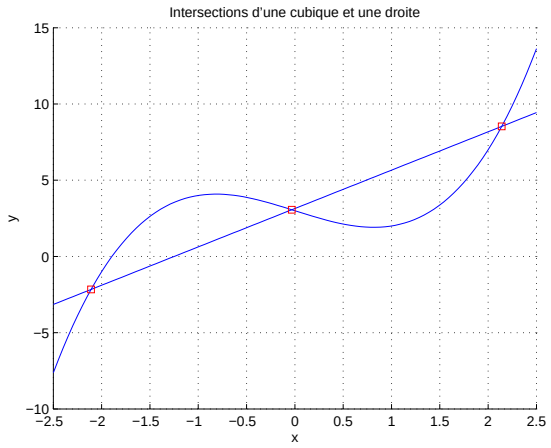


FIGURE 6.50 – Construction interactive de la droite $P_1(x) = b_0 + b_1 x^1$ intersectant $P_3(x)$

que l'usager positionne sucessivement en deux points au travers desquels la droite est construite.

Le source de cette partie du programme, qui construit les deux courbes, est montré à la Fig. 6.52. A partir de ces paramètres, les deux courbes sont construites et ensuite à partir du calcul du discriminant, les racines sont calculées comme montré au programme de la Fig. 6.53.

Le résultat est imprimé dans la sortie standard (fenêtre de commande sous Matlab) tel que montré et illustré graphiquement ci-dessous,




```
Trois racines reelles distinctes
  2.0721   -2.1446    0.0725

>>
```

FIGURE 6.51 – Intersections des courbes $P_1(x)$ et $P_3(x)$ calculées par le script de la 6.53

1. Voir Section 4.1.3

```
%-----------------------------------------------------------------
% Calcul de l'intersection entre une cubique et une droite
% P3(x) = ax^3 + bx^2 + bx + c     et      P1(x) =  Mx + N
% P3(x) -P1(x) = Ax^3 + Bx^2 + Cx + D=0
% Le discriminant DD = Q*Q + 4*P^3/27
% ou Q = b*(2*b*b/a/a - 9*c/a)/27/a + d/a;
% et P = -b*b/3/a/a + c/a;
% dÃl'termine le nombre et le type de racines.
%-----------------------------------------------------------------
clc; clear all; close all
%-----------------------------------------------------------------
a= 1.0;%------------------- Initialisation des coefficients de P3
b= 0.0;
c= -2.0;
d= 3.0;
xINI = -2.5;
xFIN = 2;
nPTS = uint8(100);
dx = (xFIN-xINI)/double(nPTS-1);
x = (xINI:dx:xFIN);
y = a*x.^3 + b*x.^2 + c*x + d;
figure(1)%------------------------------------------- Affiche P3
hold on; grid on
plot(x,y)
[x1,y1,but]=ginput(1); [x2,y2,but]=ginput(1);%------ saisie de P1
M=(y1-y2)/(x1-x2);
N=y1-M*x1;
z=M*x +N;
plot(x,z)%------------------------------------------- Affiche P1
hold off
```

FIGURE 6.52 – Intersection d'une cubique et une droite. Partie 1 : Spécification des paramètres

```
%----------------- Suite du programme -------------------------
A=a;%------------------------------ definition de P3(x)-P1(x)=0
B=b; C=c-M; D=d-N; Q= B*(2*B*B/A/A - 9*C/A)/27/A + D/A;
P= -B*B/3/A/A + C/A; DD = Q*Q + 4*P^3/27;% Calcul du discriminant
%-------------------- Nombre et type de racines de P3(x)-P1(x)=0
if DD > 0
    message ='Une racine rÃl'elle, et deux racines complexes';
    dd = sqrt(DD);
    U=nthroot(((-Q + dd)/2),3);
    V=nthroot(((-Q - dd)/2),3);
    xR(1) = U + V -b/3/a ;
    nbRacine = 1;
elseif DD < 0
    message ='Trois racines rÃl'elles distinctes';
    RR=acos(-Q*sqrt(-27/P/P/P)/2)/3;
    QQ=2*sqrt(-P/3);
    RQ=-b/3/a;
    xR(1) = QQ*cos(RR           ) +RQ;
    xR(2) = QQ*cos(RR +   2*pi/3) +RQ;
    xR(3) = QQ*cos(RR + 2*2*pi/3) +RQ;
    nbRacine = 3;
elseif abs(DD) < .000001
    message ='Trois racines rÃl'elles, dont deux identiques';
    if abs(P) < .00000000001
        xR(1)=0.0;       xR(2)=0.0;           xR(3)=0.0;
    else
        xR(1)=3*Q/P;     xR(2)=-3*Q/2/P;      xR(3)=xR(2);
    end
    nbRacine = 3;
end
disp(message)%--------------------------- Ecriture des resultats
yR = M*xR + N;
disp(xR(1:nbRacine))
figure(2)%-------------- Affiche les courbes et les intersections
hold on
plot(x,y); plot(x,z);
plot(xR(1:nbRacine),yR(1:nbRacine),'sr')
grid on
hold off
```

FIGURE 6.53 – Intersection d'une cubique et une droite. Partie 2 : Calcul des racines

6.8.2 Courbes nonlinéaires

Dans la situation où les courbes sont représentées par des fonctions nonlinéaires, tel que **log**, **sin** .... l'approche de la section précédente n'est plus possible et on doit utiliser une procédure itérative.

Soit deux courbes quelconques, $g(x)$ et $h(x)$. L'intersection est donnée par la condition $g(x) = h(x)$. Ce qui revient à trouver les racines de

$$g(x) - h(x) = 0$$

avec la méthode de bissection présentée à la Section 6.7.

Dans l'exemple suivant on applique cette méthode à l'intersection de

$$g(x) = sin(x) \text{ et } h(x) = log(x)$$

qui comprend trois étapes.

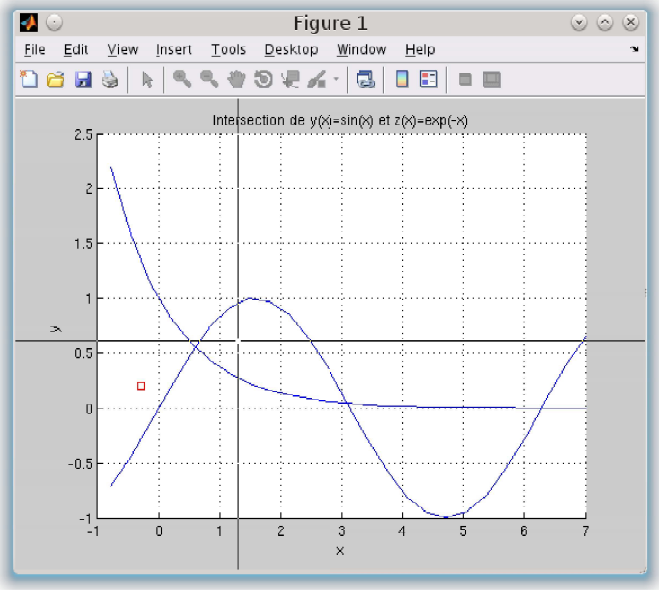


FIGURE 6.54 – Saisie interactive de l'encadrement de l'intersection

Dans un premier temps, dans le programme de la Fig. 6.56, on construit les fonctions $g(x) = sin(x)$ et $h(x) = log(x)$ qui sont affichées. Dans cette fenêtre graphique, l'appel à la fonction **ginput** donne un curseur que l'usager positionne pour saisir deux points(xG, yG) et (xD, yD) qui encadrent l'intersection, tel qu'illustré à la Fig. 6.54.

```
[xG,yG,but]=ginput(1);%saisie de l'intervalle encadrant la racine
plot(xG,yG,'sr')
[xD,yD,but]=ginput(1);
plot(xD,yD,'sr')
```

Ensuite, dans le script de la Fig. 6.57, on saisi la valeur de la précision, avec une boîte de dialogue, **inputdlg**, tel qu'illustré à la Fig. 6.55, ci-dessous.

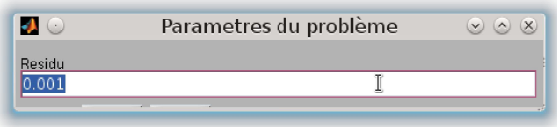


FIGURE 6.55 – Saisie de la précision du calcul de la racine

```
%-----------------------------------------------------------------
% Intersection entre deux courbes: y(x)=sin(x) et  z(x)=exp(-x)
% par la methode de bissection, tel que f(x)=sin(x) - exp(-x) =0
% L'usager doit fournir la fonction f(x) sous la forme de deux
% fichiers: gINT(x)=sin(x) et hINT(x)=exp(-x)
% La methode necessite deux valeurs xG et xD encadrant la racine.
%-----------------------------------------------------------------
clc; clear all; close all
%-----------------------------------------------------------------
xIni=-.25*pi;%----------------- definition de la plage de calcul
xFin=7.0;
nPoints = uint8(25);
dx= (xFin -xIni)/double(nPoints -1);
x = [xIni:dx:xFin];
for i=1:nPoints
    g(i) = gINT(x(i));    h(i) = hINT(x(i));
end
figure(1)
hold on
grid on
title('Intersection de y(x)=sin(x) et z(x)=exp(-x)')
xlabel('x') ylabel('y')
plot(x,g) plot(x,h)
[xG,yG,but]=ginput(1);%saisie de l'intervalle encadrant la racine
plot(xG,yG,'sr')
[xD,yD,but]=ginput(1);
plot(xD,yD,'sr')
hold off
```

FIGURE 6.56 – Intersection de courbes nonlinéaires. 1ère partie : encadrement de l'ntersection

La mise en oeuvre de l'algorithme de bissection est montré au programme de la Fig. 6.60. La convergence de procédé itératif est imprimé dans la fenêtre de commande (Fig. 6.58), et illustré graphiquement à la Fig. 6.59.

```
%-------------------------------------- saisie  de la precision
prompt={'Residu'};
name='Parametres du problÃÍme';
numlines=1;
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';
defaultanswer={'0.001'};
N=inputdlg(prompt,name,numlines,defaultanswer,options);
precision =str2double(N{1});
```

FIGURE 6.57 – Intersection de courbes nonlinéaires. 2ième partie : la précision

```
INTERSECTION DE DEUX COURBES PAR LA METHODE DE BISSECTION
 Iteration     xG         xD     Intervalle    Residu
      0     -0.1244     2.3088     2.4332     -0.5521
      1     -0.1244     1.0922     1.2166      0.1512
      2      0.4839     1.0922     0.6083     -0.2542
      3      0.4839     0.7880     0.3041     -0.0645
      4      0.4839     0.6359     0.1521      0.0402
      5      0.5599     0.6359     0.0760     -0.0130
      6      0.5599     0.5979     0.0380      0.0134
      7      0.5789     0.5979     0.0190      0.0002
      8      0.5884     0.5979     0.0095     -0.0064
      9      0.5884     0.5932     0.0048     -0.0031
     10      0.5884     0.5908     0.0024     -0.0015
     11      0.5884     0.5896     0.0012     -0.0007
L'intersection ( 0.5887, 0.5550) a ete calculee avec
un intervalle =<  0.0003, et un residu = -0.0003
apres   12 itÃl'rations
```

FIGURE 6.58 – Intersection de courbes nonlinéaires. Convergence du calcul itératif

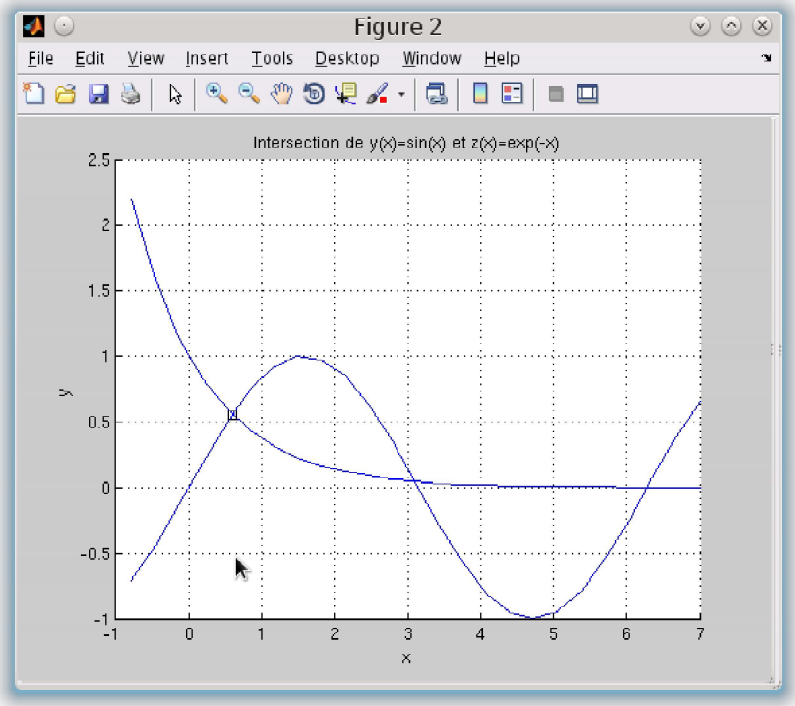


FIGURE 6.59 – Intersections des courbes $g(x) = sin(x)$ et $h(x) = log(x)$ calculées par l'algorithme donné à la Fig. 6.60

```
intervalle= abs(xD-xG);%------------------------ erreur initiale
yG = gINT(xG) - hINT(xG); yD = gINT(xD) - hINT(xD);
xM = .5*(xD+xG);          yM = gINT(xM) - hINT(xM);
residu= yM;%------------------------------------ residu initial
nbIteration =0;%---------------- On initialise le compteur a zero
sousTitre='␣Iteration␣␣␣␣␣xG␣␣␣␣␣␣␣␣␣xD␣␣␣␣␣Intervalle␣␣␣␣Residu';
titre='INTERSECTION␣DE␣DEUX␣COURBES␣PAR␣LA␣METHODE␣DE␣BISSECTION';
disp(titre)
disp(sousTitre)
while intervalle > precision
    resultat=sprintf('%7d␣␣%10.4f␣%10.4f␣%10.4f␣%11.4f␣',
                     nbIteration,xG,xD,intervalle,residu);
    disp(resultat)
    nbIteration=nbIteration + 1;%----- Le compteur est incremente
    if yM*yG > 0
        xG = xM;
    else
        xD = xM;
    end
    yG = gINT(xG) - hINT(xG);     yD = gINT(xD) - hINT(xD);
    xM=.5*(xD+xG);                yM = gINT(xM) - hINT(xM);
    intervalle= abs(xD-xG);       residu= yM;
end
%---------------------------------------- On imprime la reponse
zM=gINT(xM);%zM=hINT(xM);
R1=sprintf('L''intersection␣(%7.4f,%7.4f)␣',xM,zM);
intervalle=abs(xG - xD)/2;
R2=sprintf('avec␣un␣intervalle␣=<␣%7.4f,␣et␣un␣residu␣=␣%7.4f',
           intervalle,residu);
R3=sprintf('apres␣%4d␣itÃl'rations', nbIteration);
disp(R1); disp(R2); disp(R3)
figure(2)%------------------------------- trace de la solution
hold on; grid on
xlabel('x'); ylabel('y␣=␣f(x)')
title('Intersection␣de␣y(x)=sin(x)␣et␣z(x)=exp(-x)')
xlabel('x');ylabel('y')
plot(x,y);  plot(x,z)
plot(xM,zM,'sk')
hold off
```

FIGURE 6.60 – Intersection de courbes nonlinéaires. 3ième partie : algorithme de bissection