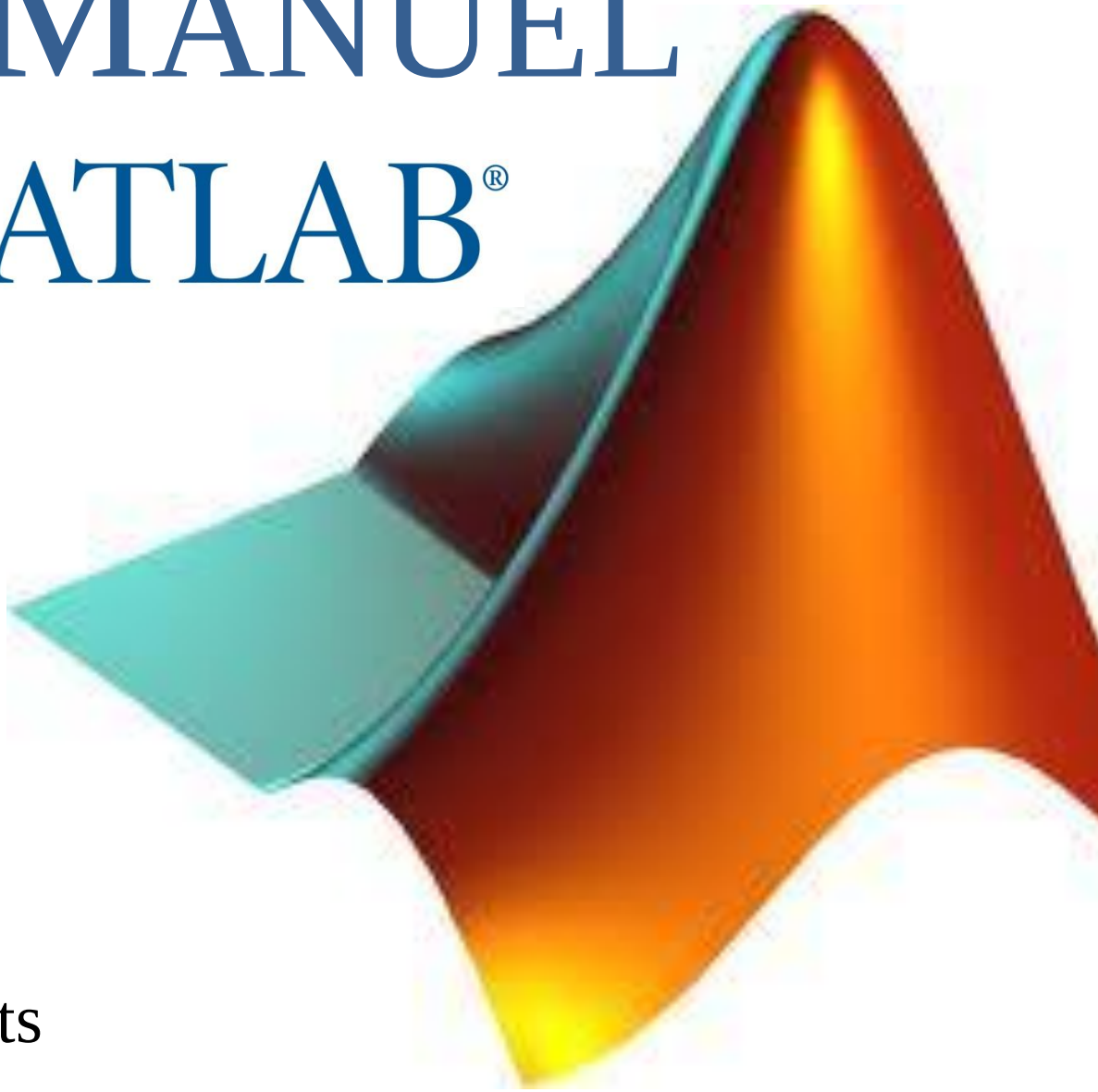
MANUEL MATLAB®



Départements

GEI & Mécanique

Version 0.1

Yassine Ariba - Jérome Cadieux

Icam de Toulouse

Auteurs : Yassine Ariba et Jérome Cadieux.

Table des matières

I. Introduction 6
1 - Qu'est ce que Matlab® ? 6
2 - Objectifs 7
3 - Logiciels alternatifs 7
II. Généralités 9
1 - Interface principale 9
2 - Premiers pas sur des exemples 10
2.1 Calcul et tracé d'une fonction sinus 10
2.2 Résolution d'un système d'équations linéaires 11
3 - Les variables 12
3.1 Aspects élémentaires 12
3.2 Constantes prédéfinies 13
3.3 Quelques fonctions mathématiques 14
4 - Vecteurs et Matrices 14
4.1 Définition d'un vecteur 14
4.2 Quelques fonctions utiles 15
4.3 Définition d'une matrice 16
4.4 Quelques fonctions utiles 18
4.5 Opérations sur les matrices 19
5 - Représentations graphiques 21
5.1 Graphiques 2D 21
5.2 Graphiques 3D 25
6 - Programmation avec Matlab 28
6.1 Fichiers `SCRIPT` 29
6.2 Fichiers `FUNCTION` 30
6.3 Opérateurs relationnels et logiques 32
6.4 Structures de contrôle 33
III. Application à la Mécanique 37
1 - Les structures 37
IV. Application à l'Automatique 39
1 - Représentation des systèmes linéaires invariants 39
1.1 Fonctions de transfert 39
1.2 Représentation d'état 40
1.3 Systèmes discrets et échantillonnés 41
1.4 Conversions de modèles 42
1.5 Connexions de systèmes 43

2 - Analyse des systèmes dynamiques 44
2.1 Quelques fonctions utiles 44
2.2 Réponses temporelles.....45
2.3 Lieux de transfert 47
3 - Simulink 47
3.1 Création d'un modèle 48
3.2 Quelques bibliothèques.....50
3.3 Simulation.....52
3.4 Exemple.....53

I. INTRODUCTION

Ce document est une introduction à Matlab, un logiciel de calcul scientifique. Il a pour objectif de préparer l'étudiant aux travaux pratiques d'Automatique, de Mécanique et d'Analyse Numérique dans lesquels cet outil est intensivement utilisé pour la mise en application et la simulation des principes théoriques présentés en cours. Par ailleurs, ce manuel offre la possibilité à l'étudiant de se former à un logiciel professionnel largement répandu.

1 - Qu'est ce que Matlab® ?

Matlab est un logiciel de calcul numérique commercialisé par la société MathWorks[1]. Il a été initialement développé à la fin des années 70 par Cleve Moler, professeur de mathématique à l'université du Nouveau-Mexique puis à Stanford, pour permettre aux étudiants de travailler à partir d'un outil de programmation de haut niveau et sans apprendre le Fortran ou le C.

Matlab signifie **Mat**rix **lab**oratory. Il est un langage pour le calcul scientifique, l'analyse de données, leur visualisation, le développement d'algorithmes. Son interface propose, d'une part, une fenêtre interactive type console pour l'exécution de commandes, et d'autre part, un environnement de développement intégré (IDE) pour la programmation d'applications.

Matlab trouve ses applications dans de nombreuses disciplines. Il constitue un outil numérique puissant pour la modélisation de systèmes physiques, la simulation de modèles mathématiques, la conception et la validation (tests en simulation et expérimentation) d'applications. Le logiciel de base peut être complété par de multiples *toolboxes*, c'est-à-dire des boîtes à outils. Celles-ci sont des bibliothèques de fonctions dédiées à des domaines particuliers. Nous pouvons citer par exemple : l'Automatique, le traitement du signal, l'analyse statistique, l'optimisation…

Voici une liste non exhaustive (loin de là) et en vrac de toolboxes, montrant la diversité des fonctionnalités de Matlab :

Control System Toolbox	Symbolic Math Toolbox	Signal Processing Toolbox
Neural Network Toolbox	Optimization Toolbox	Parallel Computing Toolbox
Statistics Toolbox	Fuzzy Logic Toolbox	Image Processing Toolbox
Aerospace Toolbox	Data Acquisition Toolbox	Bioinformatics Toolbox
MATLAB Compiler	Vehicle Network Toolbox	Model-Based Calibration Toolbox
Financial Toolbox	RF Toolbox	System Identification Toolbox

[1] http://www.mathworks.com/

2 - Objectifs

Ce document propose une introduction à Matlab et développe un ensemble de fonctionnalités spécifiques à certains domaines des sciences de l'ingénieur. Il ne constitue en aucun cas une documentation exhaustive du logiciel. Toutefois, les principales notions sont présentées et invitent l'étudiant à chercher par lui-même les informations complémentaires pour mener à bien son projet. En plus de l'aide intégrée à l'environnement et des nombreux ouvrages dédiés, une quantité abondante de ressources sont disponibles sur Internet :

Documentation en ligne de MathWorks : http://www.mathworks.com/help/techdoc/

Espaces d'entraide :

- Developpez.com : http://matlab.developpez.com/
- Matlab Central : http://www.mathworks.com/matlabcentral/

Notes mises en ligne par des Universités/Ecoles :
(par exemples)

- http://nte.mines-albi.fr/MATLAB/co/Matlab_web.html
- http://personnel.isae.fr/sites/personnel/IMG/pdf/InitiationMatLab.pdf
- http://perso.telecom-paristech.fr/~prado/enseignement/polys/matlab.html
- http://asi.insa-rouen.fr/enseignement/siteUV/tds/
- http://www-gmm.insa-toulouse.fr/~roussier/poly_info_icbe.pdf
- …

Il existe également un espace d'échange de codes et applications Matlab :
File exchange : http://www.mathworks.com/matlabcentral/fileexchange/

L'objectif de ce document est double. L'étudiant doit acquérir une connaissance suffisante du logiciel pour pouvoir travailler efficacement et comprendre les notions vues dans les travaux pratiques d'Automatique, de Mécanique et d'Analyse Numérique. Le second objectif est de proposer une initiation consistante à un logiciel puissant, très largement utilisé dans le monde industriel et académique (laboratoires, universités/écoles).

3 - Logiciels alternatifs

Si le prix d'une licence[2] Matlab, type éducation (classroom), est relativement intéressant (<~100€), celui d'une version industrielle est plutôt onéreux (>~2500€). A cela il faut ajouter un coût supplémentaire pour chaque toolbox commandée.

Voici quelques solutions alternatives.

Scilab est un logiciel open-source sous licence GPL (ou du moins dans l'esprit). Développé depuis 1990 par des chercheurs de l'INRIA (institut national de recherche en informatique et automatique), il est maintenant maintenu par la fondation de coopération scientifique Digiteo[3]. Il est disponible sur les plateformes Windows, Mac OS X, Linux et BSD.

Pour plus d'informations et pour télécharger le logiciel : http://www.scilab.org/

[2] Les valeurs données ne sont que des ordres de grandeur et peuvent changer. Les prix dépendent du type de licence (éducation, industriel, recherche) et du format (individuel, groupe, concurrent).
[3] http://www.digiteo.fr/fr

Octave est également un logiciel open-source sous licence GPL. Son développement a commencé au début des années 90 par John W. Eaton dans le cadre du projet GNU[4]. Sa syntaxe est proche de celle de Matlab. Il est disponible sur les plateformes Windows, Mac OS X, Linux et BSD.

Pour plus d'informations et pour télécharger le logiciel : http://www.gnu.org/software/octave/

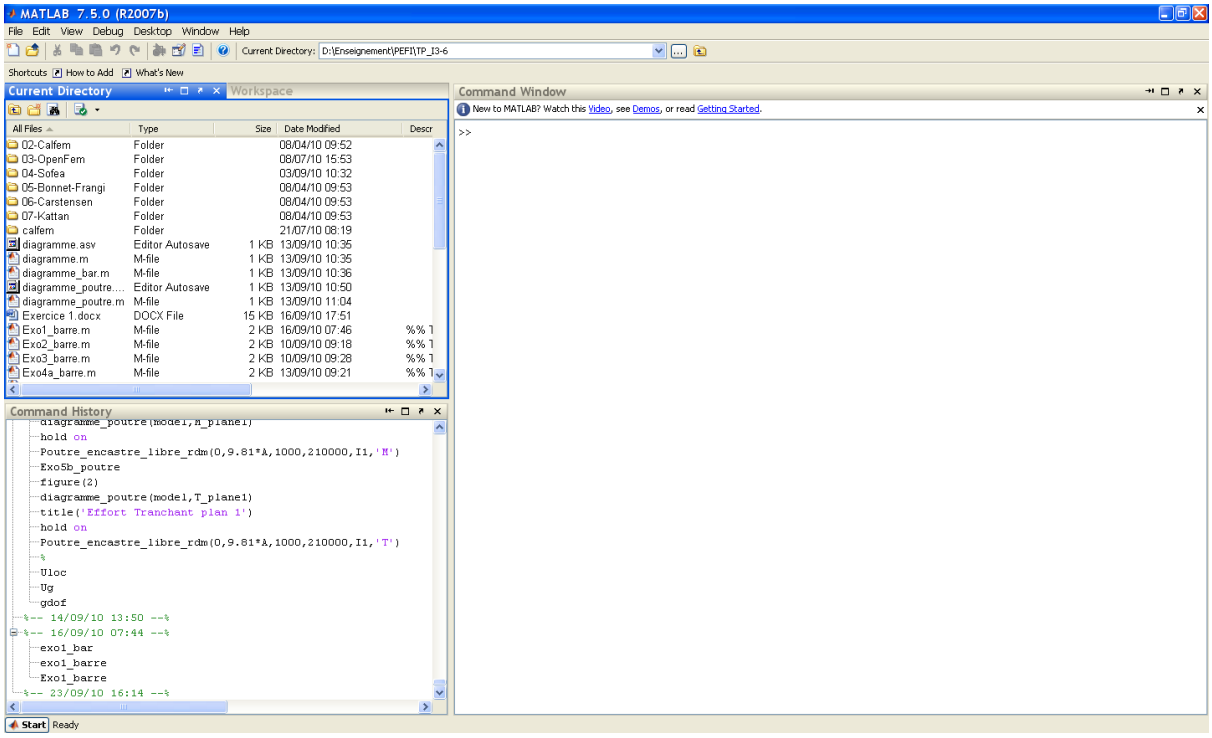
[4] http://www.gnu.org/

II. GENERALITES

Dans ce chapitre, l'environnement de Matlab est présenté. Un premier exemple introductif montre rapidement le principe de fonctionnement du logiciel. Nous présentons ensuite un ensemble de fonctions de base nécessaire pour débuter en Matlab.

1 - Interface principale

Au lancement de Matlab, l'interface suivante apparait :



Le logiciel propose un véritable environnement de travail composé de multiples fenêtres. Nous pouvons distinguer quatre blocs :

- ***Command window*** (console d'exécution) : à l'invite de commande « >> », l'utilisateur peut entrer les instructions à exécuter. Il s'agit de la fenêtre principale de l'interface.
- ***Current directory*** (répertoire courant) : permet de naviguer et de visualiser le contenu du répertoire courant de l'utilisateur. Les programmes de l'utilisateur doivent être situés dans ce répertoire pour être visible et donc exécutable[5].
- ***Workspace*** (espace de travail) : permet de visualiser les variables définies, leur type, la taille occupée en mémoire…
- ***Command history*** : historique des commandes que l'utilisateur a exécutées. Il est possible de faire glisser ces commandes vers la fenêtre de commande.

Notons que la ***command window*** est la fenêtre centrale de l'interface, c'est à partir de là que l'utilisateur pourra lancer les commandes interprétées par Matlab. Le principe est simple et intuitif, le tout est de connaitre les fonctions appropriées et de respecter leur syntaxe. Premier exemple élémentaire : à l'invite de commande, taper « `3*5` », puis entrer :

```
>> 3*5
ans =
      15
```

A la validation de l'instruction, l'interface affiche le résultat de cette dernière. Afin d'alléger l'affichage, un point-virgule « **;** » en fin de commande empêche le renvoie du résultat dans la fenêtre (évidemment l'instruction est toujours exécutée). Par exemple :

```
>> 3*5;
>>
```

Le calcul a été effectué mais le résultat n'est pas affiché.

2 - Premiers pas sur des exemples

Nous proposons en premier lieu un démarrage rapide basé sur un exemple simple. Celui-ci a pour objectif de montrer comment se pratique Matlab dans une première utilisation basique. Cette section doit donc être parcourue avec Matlab à coté. Les lecteurs non débutants pourront passer cette section.

2.1 Calcul et tracé d'une fonction sinus

La manipulation commence dans la fenêtre ***command window***, à l'invite de commande :

```
>> t = [0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6]

t =

  Columns 1 through 8

         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
3.5000

  Columns 9 through 13

    4.0000    4.5000    5.0000    5.5000    6.0000
```

Cette ligne de commande définit un tableau de 13 valeurs (allant de 0 à 6 par incrément de 0.5) nommé «`t`». On comprend l'utilité de mettre un « **;** » à la fin de la ligne pour éviter l'affichage

[5] Cette condition n'est pas nécessaire, si les programmes sont situés dans un répertoire spécifié dans le ***PATH***.

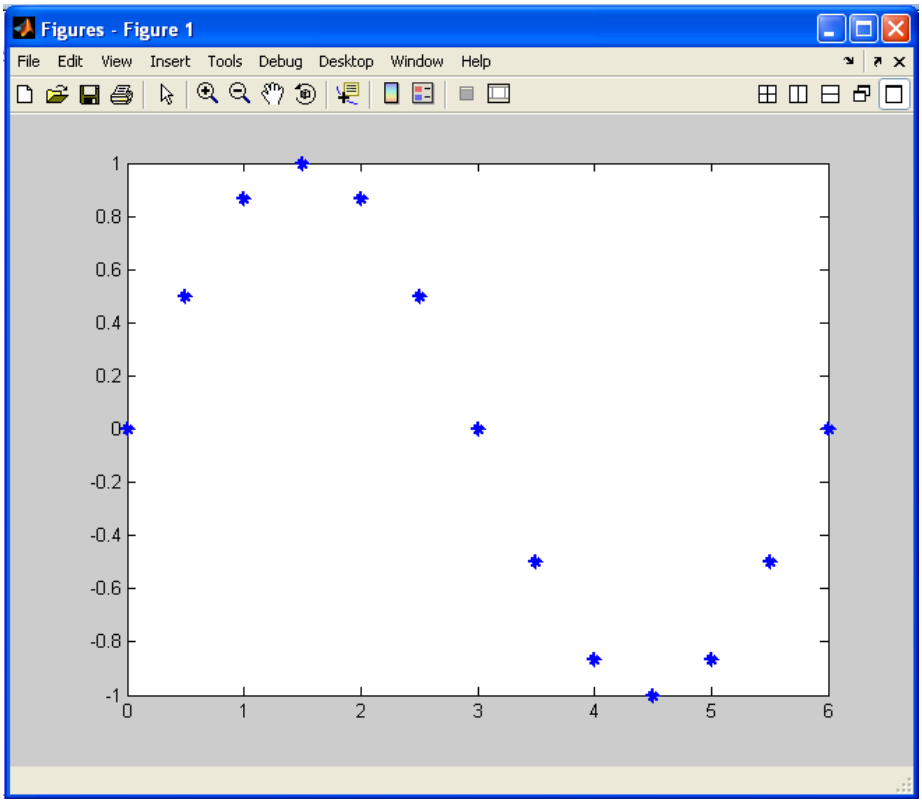
systématique du résultat des opérations envoyées. Cela peut devenir très lourd si, par exemple, la taille du tableau est importante.

```
>> w = 2*pi/6;
>> y = sin(w*t);
```

Le terme « `pi` » est une constante prédéfinie et donne donc la valeur de π. Le tableau (ou vecteur) « `y` », de même dimension que « `t` », contient les valeurs résultantes de l'opération appliquée à chaque composante de « `t` ». « `y` » représente donc la fonction sinus de période 6 (de pulsation « `w` »). Il est possible de représenter graphiquement les points du tableau à l'aide de la fonction suivante :

```
>> plot(t,y,'*');
```

Une nouvelle fenêtre s'ouvre :



2.2 Résolution d'un système d'équations linéaires

Nous souhaitons résoudre le système d'équation suivant :

$$\begin{cases} 2x + y = -5 \\ 4x - 3y + 2z = 0 \\ x + 2y - z = 1 \end{cases}$$

Ecriture sous forme matricielle AX=B:

$$\begin{bmatrix} 2 & 1 & 0 \\ 4 & -3 & 2 \\ 1 & 2 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -5 \\ 0 \\ 1 \end{bmatrix}$$

La manipulation dans Matlab commence donc par la définition des matrices.

```
>> A = [2 1 0 ; 4 -3 2 ; 1 2 -1]
A =
     2     1     0
     4    -3     2
     1     2    -1
>> B = [-5;0;1]
B =
    -5
     0
     1
```

Si A est une matrice inversible alors le système admet une solution unique qui s'exprime par

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

Le logiciel peut faire ce calcul pour nous :

```
>> X = inv(A)*B
X =
    1.7500
   -8.5000
  -16.2500
```

La solution de notre système d'équation est donc :

$$x = 1.75 \qquad y = -8.5 \qquad z = -16.25$$

3 - Les variables

3.1 Aspects élémentaires

Matlab gère de façon automatique les nombres entiers, réels, complexes, les chaînes de caractères… Ainsi, la déclaration des variables est implicite, et la mémoire nécessaire est automatiquement allouée lors de la définition de celles-ci. Le symbole d'affectation est le signe « = »

```
>> x = 4
x =
4
>> y = 2
y =
2
>> x + y
ans =
6
>> x * y
ans =
8
```

Si l'utilisateur n'affecte pas explicitement le résultat d'une opération à une variable, Matlab l'affecte automatiquement à la variable « `ans` ». Concernant le nom des variables, l'interpréteur fait la distinction entre les minuscules et les majuscules. A la définition d'une variable, celle-ci apparait, accompagnée de quelques informations, dans la fenêtre ***Workspace***. Il est également possible de connaitre les variables du workspace (espace de travail) via l'instruction « `whos` ». Par exemple, après les manipulations précédentes :

```
>> whos
Name        Size        Bytes       Class       Attribute
ans         1x1         8           double
x           1x1         8           double
y           1x1         8           double
```

La commande « `clear` » permet de supprimer une variable du workspace (« `clear all` » les supprime toutes). Toutes les commandes tapées dans la ***Command window*** peuvent être retrouvées et éditées grâce aux touches de direction. Appuyez sur la touche ⇑ pour remonter dans les commandes précédentes, ⇓ pour redescendre.

3.2 Constantes prédéfinies

Il existe des symboles auxquels sont associés des valeurs prédéfinies. En voici quelques uns :

Symbole	Signification	Valeur
`pi`	Nombre π	3.141592...
`i` ou `j`	Nombre complexe	$\sqrt{-1}$
`realmax`	Plus grand nombre flottant codable	1.7977e+308
`realmin`	Plus petit nombre flottant codable	2.2251e-308

Attention, ces valeurs peuvent être écrasées si le symbole est redéfini.

```
>> pi=1
pi =
     1
>> clear pi
>> pi
ans =
     3.1416
```

Il est donc possible de définir et manipuler explicitement des nombres complexes.

```
>> a = 2 + i*3;
>> b = 1 - i;
>> a + b
ans =
     3 + 2i
```

3.3 Quelques fonctions mathématiques

Fonctions trigonométriques :

sin	cos	tan
asin	acos	atan
sinh	cosh	tanh

Fonctions puissances, exponentielle et logarithmes :

power (ou ^) sqrt exp log log10

Fonctions réels vers entiers (arrondi, troncature…):

fix floor round ceil

Fonctions autres :

`sign(var)`: (signe) retourne 1 si `var>0`, -1 si `var<0` et 0 si `var=0`.
`abs(var)`: module de `var`.
`real(var)`: partie réelle de `var`.
`imag(var)`: partie imaginaire de `var`.
`gcd(var1,var2)`: plus grand diviseur commun des entiers `var1` et `var2`.
`lcm(var1,var2)`: plus petit multiple commun des entiers `var1` et `var2`.

4 - Vecteurs et Matrices

Avec Matlab, on travaille essentiellement avec un type d'objet : les matrices[6]. Une variable scalaire est une matrice de dimension 1 x 1 et un vecteur est une matrice de dimension 1 x n ou n x 1. Il est capital d'être à l'aise avec ces notions pour comprendre au mieux la philosophie de Matlab et l'exploiter efficacement.

4.1 Définition d'un vecteur

Un vecteur n'est rien d'autre qu'un tableau de valeurs. Il existe plusieurs façons de créer un vecteur et la plus simple d'entre elles est de l'écrire explicitement.

```
>> v = [1 2 3 4]
v =
     1     2     3     4
```

L'ensemble des composantes est donné entre crochets et les valeurs sont séparées par un espace (ou une virgule « **,** »). Nous avons ici défini un vecteur ligne. Un vecteur colonne est créé en utilisant un point-virgule « **;** » comme délimiteur.

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

     1
     2
     3
```

[6] D'où son nom : MATrix LABoratory.

```
     4
```

Bien que simple, cette méthode n'est pas pratique pour définir des vecteurs de taille importante. Une seconde méthode utilise l'opérateur deux-points « **:** ». Il permet de discrétiser un intervalle avec un pas constant.

```
>> v = 0:0.2:1
v =
         0    0.2    0.4    0.6    0.8    1
```

Cette instruction crée un vecteur contenant des valeurs allant de 0 à 1 avec un pas de 0.2. La syntaxe est la suivante : `vecteur = valeur_initial:incrément:valeur_finale`. Par défaut, le pas est égal à 1.

```
>> v = 0:5
v =
         0    1     2     3    4     5
```

Enfin, des fonctions prédéfinies permettent de générer des vecteurs automatiquement.

```
>> v = linspace(0,10,1000);
>> v = logspace(-1,2,1000);
```

La première fonction crée un vecteur de 1000 points avec des valeurs allant de 0 à 10 également espacées. La seconde crée un vecteur de 1000 points sur un intervalle de 10^{-1} à 10^{2} avec un espacement logarithmique.

On peut accéder aux différents éléments d'un tableau en spécifiant un (ou des) indice(s) entre parenthèses.

```
>> v = [6 4 -1 3 7 0.3];
>> v(3)
ans =
      -1

>> v(2:4)
ans =
      4    -1     3
```

`v(3)` retourne le 3ième élément du vecteur `v`. L'argument `2:4` permet de sélectionner un bloc d'éléments (ici du second au quatrième).

4.2 Quelques fonctions utiles

Nous présentons dans ce paragraphe un ensemble de fonctions usuelles liées à l'utilisation des tableaux.

`length(v)`	renvoie la taille du tableau.
`max(v)`	renvoie la valeur maximale du tableau.
`min(v)`	renvoie la valeur minimale du tableau.
`mean(v)`	renvoie la valeur moyenne des éléments du tableau.
`sum(v)`	calcul la somme des éléments du tableau.

`prod(v)` calcul le produit des éléments du tableau.

`sort(v)` range les éléments du tableau dans l'ordre croissant.

Toutes les fonctions mathématiques vues au paragraphe II.3.3 sont applicables aux variables de type vecteur. Dans ce cas, la fonction est opérée sur chacun des éléments du vecteur.

```
>> v = [0 pi/4 pi/2 pi 2*pi]
 v =
         0    0.7854    1.5708    3.1416    6.2832
>> cos(v)
ans =
    1.0000    0.7071    0.0000   -1.0000    1.0000
```

4.3 Définition d'une matrice

La définition d'une matrice est délimitée pas des crochets « `[]` ». Les différents éléments d'une ligne sont séparés par un espace et les différentes lignes sont séparées par des points-virgules « `;` ». Ainsi pour définir une variable matricielle

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

on écrira :

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

ou

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

L'accès à un élément d'une matrice s'opère en spécifiant des indices entre parenthèses à la suite de son nom. L'élément situé la $i^{ième}$ ligne et la $j^{ième}$ colonne est obtenu par la commande `M(i,j)`. Par exemple, la valeur M_{23} est récupérée en tapant

```
>> M(2,3)
ans =
      6
```

On peut également modifier directement un des éléments en lui affectant une nouvelle valeur.

```
>> M(2,3)=13;
>> M
M =
     1     2     3
     4     5    13
     7     8     9
```

Matrices particulières

Quelques matrices particulières, et très utilisées, sont définies plus aisément au travers de fonctions. Ces fonctions prennent en argument les dimensions de la matrice que l'on souhaite construire. Le premier désigne le nombre de lignes et le second le nombre de colonnes.
La matrice nulle :

```
>> Z = zeros(2,3)
Z =
```

```
     0     0     0
     0     0     0
```

Une matrice pleine de 1 :

```
>> U = ones(4,3)
U =
     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

La matrice identité :

```
>> I  = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

Une matrice aléatoire (éléments compris entre 0 et 1) :

```
>> R = rand(2,2)
R =
    0.9575    0.1576
    0.9649    0.9706
```

Une matrice diagonale :

```
>> D = diag([2,4,0,7])
D =
     2     0     0     0
     0     4     0     0
     0     0     0     0
     0     0     0     7
```

Contrairement aux précédentes, cette dernière fonction prend en argument un vecteur. La taille de la matrice diagonale est donc déterminée par la taille du vecteur.

Extraction de sous-tableaux

Il est souvent utile d'extraire des blocs d'un tableau existant. Pour cela on utilise l'opérateur « : ». Pour cela, il faut spécifier pour chaque indice la valeur de début et la valeur de fin. La syntaxe générale est donc la suivante (pour un tableau à deux dimensions) : `tableau(début:fin, début:fin)`.

Ainsi pour extraire le bloc $\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$ de la matrice M, on tapera :

```
>> M(1:2,2:3)
ans =
     2     3
     5     6
```

Le caractère « : » seul, signifie toute la longueur est extraite. De cette façon, on peut isoler une ligne, ou une colonne, complète. Exemples :

```
>> M(1:2,:)
ans =
     1     2     3
     4     5     6
>> M(1,:)
ans =
```

```
     1     2     3
>> M(:,2)
ans =
     2
     5
     8
```

Construction de matrice par blocs

A partir de matrices et de vecteurs définis préalablement, on peut créer de nouvelles matrices :

$$N = \left[\begin{array}{c|c} M & V \\ \hline U & 0 \end{array}\right]$$

Cette opération est réalisée très simplement par la même syntaxe que pour les nombres.

```
>> N = [M V ; U 0];
```

Il est impératif que les matrices `M`, `V`, `U` et `O` aient été définies auparavant. De plus, les blocs composant une matrice doivent évidement être de dimension compatible. Si ces conditions ne sont pas respectées, la commande ne pourra s'exécuter et l'interface affichera une erreur.
Nous pouvons mentionner la fonction `blkdiag()` qui permet de créer une matrice diagonale à partir des éléments donnés en argument.

```
>> A = [1 1 ; 1 1];
>> B = [2 2 ; 2 2];
>> C = 3;
>> M = blkdiag(A,B,C)
M =
     1     1     0     0     0
     1     1     0     0     0
     0     0     2     2     0
     0     0     2     2     0
     0     0     0     0     3
```

4.4 Quelques fonctions utiles

Nous présentons dans ce paragraphe un ensemble de fonctions usuelles liées à l'utilisation des matrices.

« `size(M)` » renvoie les dimensions de la matrice.

« `max(M)` » renvoie un vecteur-ligne contenant les valeurs maximales associées à chaque colonne.

« `min(M)` » renvoie un vecteur-ligne contenant les valeurs minimales associées à chaque colonne.

« `rank(M)` » renvoie le rang de la matrice.

« `det(M)` » renvoie le déterminant de la matrice.

« `diag(M)` » extrait la diagonale de la matrice.

« `triu(M)` » extrait la matrice-triangle supérieure de M. `tril` donne la matrice-triangle inférieure.

« `eig(M)` » renvoie un vecteur contenant les valeurs propres de la matrice.

4.5 Opérations sur les matrices

Un des atouts remarquables de Matlab est la possibilité d'effectuer les opérations arithmétiques traditionnelles de façon naturelle sans avoir à les programmer. Les opérateurs standards sont donc directement applicables aux matrices. Si la commande entrée ne respecte pas les règles de calcul matriciel (compatibilité des opérandes), le logiciel renverra une erreur.

`+`	addition
`-`	soustraction
`*`	produit
`/`	division à droite
`\`	division à gauche
`^`	puissance
`'`	transposition
`inv()`	inversion

```
>> A = [2 1;6 9];
>> B = [1 0;-4 3];

>> A + B
ans =
     3     1
     2    12

>> A * B
ans =
    -2     3
   -30    27

>> A'
ans =
     2     6
     1     9

>> inv(B)
ans =
    1.0000         0
    1.3333    0.3333

>> A / B
ans =
    3.3333    0.3333
   18.0000    3.000

>> A^2
ans =
    10    11
    66    87
```

Si l'on souhaite effectuer une opération, non pas matricielle, mais éléments par éléments, l'opérateur doit être précédé d'un point « . » : `.*` `./` `.^` `.\`
Appliquons ces opérateurs aux matrices de l'exemple précédent.

```
>> A .* B
ans =
     2     0
   -24    27

>> B ./ A
ans =

    0.5000         0
   -0.6667    0.3333

>> A.^2
ans =
     4     1
    36    81
```

Les fonctions mathématiques vues au paragraphe II.3.3 traitent **Erreur ! Source du renvoi introuvable.**également les matrices. Dans ce cas, la fonction est appliquée à chaque élément. En outre, d'autres fonctions disponibles,

`expm` `logm` `sqrtm` `mpower`

réalisent quant à elles, des opérations matricielles. Par exemple, les fonctions « `exp` » et « `expm` » effectuent deux opérations tout à fait différentes :

Pour $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$

`exp(A)` $\rightarrow \begin{bmatrix} e^{a_{11}} & e^{a_{12}} \\ e^{a_{21}} & e^{a_{22}} \end{bmatrix}$ et `expm(A)` $\rightarrow e^{A}$

Le tableau suivant résume quelques uns des opérateurs évoqués précédemment.

Syntaxe Matlab	Ecriture mathématique	Composante ij
A	A	A_{ij}
B	B	B_{ij}
A+B	A+B	$A_{ij} + B_{ij}$
A-B	A-B	$A_{ij} - B_{ij}$
A.*B		$A_{ij} B_{ij}$
A.^B		$A_{ij}^{B_{ij}}$
A.^s		A_{ij}^{s}
A*B	AB	$\sum_k A_{ik} B_{kj}$
A/B	AB^{-1}	
A\B	$A^{-1}B$	
A'	A^{T}	A_{ji}

5 - Représentations graphiques

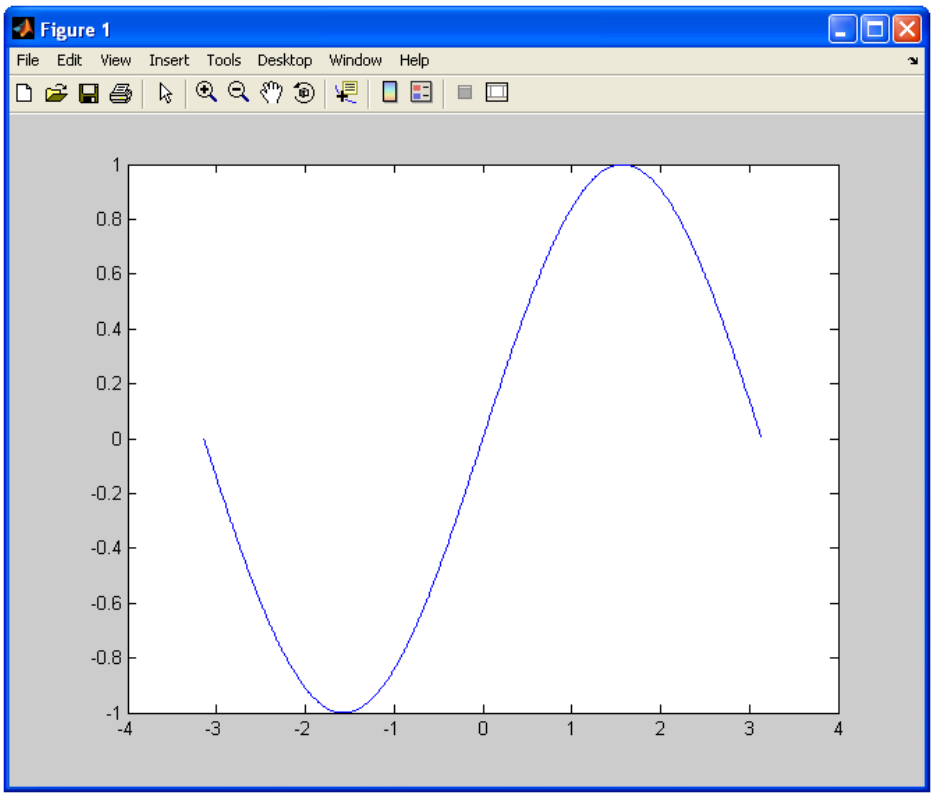
Les bibliothèques de Matlab proposent un très grand nombre de fonctions pour la manipulation d'objets graphiques. Nous ne présentons ici que quelques principes de base, utiles pour la visualisation de courbes. Si nous nous concentrons particulièrement sur la représentation graphique à 2 dimensions, il est possible d'aller bien plus loin : graphismes 3D (courbes, maillages, surfaces…), édition d'IHM (graphical user interface, GUI), animations… Quelques exemples de représentations 3D sont brièvement présentés.

5.1 Graphiques 2D

Comme nous l'avons vu dans l'exemple introductif du paragraphe II.2.1 le tracé d'une courbe s'effectue à partir de la commande `plot()`. Celle-ci prend en paramètres deux vecteurs et affiche sur un graphique à deux axes chaque couple de points (de même indice). Par exemple, `plot(x,y)` marquera un point pour chaque couple `[x(i),y(i)]` avec `i` allant de `0` à `length(x)`. On représente ainsi les valeurs de `y` en fonction des valeurs de `x`. La fonction renvoie une erreur si `x` et `y` ne sont pas de même longueur. Si le premier vecteur `x` est omis, `y` est tracé en fonction de son indice `i`. Par défaut, chaque point tracé est relié par une droite.

Traçons la fonction sinus dans l'intervalle [$-\pi$, π] avec un pas de 0.01.

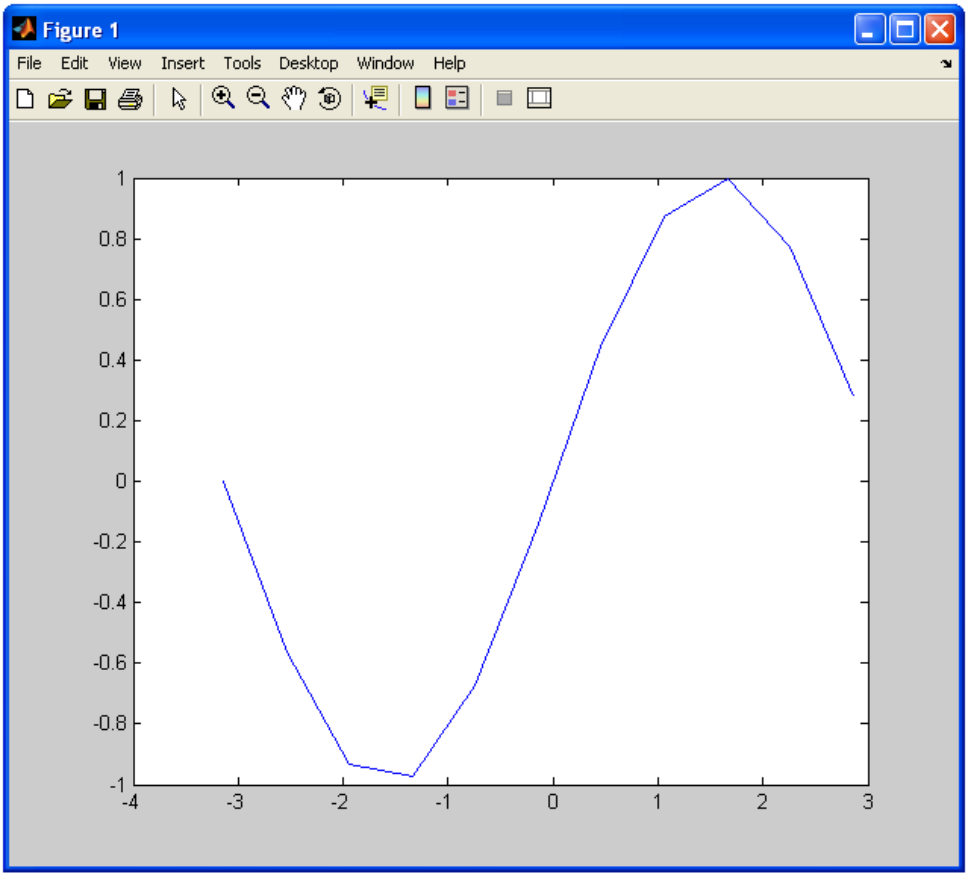
```
>> x = -pi : .01 : pi;
>> y = sin(x);
>> plot(x,y)
```



Le pas étant faible, la courbe semble parfaitement tracée. Bien évidemment, si l'on diminue le nombre de points (le pas est augmenté), la courbe apparaitra plus saccadée.

```
>> x = -pi : .6 : pi;
```

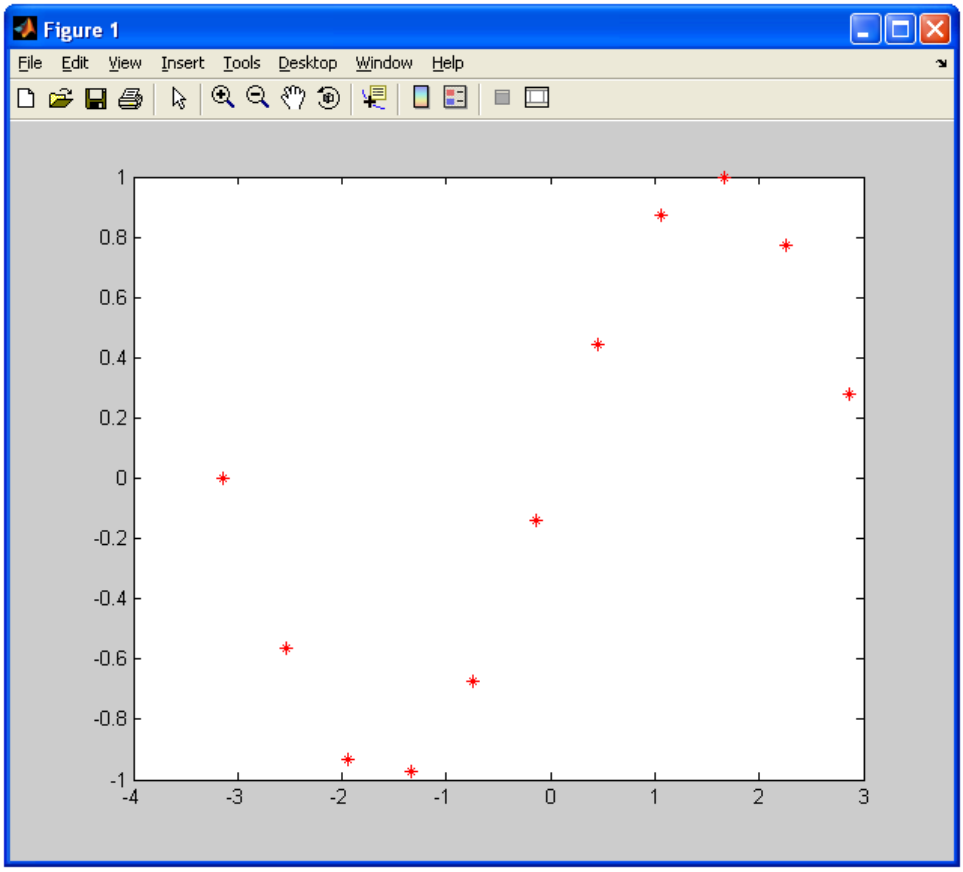
```
>> y = sin(x);
>> plot(x,y)
```



La commande `plot` prend un troisième argument permettant de spécifier la couleur du tracé et le symbole de représentation.

Retraçons l'exemple précédent en rouge avec des étoiles pour chaque point.

```
>> x = -pi : .6 : pi;
>> y = sin(x);
>> plot(x,y,'r*')
```



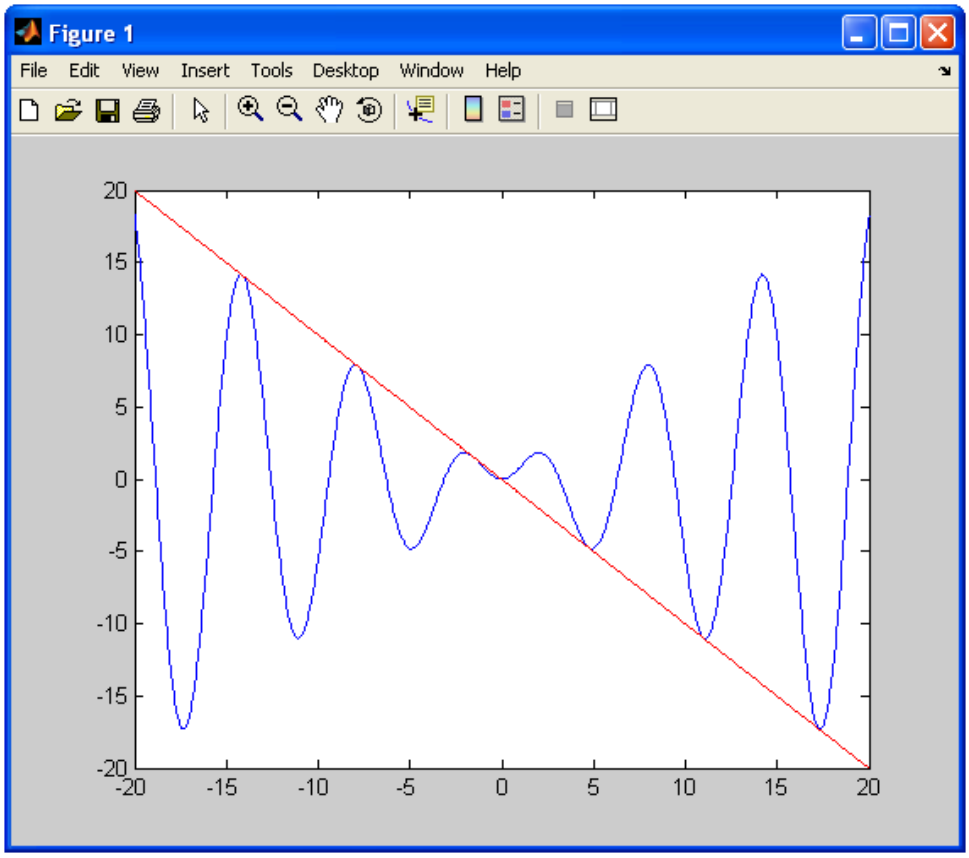
Différentes options sont disponibles (consulter le `help plot`) :

```
Various line types, plot symbols and colors may be obtained with
PLOT(X,Y,S) where S is a character string made from one element
from any or all the following 3 columns:

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

A chaque nouvelle commande `plot`, la figure est remplacée. Pour garder plusieurs courbes, il faut autoriser la superposition de graphique à l'aide de la commande `hold on`. Les `plot` suivants se superposeront jusqu'à la désactivation `hold off` ou la fermeture de la fenêtre.

```
>> x = linspace(-20,20,1000);
>> y = x.*sin(x);
>> plot(x,y)
>> hold on
>> y2 = -x;
```

```
>> plot(x,y2,'r')
```



Il est également possible de tracer plusieurs courbes sur plusieurs fenêtres. Pour cela, une nouvelle fenêtre (objet graphique `figure`) doit invoquée avant l'appel à la fonction `plot` correspondante.

```
>> plot(x,y)
>> figure(2)
>> plot(x,y2,'r')
```

D'autres fonctions permettent une représentation différente des données, par exemples sous forme discrète (`stem`), d'histogramme (`bar`), de camembert (`pie`), d'escalier (`stairs`), avec échelle logarithmique (`semilogx`, `semilogy`)…

La mise en forme d'une représentation graphique, c'est-à-dire l'insertion de labels, légende, le dimensionnement des axes, peut être éditée de deux manières. La méthode la plus simple utilise directement les menus de l'interface de la figure (`Edit` et `Insert`). Toutes ces manipulations sont également réalisables à partir de la `Command Window` en ligne de commande. Cette seconde méthode est généralement utilisée lors du développement de programmes. Voici quelques exemples parmi les manipulations les plus simples :

```
>> xlabel('valeur x')
>> ylabel('valeur y')
>> title('mon graphique')
>> legend('ma courbe')
>> grid on
>> axis([xmin xmax ymin ymax])
```

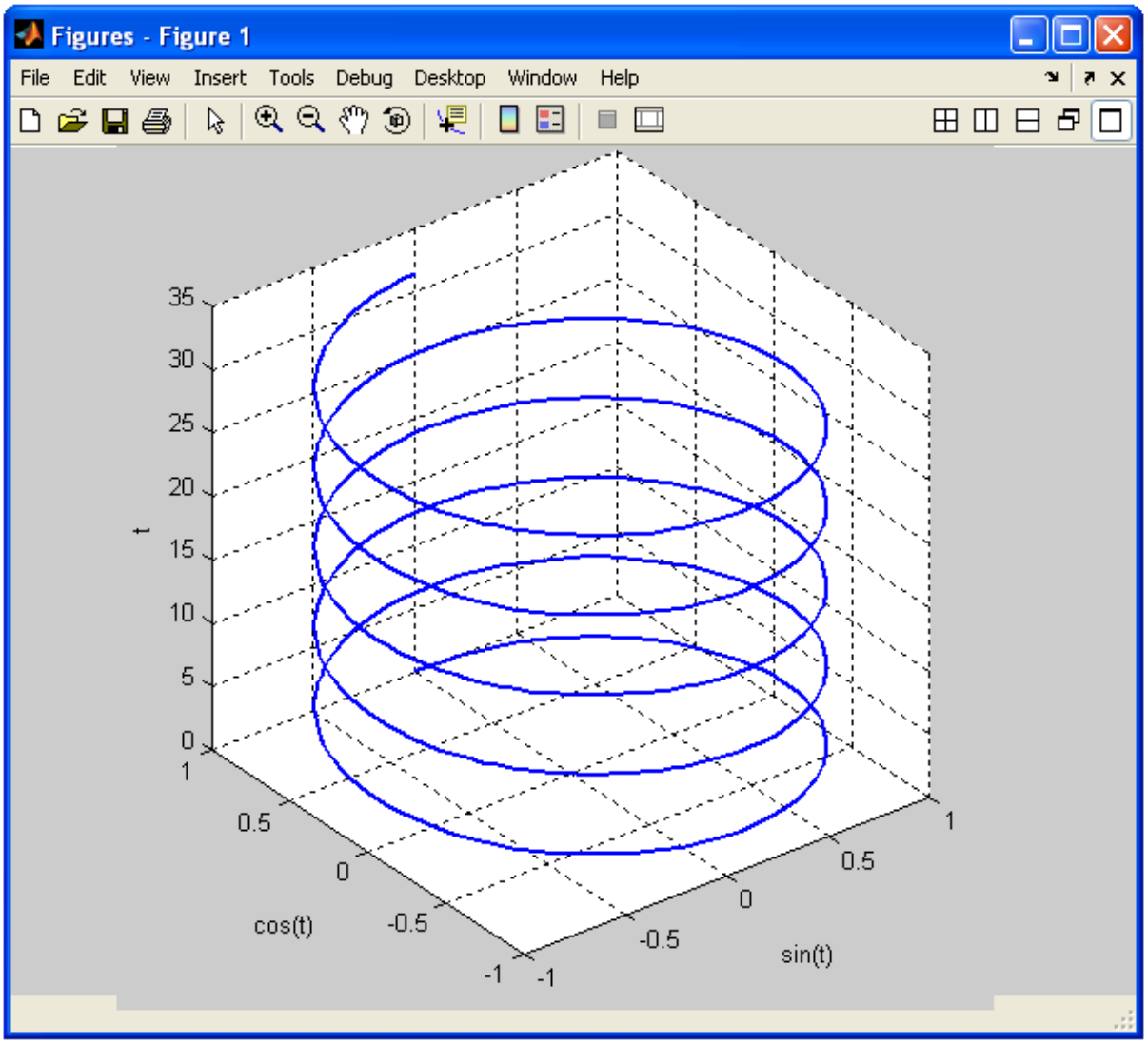
5.2 Graphiques 3D

Nous montrons ici les possibilités de Matlab en graphisme 3D sur quelques exemples.

Tracé de courbes dans l'espace

```
>> t = 0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t)
>> grid on
>> axis square
>> xlabel('sin(t)'), ylabel('cos(t)'), zlabel('t')
```

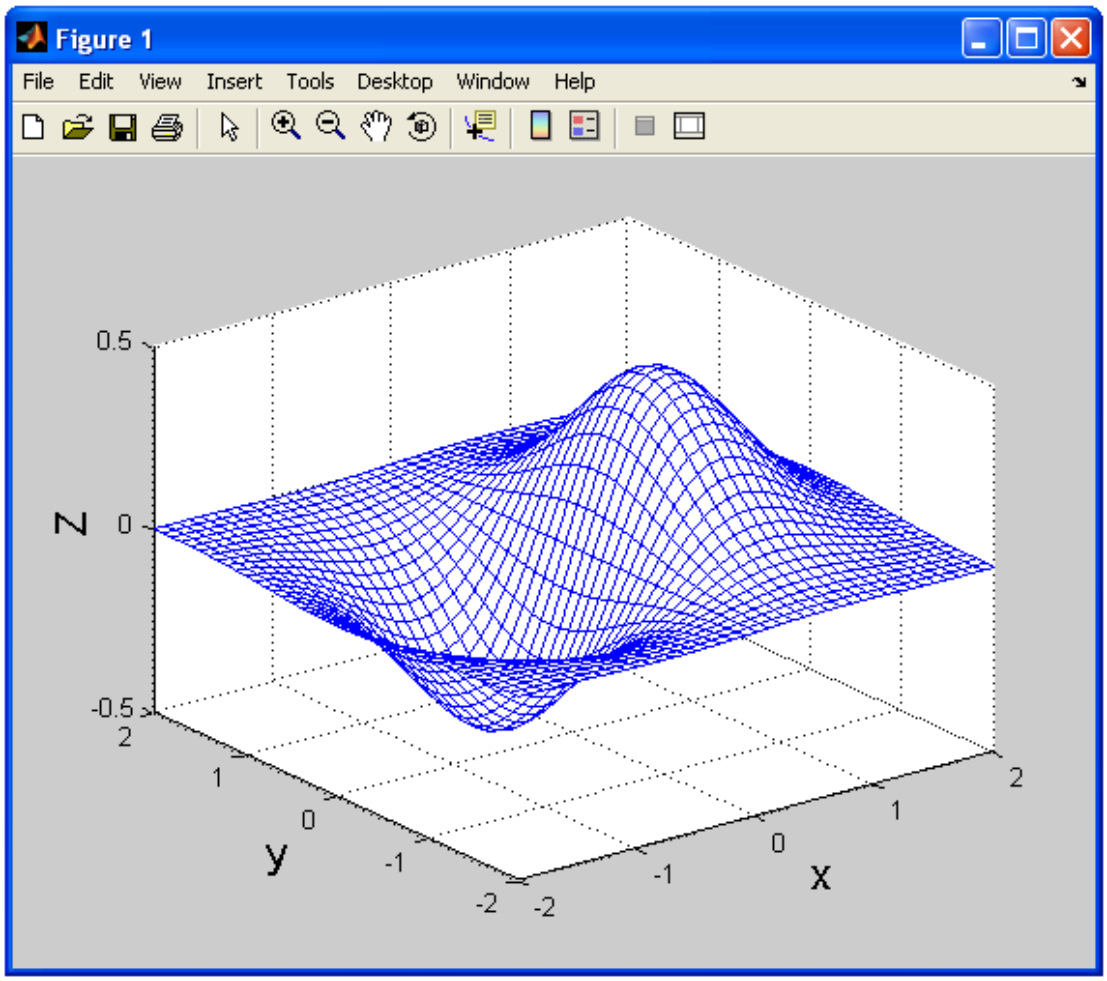
La fonction prend en argument 3 vecteurs de même taille. Son fonctionnement est similaire à celui de `plot`. Elle affiche dans un système d'axe à 3 dimensions les triplets `[x(i),y(i),z(i)]`.

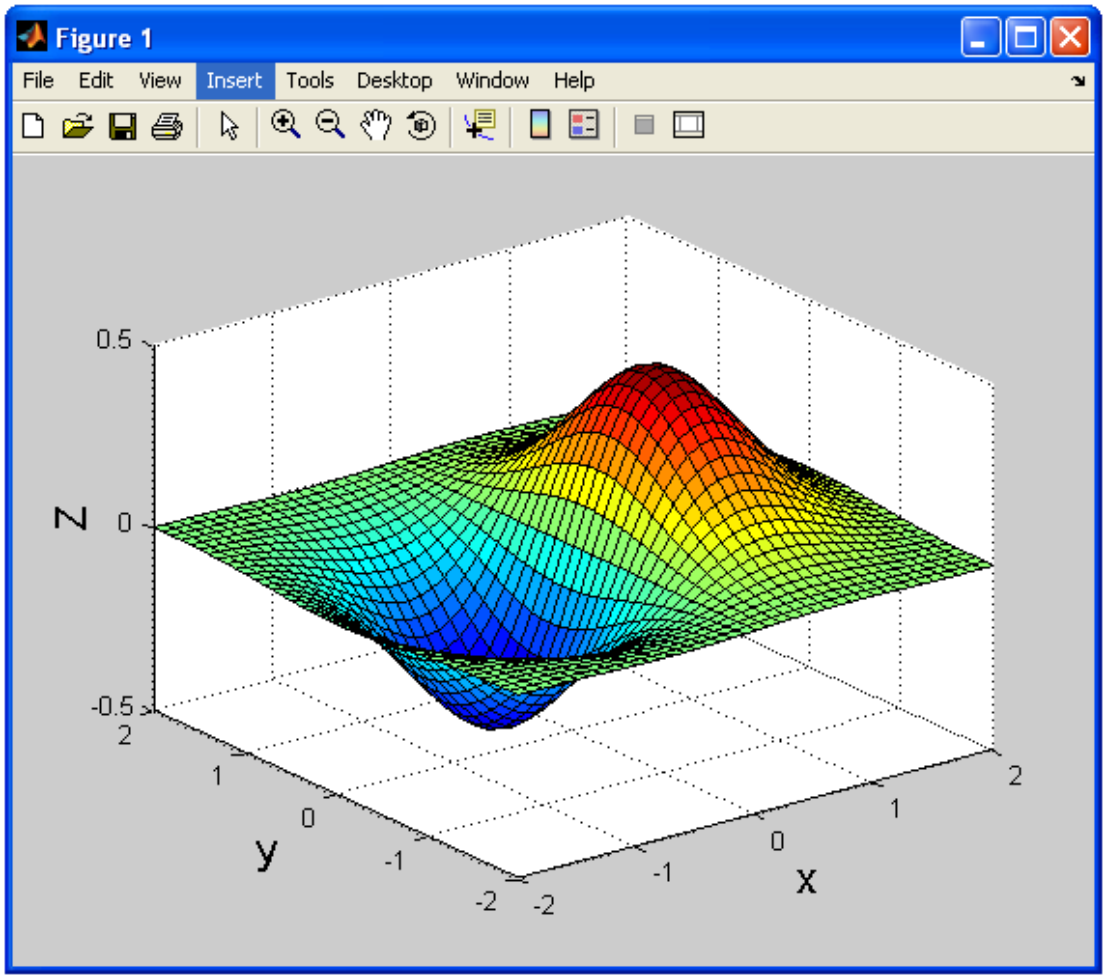


Représentation par maillage dans le plan (x,y)

```
>> x=[-2:0.1:2];
>> y=[-2:0.1:2];
>> [X,Y]=meshgrid(x,y);
>> colormap([0 0 1]);
>> Z=X.*exp(-X.^2-Y.^2);
>> mesh(X,Y,Z)
>> colormap('default') ;
>> surf(X,Y,Z)
>> xlabel('x'), ylabel('y'), zlabel('Z')
```

La fonction `mesh` trace un maillage (séries de lignes entre les points) tandis que la fonction `surf` trace une surface. Ces fonctions prennent en argument 2 matrices générées à partir de 2 vecteurs (`x` et `y`, pas nécessairement de même taille) et une matrice `Z` de dimension `(length(y),length(x))`. Pour chaque point du plan `(x(i),y(j))`, elles affichent la valeur (ou niveau) `Z(j,i)`. Les 2 premières matrices obtenues à l'aide de la fonction `meshgrid` définissent tous les points du quadrillage. En fait, le niveau `Z(j,i)` de chaque point du maillage est calculé à partir des couples `(X(j,i),Y(j,i))`. Une quatrième matrice peut être précisée pour définir une échelle de couleur associée au niveau du maillage. Cette échelle est basée sur une palette de couleurs prédéfinie par la valeur courante de la variable `colormap`.



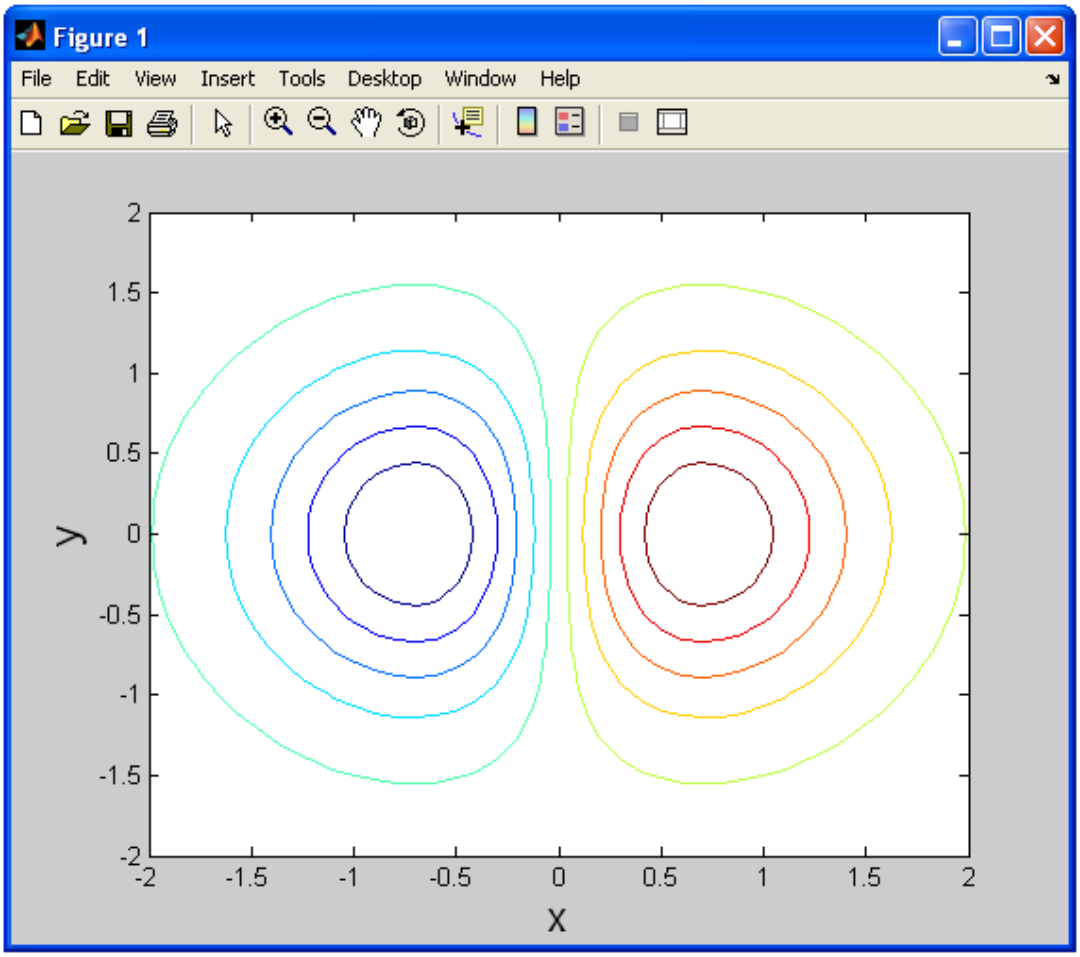


Tracé des courbes de contour

Reprenons l'exemple précédent.

```
>> x=[-2:0.1:2];
>> y=[-2:0.1:2];
>> [X,Y]=meshgrid(x,y);
>> Z=X.*exp(-X.^2-Y.^2);
>> contour(X,Y,Z,10)
>> xlabel('x'), ylabel('y')
```

La fonction `contour` trace dans le plan `(x,y)` les courbes `z=cst` d'une surface. Elle s'utilise comme les deux fonctions précédentes mais représente les courbes sur un graphique 2D avec un dégradé de couleurs associé aux valeurs de `z` correspondantes. Le quatrième paramètre représente le nombre de lignes de niveau à tracer. On peut également spécifier quelles lignes de niveau afficher. Par exemple, pour dessiner les courbes `z=-0.1,0,0.3`, on écrira `contour(X,Y,Z,[-0.1 0 0.3])`.



6 - Programmation avec Matlab

Dans les précédentes sections, nous avons présenté des séries de commandes lancées depuis la `command window`. Pour des calculs complexes et répétitifs, il est préférable (ou plutôt indispensable) de rassembler l'ensemble des commandes dans un fichier qui constituera le programme à exécuter. On distingue deux types de fichiers dans Matlab, également appelés `m-files` : les scripts et les fonctions. Bien que l'environnement de Matlab propose son propre éditeur (fenêtre `Editor`), ces fichiers sont de simples fichiers textes avec une extension `.m`. Vous pouvez donc utiliser votre éditeur de texte préféré pour créer vos programmes (sans oublier de modifier l'extension). A partir de Matlab, un `m-file` est créé ou ouvert, soit depuis le menu `Fichier` (`New > M-File`), soit depuis l'invite en tapant :

```
>> edit monfichier.m
```

Un `m-file` est reconnu, et donc exécutable, s'il se trouve dans le répertoire courant (`current directory`) ou si le répertoire contenant est spécifié dans le `PATH`.

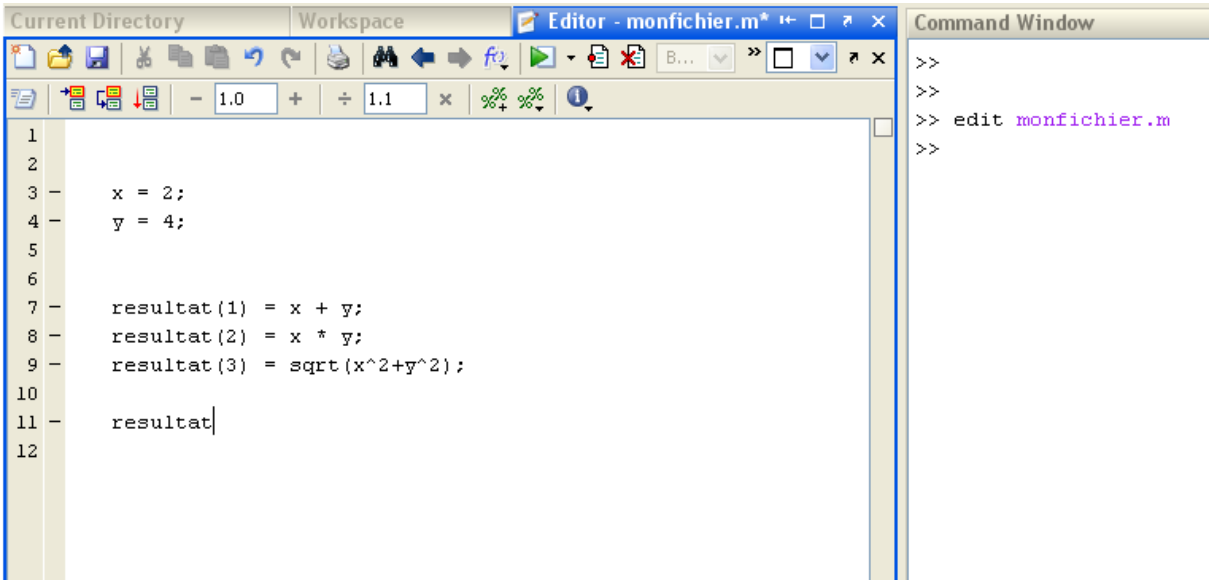
Nous allons voir que Matlab offre la possibilité de réaliser de véritables applications très élaborées. Notons qu'il utilise un langage de programmation interprété, c'est-à-dire qu'il n'y a aucune phase de compilation et les instructions du code sont directement exécutées à leur lecture.

6.1 Fichiers SCRIPT

Un fichier `script` permet regrouper des séries de commandes Matlab. Cela évite d'avoir à saisir plusieurs fois de longues suites d'instructions. A son lancement, les instructions qu'il contient s'exécutent séquentiellement comme si elles étaient lancées depuis l'invite de commande. Un script stocke ses variables dans le `workspace`, lequel est partagé par tous les scripts. Ainsi, toutes les variables créées dans les scripts sont visibles depuis la `command window` et vice versa. Lorsque Matlab détecte une erreur, le programme s'arrête et un message d'erreur s'affiche à l'écran (avec le numéro de la ligne où l'erreur est détectée).

Editons notre script `monfichier.m`.

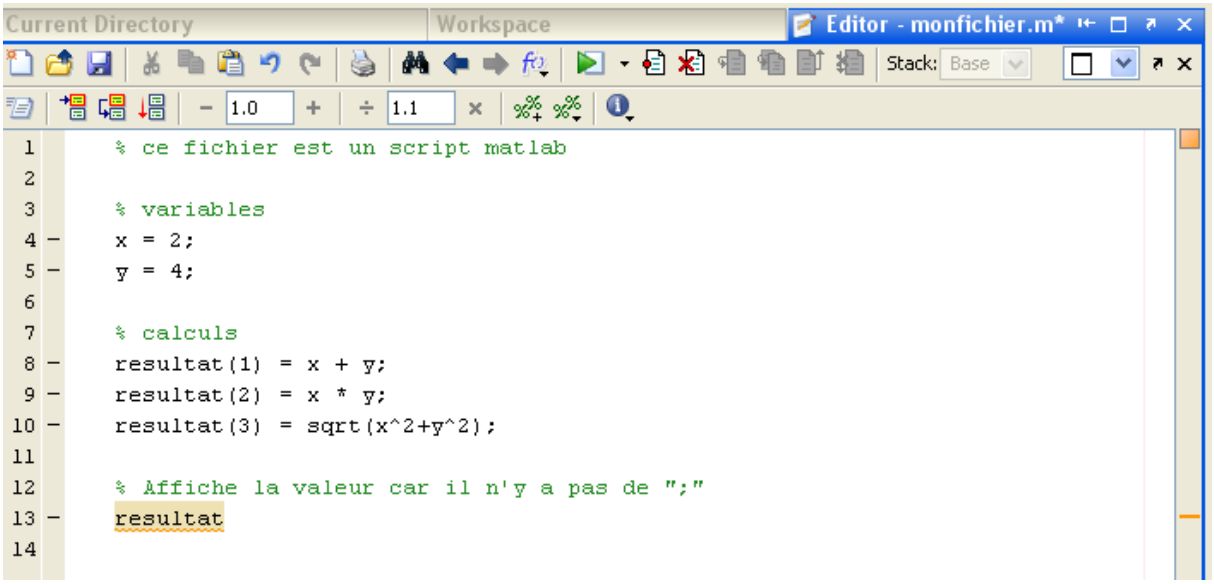
```
x = 2;
y = 4;


resultat(1) = x + y ;
resultat(2) = x * y ;
resultat(3) = sqrt(x^2+y^2);

resultat
```



Notre script peut ensuite être exécuté, soit en tapant son nom (sans l'extension) à l'invite de commande, soit en cliquant sur le bouton « run » de l'éditeur (icône avec un triangle vert).

```
>> monfichier

resultat =

    6.0000    8.0000    4.4721
```

Des annotations peuvent être ajoutées dans le code afin de le commenter. Pour cela, chaque ligne de commentaires doit être précédée par le caractère `%`. Les mots suivant ce symbole ne seront pas interprétés.



De manière générale, il est essentiel d'inclure dans le code un nombre conséquent de commentaires. Ils permettent de documenter un programme et facilitent la relecture, la maintenance de celui-ci.

6.2 Fichiers *FUNCTION*

Le principe d'une fonction est d'effectuer des opérations à partir d'une ou plusieurs entrées et fournir une ou plusieurs sorties (résultat). Les variables d'entrées sont des paramètres à spécifier en argument de la fonction, tandis que les variables de sorties sont des valeurs quelle renvoie. Un m-file function est tout à fait semblable aux fonctions intégrées de Matlab. Par exemple, la fonction `length` renvoie la taille du tableau entré en argument.

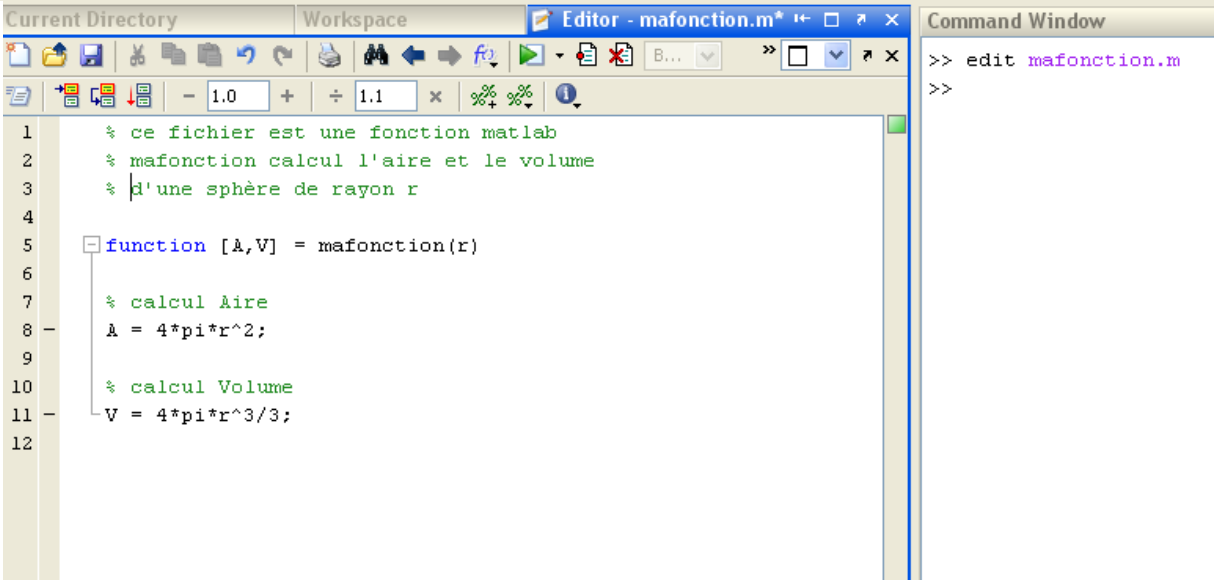
```
>> taille = length(tab);
```

Un m-file est définit comme une fonction en plaçant en tête du fichier le mot clé `function` suivit de son prototype. Un prototype est de la forme

```
[s1,s2,…] = nomfonction(e1,e2,…)
```

Le membre de gauche regroupe les sorties renvoyées par la fonction et les variables entres parenthèses sont les entrées. Le nom du fichier doit être identique au nom de la fonction. Ecrivons dans l'éditeur notre fonction qui permet de calculer l'aire et le volume d'une sphère pour un rayon donné.

```
% ce fichier est une fonction matlab
% mafonction calcul l'aire et le volume d'une sphère de rayon r

function [A,V] = mafonction(r)

% calcul Aire
A = 4*pi*r^2;

% calcul Volume
V = 4*pi*r^3/3;
```



L'appel de la fonction s'effectue de la même façon que pour les fonctions prédéfinies dans le logiciel

```
>> [aire,volume] = mafonction(2);
>> aire

aire =

   50.2655

>> volume

volume =

   33.5103
```

Le point fondamental qui différencie une fonction d'un script est le fait que les variables internes soient locales, c'est-à-dire que les variables définies dans une fonction n'existent que dans celle-ci. De plus, les variables du `workspace` ne sont pas visibles depuis une fonction. Ainsi, dans notre exemple, les paramètres `A`, `V` et `r` ne sont pas connues dans le `workspace`.

```
>>  whos
  Name         Size              Bytes  Class     Attributes

  aire         1x1                   8  double
  volume       1x1                   8  double
```

Pour pouvoir utiliser une variable partagée par le `workspace` et une (voire des) fonction(s), celle-ci doit être déclarée comme `global` à la fois dans la `command window` et dans la (les) fonction(s). Ajoutons à notre fonction

```
global x;
x = r;
```

et écrivons les lignes suivantes dans la `command window`

```
>> global x;
>> x = 10;
```

```
>> [aire,volume]=mafonction(3);
>> x

x =

     3
```

On constate donc que la variable globale `x`, a été modifiée lors de l'appel de la fonction.

Notons toutefois que l'utilisation de variables globales est déconseillée, car souvent source d'erreurs d'exécution, et doit donc être minimisée.

Des instructions permettent de contrôler les arguments d'entrées et de sorties d'une fonction:

`nargin`	retourne le nombre d'arguments d'entrée
`nargout`	retourne le nombre d'arguments de sortie
`nargchk`	vérifie le nombre d'arguments d'entrée
`inputname`	retourne le nom d'un argument d'entrée

6.3 Opérateurs relationnels et logiques

Comme dans tout langage de programmation, Matlab possède des opérateurs qui permettent d'établir des expressions renvoyant en résultat une valeur logique, c'est-à-dire 0 ou 1. Ces expressions logiques sont généralement utilisées dans les structures de contrôle présentées dans la prochaine section.

Opérateurs relationnels

Ces opérateurs comparent deux opérandes de même dimension :

`==`	égal à
`~=`	différent de
`>`	strictement supérieur à
`>=`	supérieur ou égal à
`<`	strictement inférieur à
`<=`	inférieur ou égal à

Lorsque deux scalaires sont comparés, le résultat est un scalaire qui vaut 1 si la relation est vrai et 0 si elle est fausse. Si deux matrices sont comparées, le résultat est une matrice de même dimension constituée de 1 et 0, la relation étant testée élément par élément.

```
>> 10 > 9
ans =
     1

>> 2 == 3
ans =
     0

>> 4 ~= 7
ans =
     1

>> [1 4 ; 7 3] <= [0 6 ; 7 2]
ans =

     0     1
     1     0
```

Opérateurs logiques

Ces opérateurs effectuent un test logique entre deux variables logiques de même dimension:

`&`	et
`|`	ou
`~`	non
`xor`	ou exclusif
`any(x)`	retourne 1 si un des éléments de `x` est non nul
`all(x)`	retourne 1 si tous les éléments de `x` sont nuls

Le résultat vaut 1 si le test est vrai et 0 s'il est faux. Pour des matrices, l'opération s'effectue aussi élément par élément. Concernant les opérandes, une valeur est considérée comme fausse (=0) si elle est nulle. Elle est considérée comme vrai (=1) si elle est non nulle.

```
>> x = [0 1 0 1];
>> y = [0 0 1 1];
>> x & y
ans =
     0     0     0     1

>> x | y
ans =
     0     1     1     1

>> ~x
ans =
     1     0     1     0

>> xor(x,y)
ans =
     0     1     1     0

>> 0 | 3
ans =
     1

>> ~(-2.4)
ans =
     0
```

6.4 Structures de contrôle

Dans sa forme la plus simple, le déroulement d'un programme est linéaire dans le sens où les instructions qui le composent s'exécutent successivement. Les structures de contrôle sont des mécanismes qui permettent de modifier la séquence d'exécution des instructions. Plus précisément, lors de l'exécution, en fonction des conditions réalisées certaines parties précises du code seront exécutées.

Branchement conditionnel (`if … elseif … else`)

Cette structure permet d'exécuter un bloc d'instructions en fonction de la valeur logique d'une expression. Sa syntaxe est :

```
if expression
      instructions …
end
```

L'ensemble des instructions *instructions* est exécuté seulement si *expression* est vraie. Plusieurs tests exclusifs peuvent être combinés.

```
if expression1
      instructions1 …
elseif expression2
      instructions2 …
else
      instructions3 …
end
```

Plusieurs `elseif` peuvent être concaténés. Leur bloc est exécuté si l'expression correspondante est vraie et si toutes les conditions précédentes n'ont pas été satisfaites. Le bloc *instruction3* associé au `else` est quant à lui exécuté si aucune des conditions précédentes n'a été réalisées.

```
if x > 0
     disp('x est positif');

elseif x == 0
     disp('x est nul');

else
     x = 1;

end
```

Bien évidement, la variable `x` doit être définie auparavant. La fonction `disp` permet d'afficher une chaîne de caractère spécifiée entre apostrophes. Si `x` n'est ni positif ni nul, il reçoit la valeur `1`.

Branchement multiple (switch … case)

Dans cette structure, une expression numérique est comparée successivement à différentes valeurs. Dès qu'il y a identité, le bloc d'instructions correspondant est exécuté. Sa syntaxe est :

```
  switch expression
      case valeur1,
          instructions1 …
      case valeur2,
          instructions2 …
      case valeur3,
          instructions3 …
      ……
      otherwise
          instructions …
  end
```

L'expression testée, *expression*, doit être un scalaire ou une chaîne de caractère. Une fois qu'un bloc *instructionsi* est exécuté, le flux d'exécution sort de la structure et reprend après le `end`. Si aucun `case` vérifie l'égalité, le bloc qui suit `otherwise` est exécuté.

```
switch x
     case 0,
         resultat = a + b;
     case 1,
         resultat = a * b;
     case 2,
         resultat = a/b;
```

```
    case 3,
        resultat = a^b;
    otherwise
        resultat = 0;
end
```

En fonction de la valeur de `x` une opération particulière est effectuée. Par défaut, `resultat` prend la valeur `0`.

Boucle conditionnelle (while …)

Ce mécanisme permet de répéter une série d'instructions tant qu'une condition est vérifiée. Sa syntaxe est :

```
while expression

    instructions …
end
```

Le terme *expression* est une expression logique. Si cette dernière est vraie, le bloc *instructions* est exécuté. Puis, *expression* est de nouveau testé. L'exécution du bloc est répétée tant que le test est vrai.

```
compteur = 0;
while compteur < 10

        disp('toujours dans la boucle') ;
        compteur = compteur + 1;
end
```

Cet exemple affiche 10 fois la chaîne de caractère `toujours dans la boucle`.

Boucle itérative (for …)

Cette boucle exécute le bloc interne autant de fois que spécifié par une variable jouant un rôle de compteur. Sa syntaxe est :

```
for variable = debut:increment:fin

    instructions …
end
```

Le compteur *variable* est initialisé à la valeur *debut* et évolue jusqu'à la valeur *fin* par pas de *increment*. A chaque itération, le bloc *instructions* est exécuté. Généralement, *variable* est un scalaire, et souvent un entier.

```
N = 5 ;
for k = 1:N

      x(k)= 1/k;
end
```

Cet exemple construit élément par élément un vecteur `x` de dimension 5.

III. APPLICATION A LA MECANIQUE

1 - Les structures

On a vu jusqu'à maintenant que MATLAB permet de manipuler facilement des matrices. Mais, il permet également de manipuler un autre type de structure de données appelée les structures, qui permettent notamment de stocker des éléments de différents types et de différentes tailles sous une même variable.

L'avantage principal réside dans la capacité d'une structure à regrouper, sous un seul nom, l'information qui serait à défaut dispersée. Par exemple, si on doit simuler la chute d'une balle, il faut des variables pour représenter sa masse, un vecteur vitesse et un vecteur position. Au lieu de définir une variable par propriété, on peut regrouper ces propriétés dans une structure appelée balle

```
>> clear
>> balle.masse = 10;
>> balle.position = [0, 0, 100];
>> balle.vitesse = [0, 0, 0];
```

Pour voir le contenu de la structure, il faut juste taper son nom à l'invite de commande

```
>> balle
balle =

       masse: 10
    position: [0 0 100]
     vitesse: [0 0 0]
```

Chaque constituant de la structure est appelé un champ. On peut également avoir le contenu d'un champ en tapant :

```
>>balle.position
ans =

     0     0   100
```

On peut faire les opérations sur les champs d'une structure comme sur n'importe quelle variable. Pour modifier la position de la balle de 10 suivant l'axe y :

```
>> balle.position = balle.position + [0, 10, 0]
balle =
```

```
        masse: 10
     position: [0 10 100]
      vitesse: [0 0 0]
```

On peut facilement ajouter un nouveau champ à une structure existante :

```
>> balle.rayon = 2.0
>> balle =

       masse: 10
    position: [0 10 100]
     vitesse: [0 0 0]
       rayon: 2
```

IV. APPLICATION A L'AUTOMATIQUE

Matlab est le logiciel de prédilection des automaticiens, il possède de nombreuses fonctionnalités spécifiques à l'Automatique. Les fonctions les plus standards sont regroupées dans la Control System Toolbox. Ce chapitre présente, sur divers exemples, l'utilisation de Matlab pour l'Automatique. Dans un second temps nous parlerons d'un second logiciel, Simulink®, intégré à Matlab, lui aussi très utilisé par la communauté des automaticiens. Il s'agit d'un environnement graphique pour la modélisation et la simulation de systèmes multi-domaine.

1 - Représentation des systèmes linéaires invariants

1.1 Fonctions de transfert

La Control System Toolbox permet de définir des objets de type `tf`, c'est-à-dire transfert function. La fonction du même nom, `tf()`, crée une fonction de transfert particulière. Elle prend en argument deux vecteurs contenant les coefficients des polynômes du numérateur et du dénominateur. Ainsi, une fonction de transfert de la forme :

$$G(p) = \frac{b_m p^m + b_{m-1} p^{m-1} + \cdots + b_1 p + b_0}{a_n p^n + a_{n-1} p^{n-1} + \cdots + a_1 p + a_0}$$

sera définie par la commande `tf(num,den)`

avec $\begin{cases} num = [\, b_m \,|\, b_{m-1} \,|\, \ldots \,|\, b_1 \,|\, b_0 \,] \\ den = [\, a_n \,|\, a_{n-1} \,|\, \ldots \,|\, a_1 \,|\, a_0 \,] \end{cases}$

```
>> G = tf([1 2],[1 3 2])

Transfer function:
    s + 2
-------------
s^2 + 3 s + 2
```

```
>> H = tf([1 -1],[3 1 0])
 transfer function:
  s - 1
```

```
---------
3 s^2 + s
```

Dans Matlab, et plus généralement dans la littérature anglo-saxonne, la variable de Laplace `p` est notée `s`.

1.2 Représentation d'état

Dans la cas d'une modélisation dans l'espace d'état, le type d'objet utilisé est le type `ss` (state-space). La fonction associée, `ss()`, prend en argument les 4 matrices définissant le système. Soit le système linéaire invariant suivant :

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

où A, B, C et D sont, respectivement, les matrices dynamique, de commande, de sortie et de transfert direct. Ce modèle sera défini par la commande : `ss(A,B,C,D)`
Bien évidemment, les matrices doivent être définies au préalable. Attention à la compatibilité des dimensions.

```
>> A=[-1 4 ; 0 -2];
>> B=[2;3];
>> C=[1 0];
>> D=0;
>> sys = ss(A,B,C,D)

a =
       x1  x2
   x1  -1   4
   x2   0  -2

b =
       u1
   x1   2
   x2   3

c =
       x1  x2
   y1   1   0

d =
       u1
   y1   0

Continuous-time model.
```

Les différentes matrices d'un objet de type `ss` peuvent être récupérées par les commandes `sys.a`, `sys.b`, `sys.c` et `sys.d`.

```
>> sys.a

ans =
    -1     4
     0    -2
```

1.3 Systèmes discrets et échantillonnés

La définition de fonctions de transfert en Z et de modèles d'état discrets suit la même notation que dans le cas des systèmes continus. Dans les deux cas un argument supplémentaire doit être spécifié : la période d'échantillonnage.

Exemple d'une fonction de transfert en Z avec une période d'échantillonnage de 200ms:

```
>> Gd = tf([1 1],[1 0.5 0.25],0.2)

Transfer function:
      z + 1
------------------
z^2 + 0.5 z + 0.25

Sampling time: 0.2
```

Une fonction de transfert en Z peut également être définie en z^{-1} en précisant explicitement la forme de la variable.

```
>> Gd2 = tf([7 -1],[1 4 3],0.2,'variable','z^-1')

Transfer function:
     7 - z^-1
-------------------
1 + 4 z^-1 + 3 z^-2

Sampling time: 0.2
```

Exemple d'une représentation d'état d'un système discret avec une période de 1s:

```
>> A=[0.1 3 ; 0 -0.5];
>> B=[0;1];
>> C=[1 1];
>> D=0;
>> sysd = ss(A,B,C,D,1)

a =
         x1    x2
   x1   0.1     3
   x2     0  -0.5

b =
       u1
   x1   0
   x2   1

c =
       x1  x2
   y1   1   1

d =
       u1
   y1   0

Sampling time: 1
Discrete-time model.
```

La valeur particulière -1 en argument indique que la période d'échantillonnage est indéterminée. Les commandes `Gd.Ts` et `sysd.Ts` renvoient la période du système discret correspondant.

1.4 Conversions de modèles

Lors de l'étude d'un système, il est souvent pratique de passer d'une représentation à une autre en fonction des objectifs. En effet, les fonctions de transfert et la représentation d'état sont deux formalismes qui possèdent leurs propres outils d'analyse et méthodes de synthèse de lois de commande. Par ailleurs, la Control System Toolbox fournit aussi des fonctions pour convertir un modèle à temps continu en un modèle à temps discret. Ces fonctions sont très utiles pour l'échantillonnage et la discrétisation de systèmes continus.

Premièrement, la conversion de la classe d'un modèle (fonction de transfert ou espace d'état) s'effectue aisément en appliquant directement la fonction correspondante à l'objet à convertir (`tf` ou `ss`). Notons la commande `class` qui permet, entre autres, de retourner le type de l'objet donné en argument.

fonction de transfert → représentation d'état

```
>> G = tf([1 2],[1 3 2]);
>> class(G)
ans =

tf
>> ss(G)

a =
       x1  x2
   x1  -3  -2
   x2   1   0

b =
       u1
   x1   2
   x2   0

c =
        x1   x2
   y1  0.5    1

d =
       u1
   y1   0

Continuous-time model.
```

représentation d'état → fonction de transfert

```
>> A=[-1 4 ; 0 -2];
>> B=[2;3];
>> C=[1 0];
>> D=0;
>> sys = ss(A,B,C,D);
>> class(sys)
ans =

ss
>> G = tf(sys)

Transfer function:
  2 s + 16
-------------
s^2 + 3 s + 2
```

Les représentations d'état n'étant pas uniques il est possible d'opérer des changements de bases. La fonction `ss2ss` réalise, en précisant une matrice de passage, un changement de coordonnées de l'état. La fonction `canon` transforme un modèle dans l'espace d'état sous une forme canonique (forme modale ou compagne).

Le calcul de modèles échantillonnés ou la discrétisation de systèmes continus s'effectuent à l'aide de la fonction `cd2`, c'est-à-dire « continuous-to-discrete ». Elle nécessite trois arguments : le modèle du système continu à transformer (`tf` ou `ss`), la période d'échantillonnage et la méthode de conversion. Deux méthodes sont principalement utilisées : `zoh` et `tustin`. La première donne un système numérique obtenu par échantillonnage tandis que la seconde un système numérique obtenu pas discrétisation via la transformation de Tustin (ou la méthode des trapèzes, ou approximation bilinéaire).

système continu → système échantillonné

```
>> G = tf([1 2],[1 3 2]);
>> Ge = c2d(G,0.5,'zoh')

Transfer function:
   0.3935 z - 0.1447
-----------------------
z^2 - 0.9744 z + 0.2231

Sampling time: 0.5
```

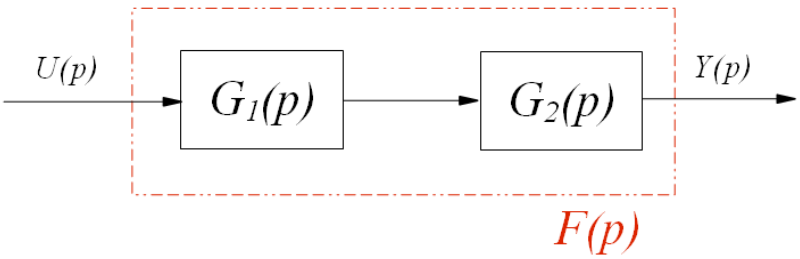
système continu → système discret approché par la méthode de Tustin

```
>> G = tf([1 2],[1 3 2]);
>> Gd=c2d(G,0.5,'tustin')

Transfer function:
0.2 z^2 + 0.1333 z - 0.06667
----------------------------
    z^2 - 0.9333 z + 0.2

Sampling time: 0.5
```

1.5 Connexions de systèmes

Des systèmes complexes peuvent être construits à partir de sous-systèmes élémentaires. Voici les trois opérations de base pour l'interconnexion de modèles.

Mise en série de deux systèmes :

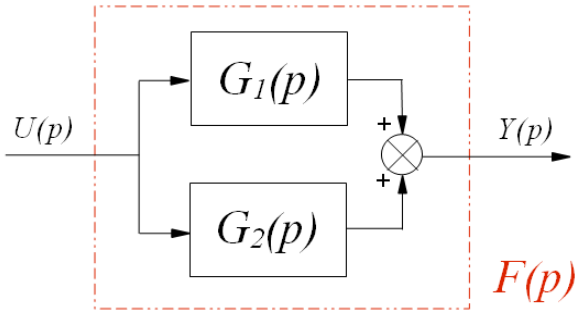



```
>> F = series(G1,G2);
```

ou bien

```
>> F = G1 * G2;
```

Mise en parallèle de deux systèmes :

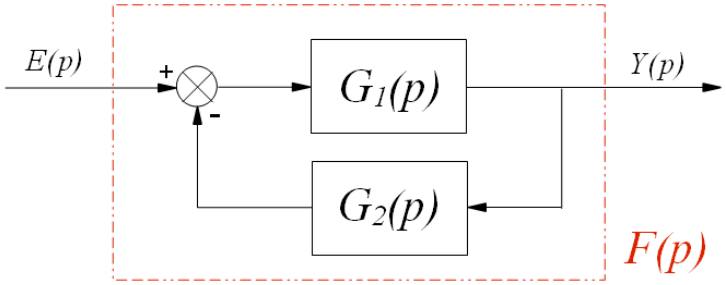



```
>> F = parallel(G1,G2);
```

ou bien

```
>> F = G1 + G2;
```

Bouclage d'un système :




```
>> F = feedback(G1,G2);
```

2 - Analyse des systèmes dynamiques

2.1 Quelques fonctions utiles

Bien qu'il soit possible de calculer les caractéristiques d'un système à partir des fonctions standards, il existe des fonctions dédiées.

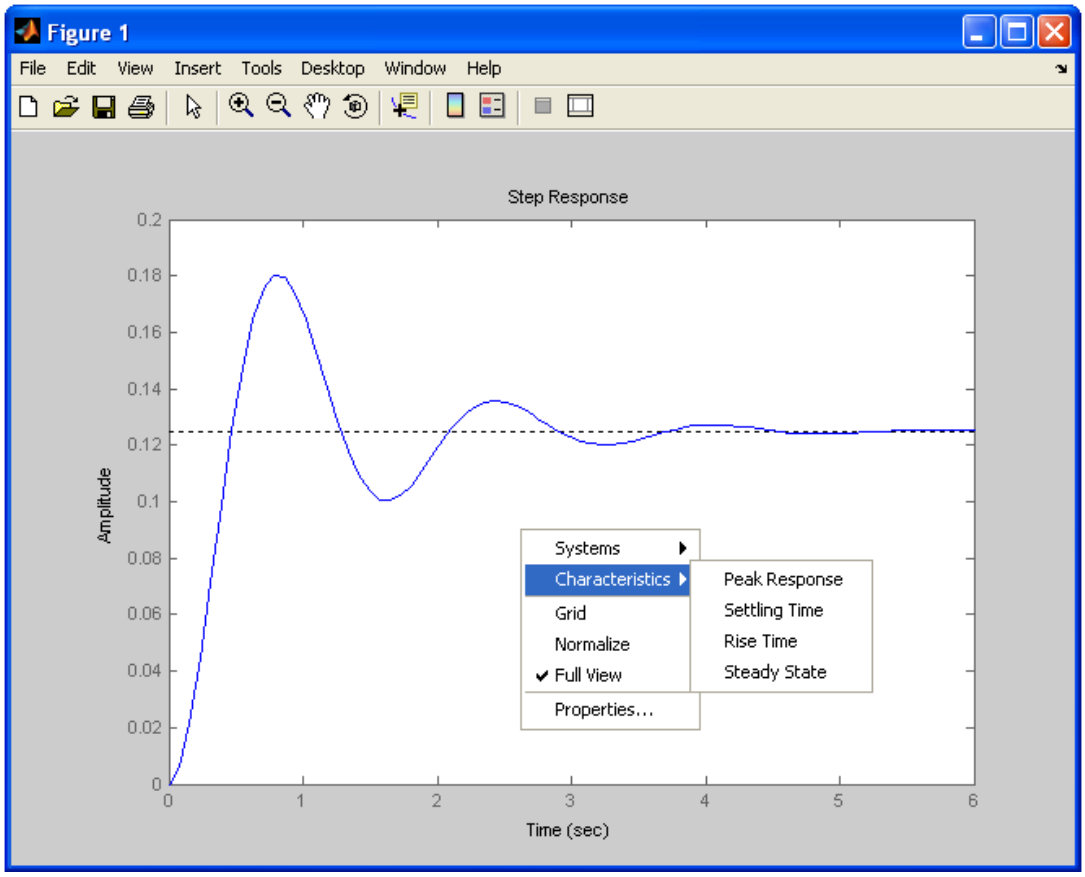
`pole(G)`	renvoie les pôles du système.
`zero(G)`	renvoie les zéros du système.
`damp(G)`	renvoie l'amortissement et la pulsation propre associés aux pôles.
`dcgain(G)`	renvoie le gain statique du système.
`pzmap(G)`	représentation graphique des pôles et zéros dans le plan complexe
`ctrb(sys)`	renvoie la matrice de commandabilité de la représentation d'état.

`obsv(sys)` renvoie la matrice d'observabilité de la représentation d'état.

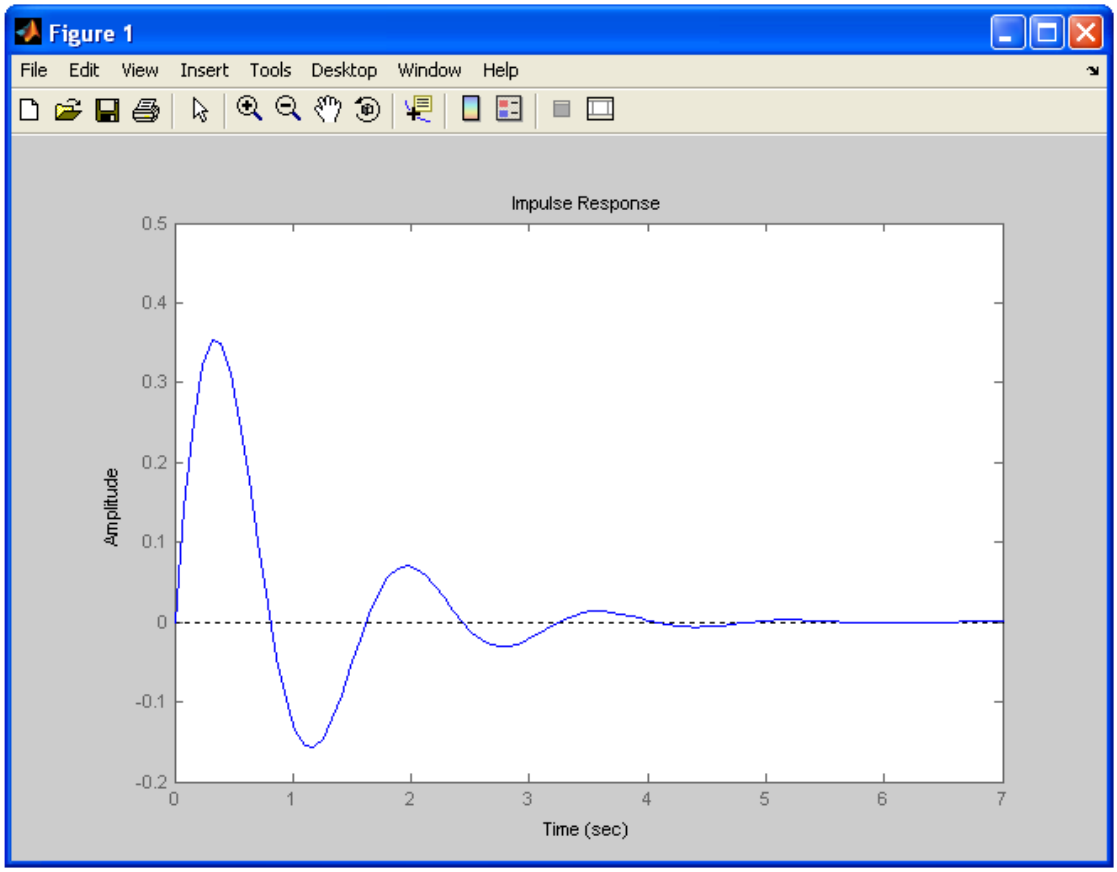
2.2 Réponses temporelles

Un ensemble de fonctions permettent de calculer et tracer la réponse d'un système à une entrée donnée. Plus particulièrement, la fonction `step` simule la réponse indicielle et `impulse` la réponse impulsionnelle d'un système donné en argument. Pour les systèmes de type `ss`, `initial` calcule le régime libre d'un système pour des conditions initiales spécifiées. Concernant les options d'affichage, la syntaxe est identique à la fonction `plot`.

```
>> G = tf(2,[1 2 16]);
>> step(G)
```




```
>> impulse(G)
```

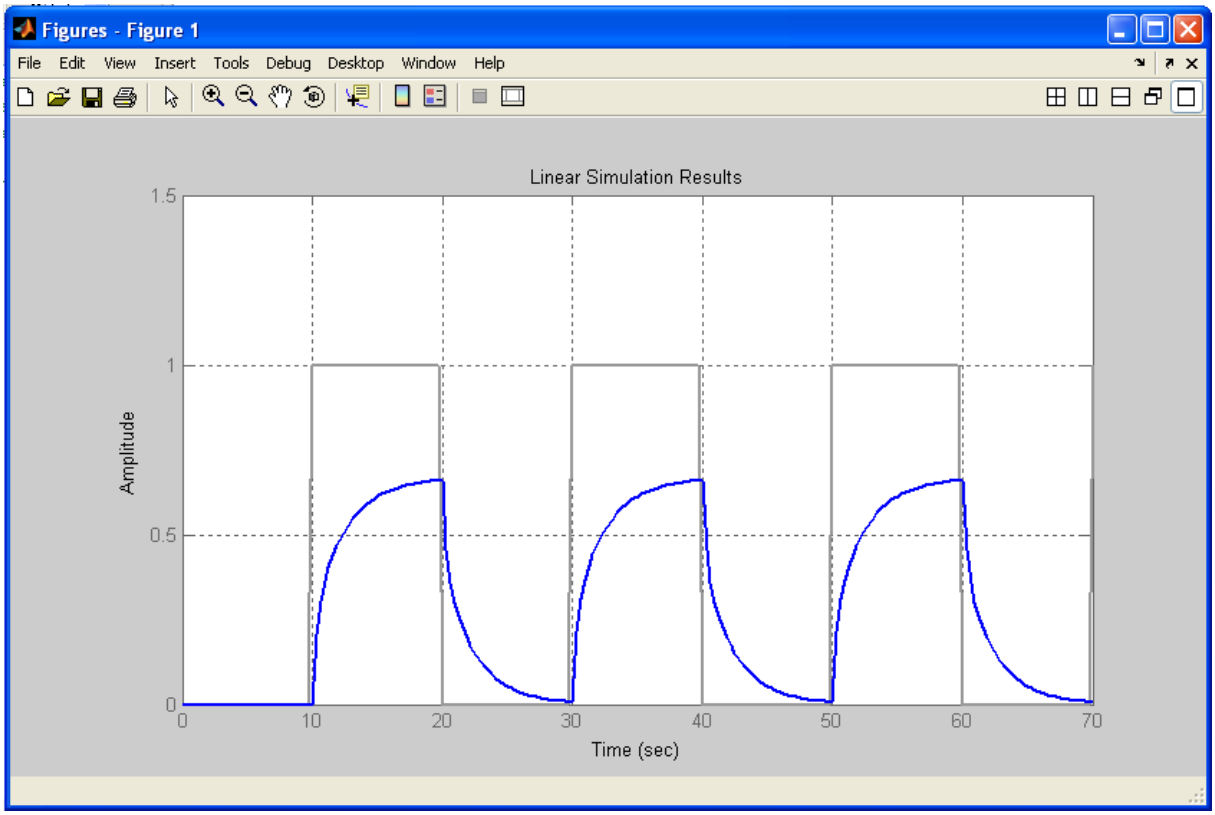


Un clic droit sur la figure permet d'obtenir des informations sur différentes caractéristiques du système testé.

De manière générale, il est possible de calculer la réponse d'un système pour une entrée arbitraire via la commande `lsim`. Elle peut prendre trois arguments: le système, le vecteur d'entrée et le vecteur temps. Ces deux vecteurs doivent évidemment être de même taille. Le vecteur d'entrée peut être construit point par point ou encore être généré à l'aide de `gensig` (pour des signaux carré, sinusoïdale ou impulsions).

```
>> G=tf([1 1],[1 4 3/2]);
>> [U,T]=gensig('square',20,70);
>> lsim(G,U,T)
```

La seconde ligne crée un signal carré périodique de période 20 sur un horizon temporelle de 70s. La fonction renvoie le vecteur du signal et le vecteur temps.



2.3 Lieux de transfert

L'analyse fréquentielle d'un système peut être effectuée facilement à l'aide des fonctions spécifiques, réalisant les tracés des lieux de Bode, Nyquist et Black-Nichols :

```
bode(G)
nyquist(G)
nichols(G)
```

Pour chacune des commandes, il est possible de spécifier explicitement l'intervalle de pulsations considérées. La syntaxe des options de traçage est identique à celle de `plot`. En précisant des arguments de sortie, les fonctions retournent les vecteurs des valeurs du tracé. Un clic droit sur la figure permet d'obtenir des informations sur différentes caractéristiques du système testé. Dans le cas de la fonction de `nichols`, la commande `grid` permet de visualiser l'abaque de Black. Enfin, notons la fonction `margin` qui est similaire à `bode` mais affiche en plus les marges de gain et de phase.

3 - Simulink

Simulink[7] est un logiciel muni d'une interface graphique pour la modélisation, la simulation et l'analyse des systèmes dynamiques. Etant intégré à MATLAB, les deux environnements sont parfaitement compatibles et les différentes fonctionnalités de ce dernier sont alors directement accessibles. Simulink est basé sur une interface graphique qui permet une contruction aisée et conviviale de

[7] Simulink est un produit développé par la société The MathWorks, Inc.. Simulink® est une marque déposée par cette même société.

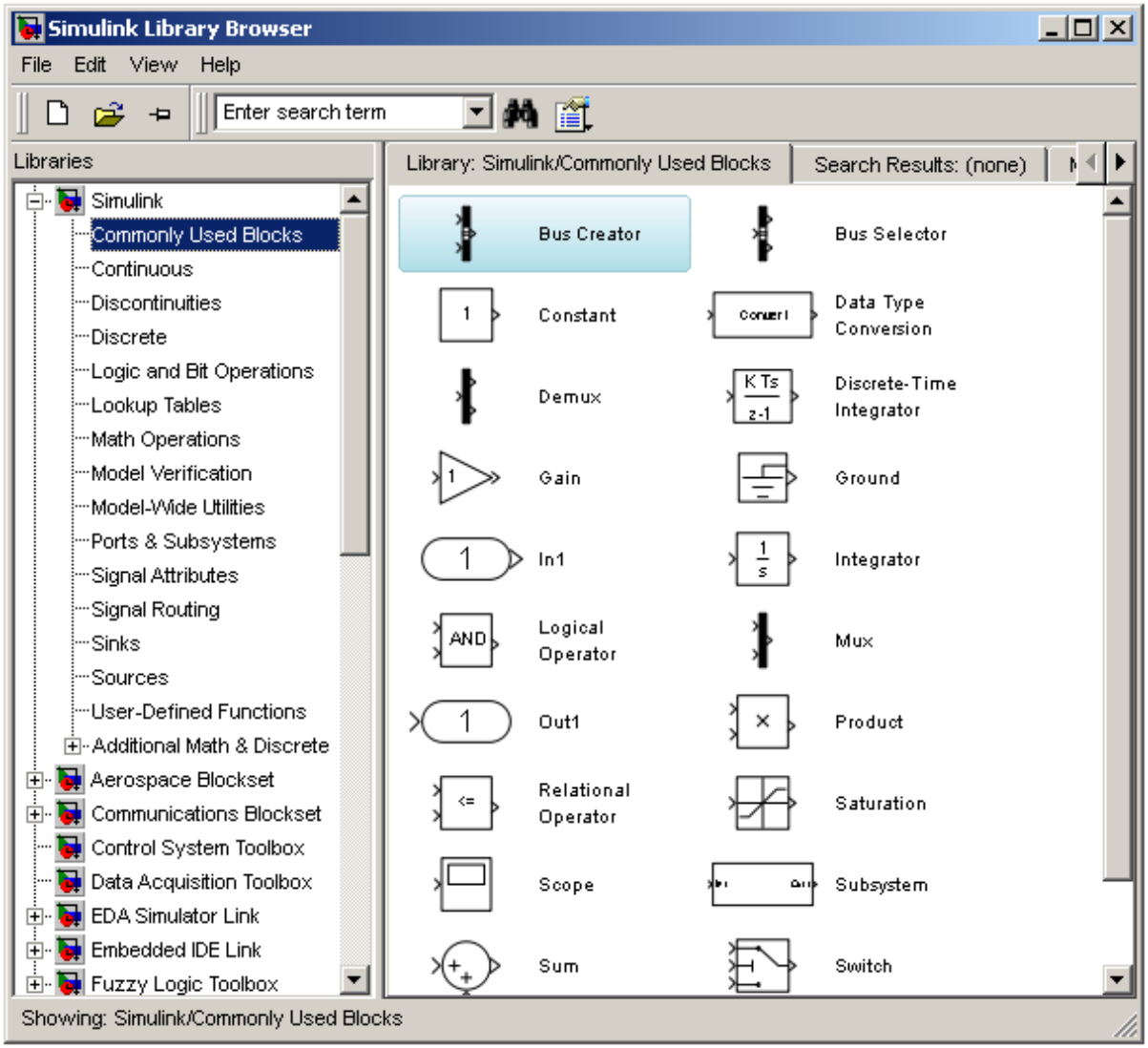
schémas-blocs. Chaque bloc composant le système est sélectionné depuis un ensemble de bibliothèques prédéfinies.

3.1 Création d'un modèle

Simulink peut être lancé depuis l'environnement de MATLAB

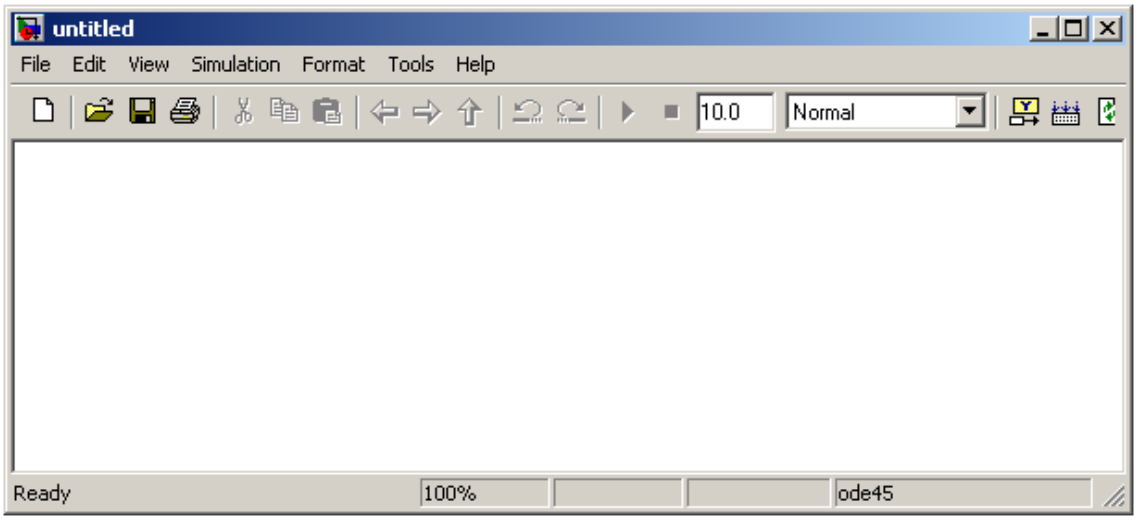
- en cliquant, dans la barre d'outils, sur le bouton 
- ou en tapant `simulink`

La fenêtre suivante apparaît

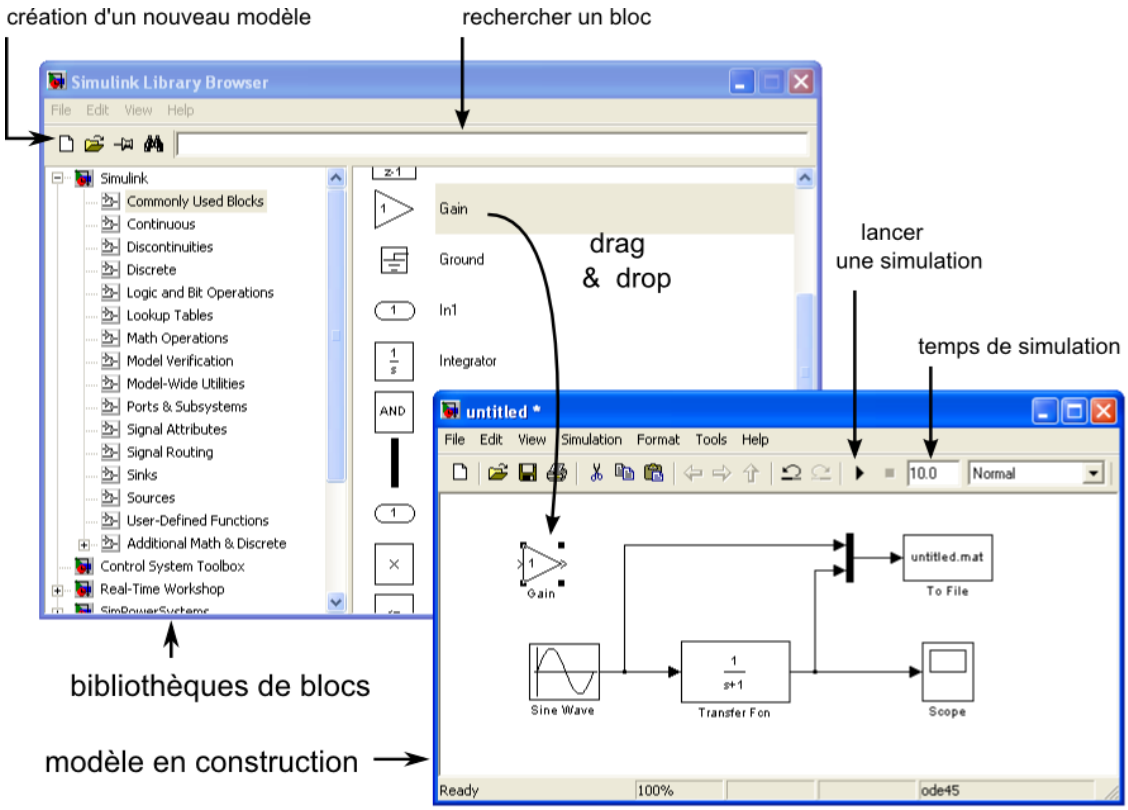


La colonne de gauche liste les bibliothèques disponibles. Celles-ci sont rangées par disciplines (Automatique, Aerospace, Communication, Traitement du signal, Système embarqués…) et regroupent un ensemble de blocs fonctionnels liés à une catégorie de fonctions particulières. A la sélection d'une bibliothèque, les blocs qui la composent sont affichés dans la partie de droite.

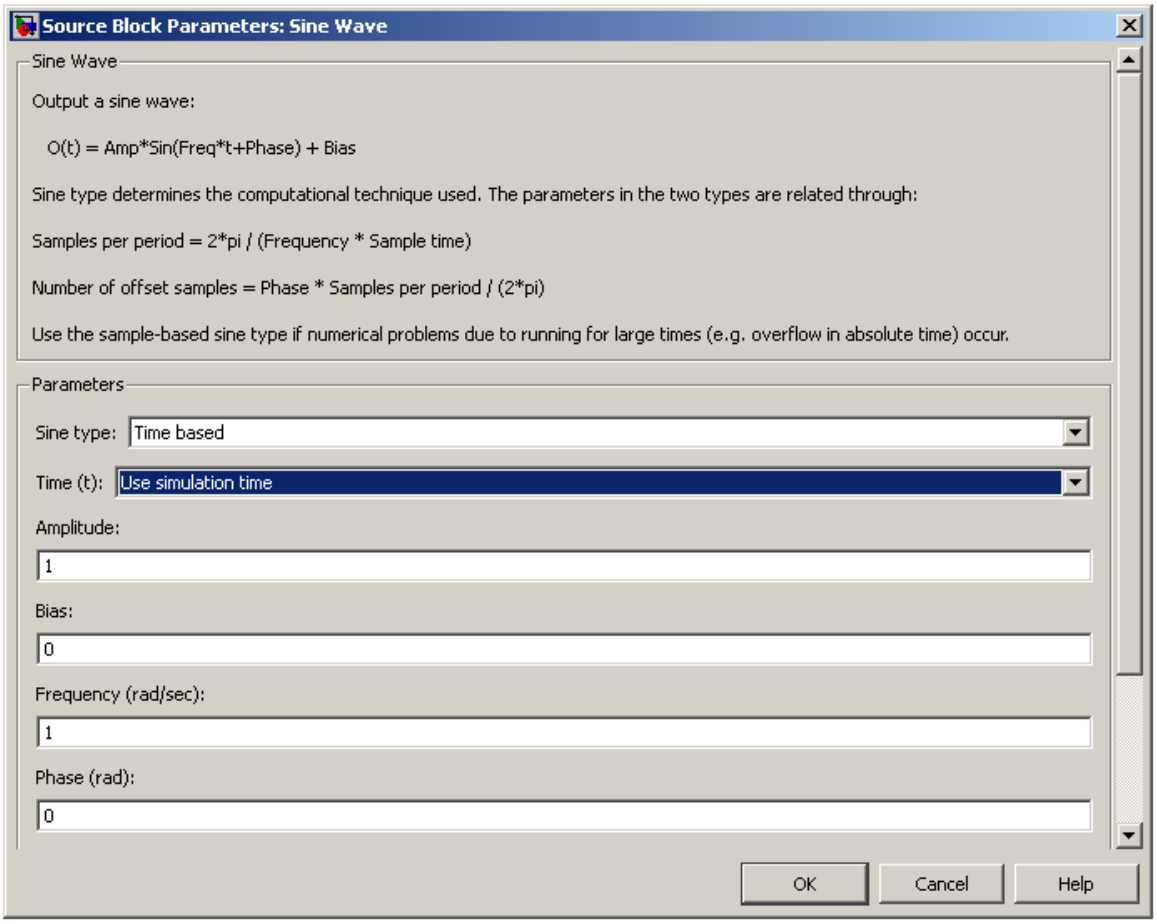
Un nouveau modèle est créé en cliquant sur l'icone « feuille blanche » (ou dans le menu `File > New > Model`). Une fenêtre vide apparaît, elle va servir de support pour construire votre modèle.



L'interface est intuitive et facile à utiliser, un modèle est construit à partir des blocs fonctionnels par glisser-déposer. Les blocs sont ensuite interconnectés par des flèches dessinées à l'aide de la souris (cliquer-maintenir sur une entrée ou une sortie puis relâcher sur une entrée ou une sortie). Chaque bloc peut être édité (réglages de ses paramètres) en double-cliquant dessus. Cette dernière action permet d'ouvrir une fenêtre de dialogue dans laquelle une description du bloc et des champs paramétrables sont donnés.



Par exemple, si l'on souhaite éditer le bloc `Sine Wave` (obtenu dans la bibliothèque `Sources`), un double-clique donne la fenêtre



Cette fenêtre décrit la fonction réalisée par le bloc ainsi que les différents paramètres à régler.

Les modèles peuvent être sauvegardés dans des fichiers d'extension « *.mdl* ». Simulink manipule les fichiers (sauvegarde, ouverture, fermeture, création…) de façon classique à l'aide du menu `File` ou des icones. Un fichier existant (par exemple « monschema.mdl ») peut également être directement invoqué depuis la fenêtre de commande MATLAB en tapant son nom sans extension :

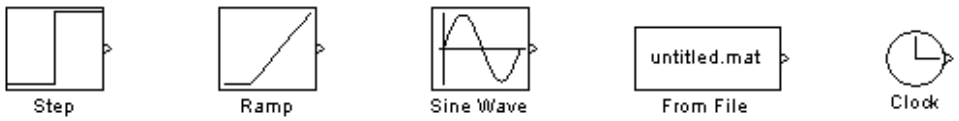
```
>> monschema;
```

3.2 Quelques bibliothèques

Simulink possède de nombreuses bibliothèques, adaptées pour la modélisation d'une multitude de systèmes de différentes natures. De plus, l'ajout de toolboxes supplémentaires permet d'enrichir les librairies et les fonctionnalités du logiciel. Nous présentons dans ce paragraphe seulement quelques blocs simples et présents dans la version de base de Simulink. Nous allons voir quelques blocs standards, souvent utilisés pour la simulation de systèmes asservis LTI.

Sources

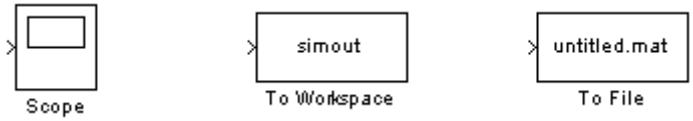
Les sources sont des blocs possédant une ou plusieurs sorties et aucune entrée. Ils sont utilisés pour la génération de signaux



Signal échelon ; signal rampe ; signal sinusoïdal ; valeurs fournies par un fichier ; temps.

Sinks

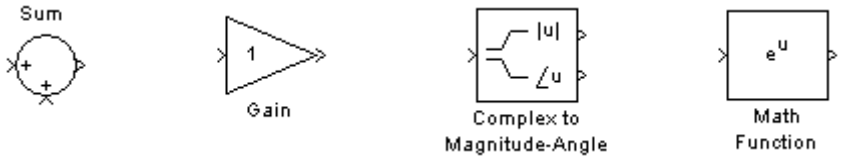
Les blocs de cette librairie, à l'inverse des sources, ne possèdent qu'une ou plusieurs entrées (*sink* signifie lavabo, c'est-à-dire qui collecte le flux d'information). Ils sont utilisés pour l'affichage (digital, oscilloscope) ou la mémorisation de signaux (vers une variable, un fichier).



Affichage type oscilloscope ; stockage dans une variable; stockage dans un fichier.

Math Operations

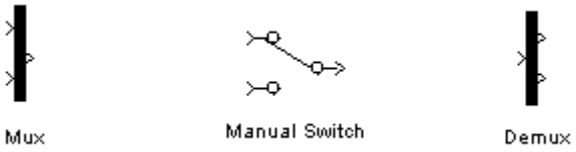
Ensemble de blocs réalisant une fonction mathématique appliquée aux signaux entrants. Le (ou les) résultat(s) est (sont) renvoyé(s) sur le (les) point(s) de sortie.



Somme/soustrait deux signaux ; multiplie un signal; fournit le module et la phase ; applique une fonction standard.

Signal Routing

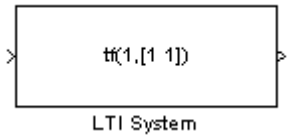
Ensemble de blocs utiles pour l'aiguillage de signaux ou la connexion de blocs.



Multiplexeur: combine deux entrées en une seule (vecteur) ; switch: permet de sélectionner manuellement une entrée; démultiplexeur: sépare une entrée (vecteur) en plusieurs composantes.

Control System Toolbox

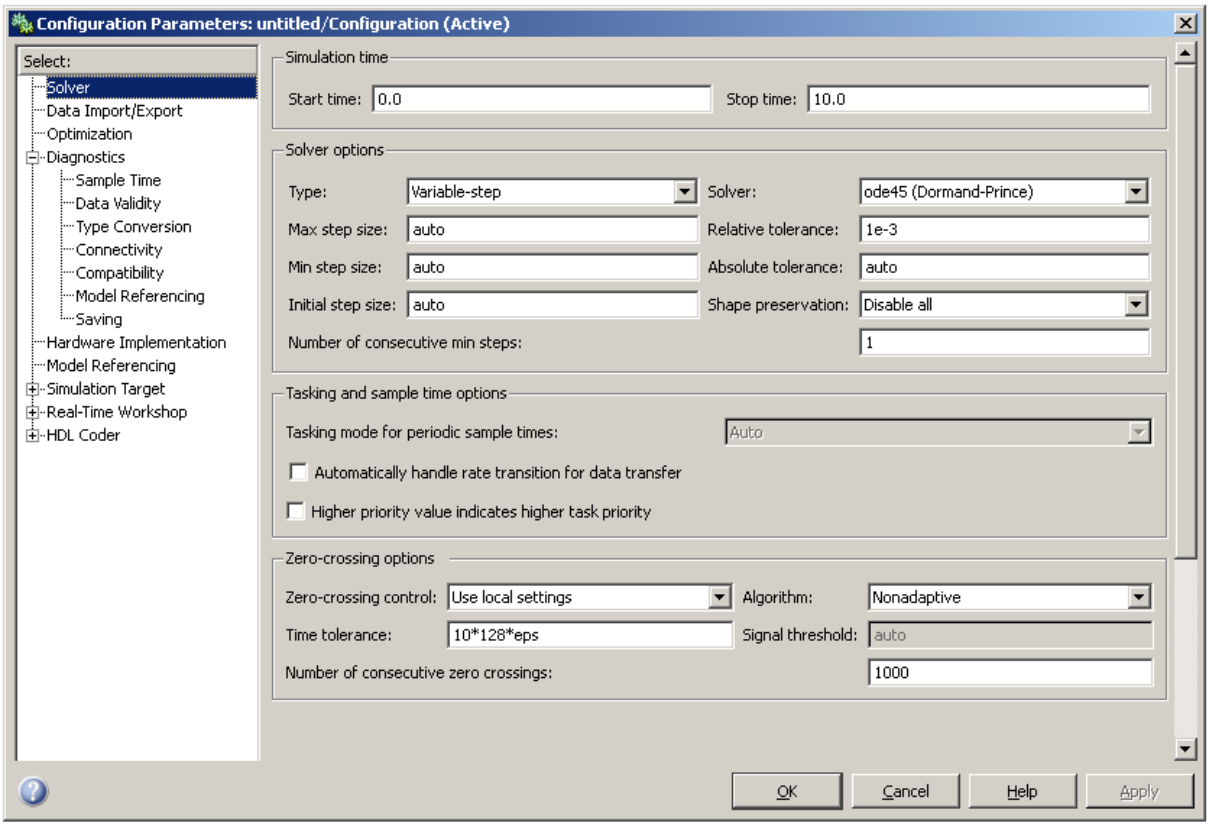
Bloc « LTI system » pour la définition de fonctions de transfert.



La fonction de transfert peut être directement définie via les paramètres du bloc en suivant la syntaxe de la fonction `tf` (vue précédemment). Par ailleurs, si une fonction de transfert a déjà été définie dans le workspace de MATLAB, elle peut être reprise dans le bloc en spécifiant simplement son nom.

3.3 Simulation

Une fois le modèle réalisé, l'intérêt de Simulink consiste à le simuler. Une simulation peut être lancée soit à partir de l'icône « lecture » en forme de triangle (ou dans le menu `Simulation > Start`), soit à partir de MATLAB avec la commande `sim()`.
Le premier paramètre de simulation (et le seul, dans le cas d'utilisation simpliste de Simulink) à régler est le temps de simulation. Il peut être spécifié dans le champ à droite de l'icône de lancement d'une simulation. Il doit être exprimé en seconde.
Pour une utilisation avancée, de nombreuses options permettent de configurer de manière très précise les paramètres de simulation. De telles configurations sont accessibles dans le menu `Simulation > Configuration Parameters…` :



Cette interface donne la possibilité de configurer (entre autres):

- l'horizon temporel de simulation (en sec), le solver utilisé (et ses paramètres) pour la résolution numérique des équations différentielles,
- la gestion des entrées/sorties avec MATLAB, les variables sauvegardées,
- les notifications d'erreurs ou d'alertes sur différents événements, diagnostic de simulation,
- …

Nous n'entrerons pas plus dans les détails du paramétrage d'une simulation car très complexe et inutile dans le cadre d'une simple introduction à Simulink.

3.4 Exemple

Considérons un système modélisé par la fonction de transfert suivante :

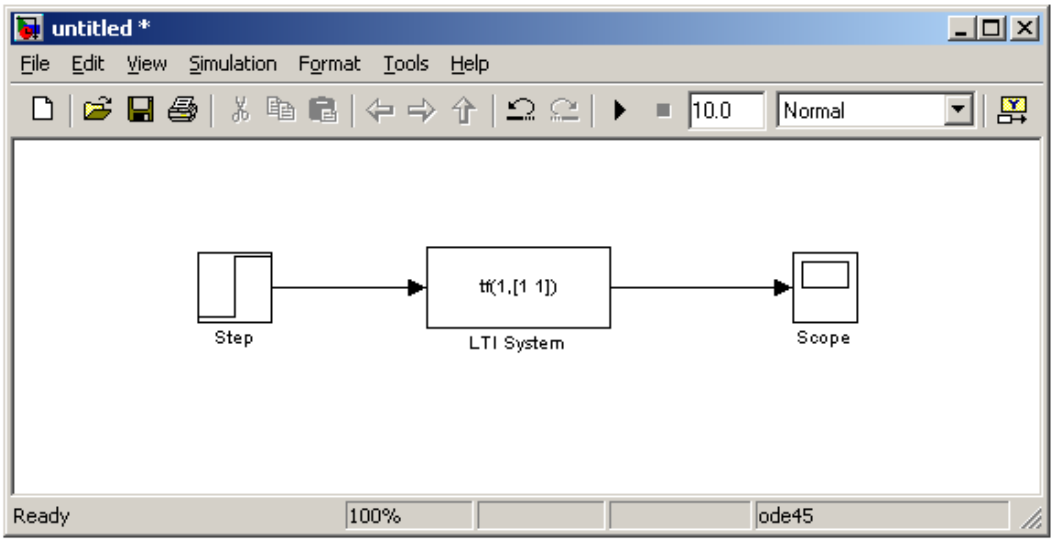
$$G(p) = \frac{1}{p^2 + p + 2}$$

Définissons celle-ci dans MATLAB:

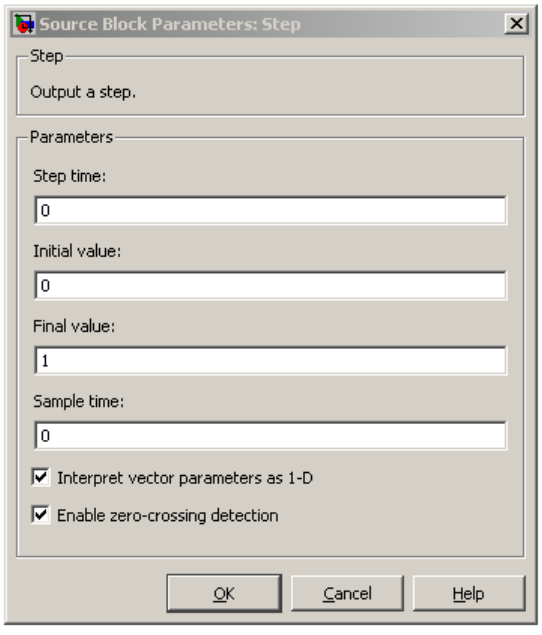
```
>> G = tf(1,[1 1 2])
```

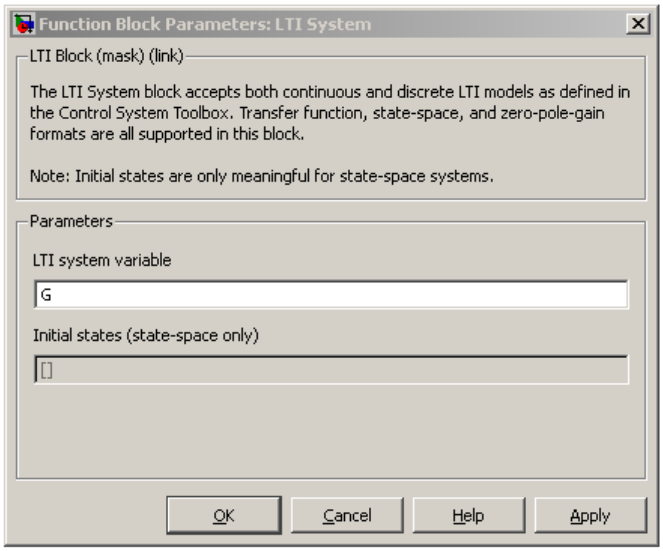
Nous souhaiterions savoir comment le système réagit à une sollicitation de type échelon unitaire. Pour cela, construisons un modèle Simulink afin de simuler la réponse indicielle de notre système. Nous avons donc besoin :

- d'un bloc pour représenter la fonction de transfert G(p),
- d'une source pour générer le signal échelon,
- et un bloc permettant la visualisation graphique de la sortie du système.

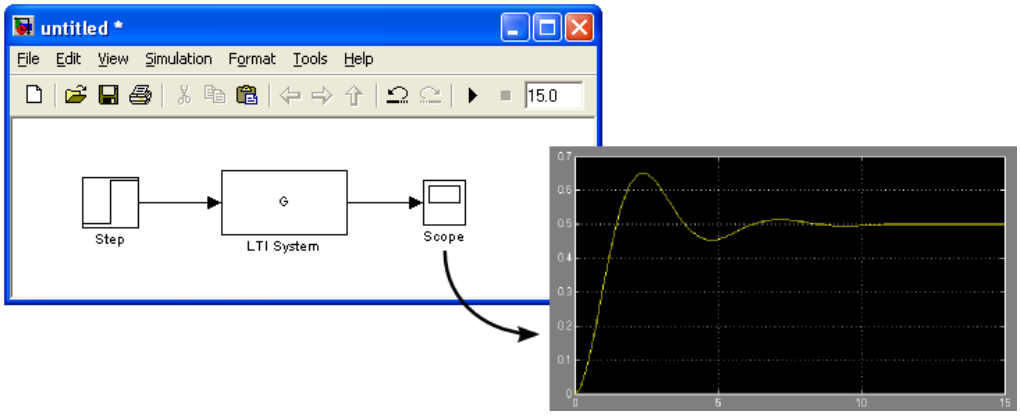


Paramétrons les blocs `Step` et `LTI System` pour notre exemple.

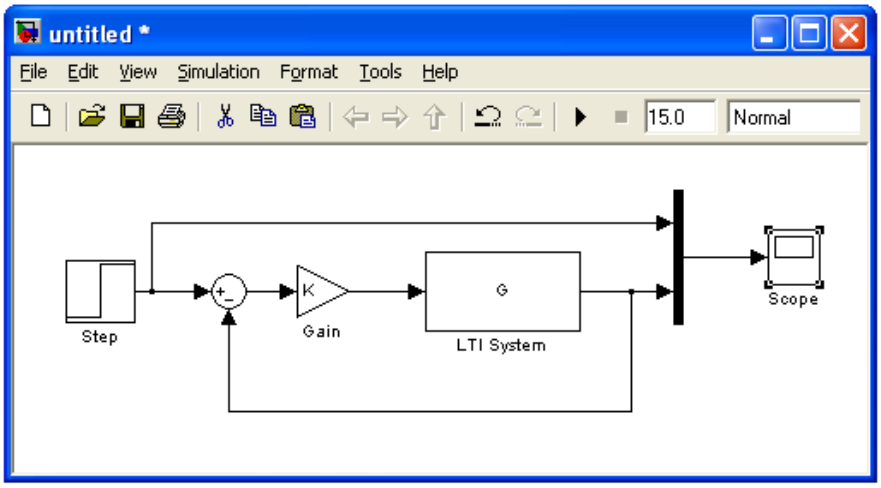




Nous fixons un échelon de valeur initiale 0 (initial value), d'amplitude 1 (final value) et se déclanchant à t=0 (step time). Nous définissons ensuite le système que nous étudions par la variable `G` précédement définie dans MATLAB. Enfin, la simulation sera effectuée sur un horizon temporel de 15s.
Après execution de la simulation, la réponse peut être observée en double-cliquant sur l'oscilloscope

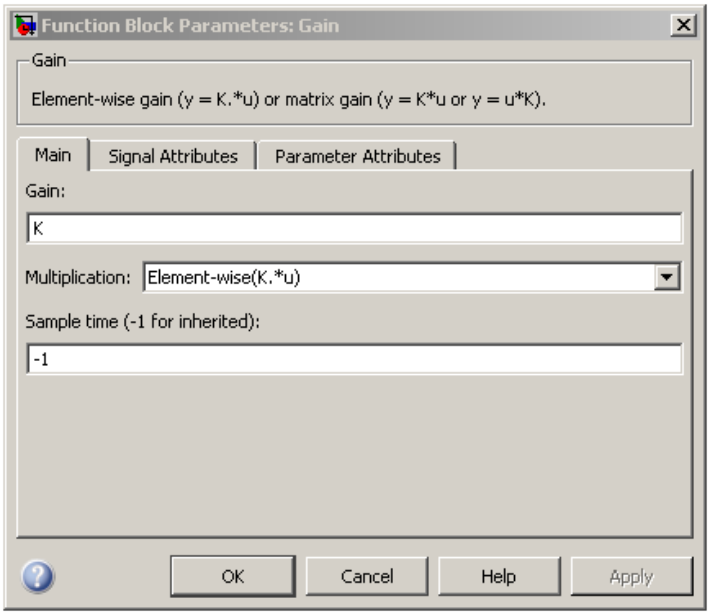


Réalisons maintenant un asservissement avec une commande proportionnelle de gain K. Notre nouveau modèle de simulation nous permettra de tester les performances de l'asservissement pour différentes valeurs de K. Il peut être fixé directement depuis le bloc ou via MATLAB. Ici le multiplexeur permet d'afficher sur un seul graphe le signal de consigne et le signal de sortie.

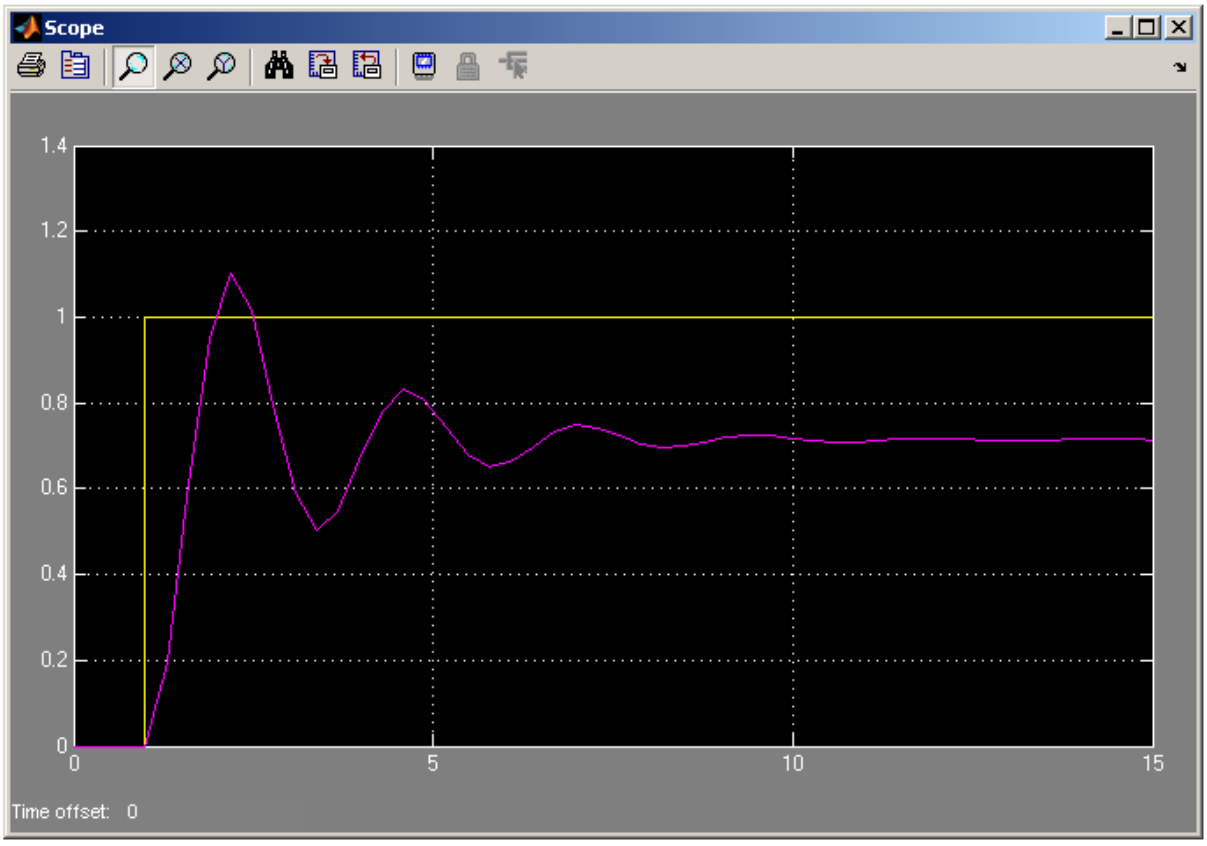


La variable K a été spécifiée dans la boite de dialogue du bloc `Gain`. Celle-ci doit être préalablement définie avec une valeur numérique. Exécutons dans MATLAB la commande :

```
>> K = 5
```



Simulons notre modèle et visualisons la réponse du système asservi via le `Scope`.



En modifiant la valeur de K dans MATLAB, on peut facilement et rapidement analyser (par simulation) le comportement de l'asservissement en fonction du gain de correction.