TUTORIEL MATLAB

L'aide de Matlab

L'aide intégrée de Matlab est très efficace, apprenez à en utiliser les bases. Cliquez [ici].

Initiation à Matlab

Nous vous conseillons d'ouvrir une fenêtre Matlab et de reproduire les exemples donnés dans chacun des points suivants.

- **Les bases**
  Création, modification de matrices, accès aux éléments, gestion de la mémoire. Cliquez [ici]
- **Les matrices**
  Opérations élémentaires sur les matrices. Cliquez [ici]
- **L'opérateur "colon"**
  Permet de générer des suites de nombres. Cliquez [ici]
- **Structures de contrôle**
  Boucles, tests et conditions. Cliquez [ici]
- **Scripts et fonctions**
  Réaliser des fonctions personnelles, découper un programme en plusieurs fichiers. Cliquez [ici]

Matlab est un logiciel extrêmement bien documenté, avec plusieurs fonctions d'aide très performantes. Lorsqu'on est confronté à un problème technique dans l'utilisation de Matlab, ces fonctions d'aide permettent souvent de le résoudre rapidement.

Fonction `help`

Lorsque vous recherchez de l'aide sur une fonction précise de Matlab, vous pouvez simplement taper la commande:

`>> help` *nom_de_la_fonction*

Exemple

```
>> help det
DET Determinant.
DET(X) is the determinant of the square matrix X.
Use COND instead of DET to test for matrix singularity.
See also COND. Overloaded methods
help sym/det.m
```

Note

Dans l'aide de Matlab, même si les noms des fonctions sont indiqués en majuscule, il convient d'écrire ces noms en minuscule lors de l'appel de ces fonctions. Par exemple, si on souhaite calculer le déterminant de la matrice $\begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix}$ on écrira dans Matlab:

```
>> det([4 2; 1 3])
ans =
10
>>
```

Fonction `lookfor`

La fonction `lookfor` permet de rechercher les fonctions se rapportant à un certain sujet. Vous devez simplement veiller à entrer les mots-clés en anglais. `>> lookfor` *mot_clé_en_anglais*

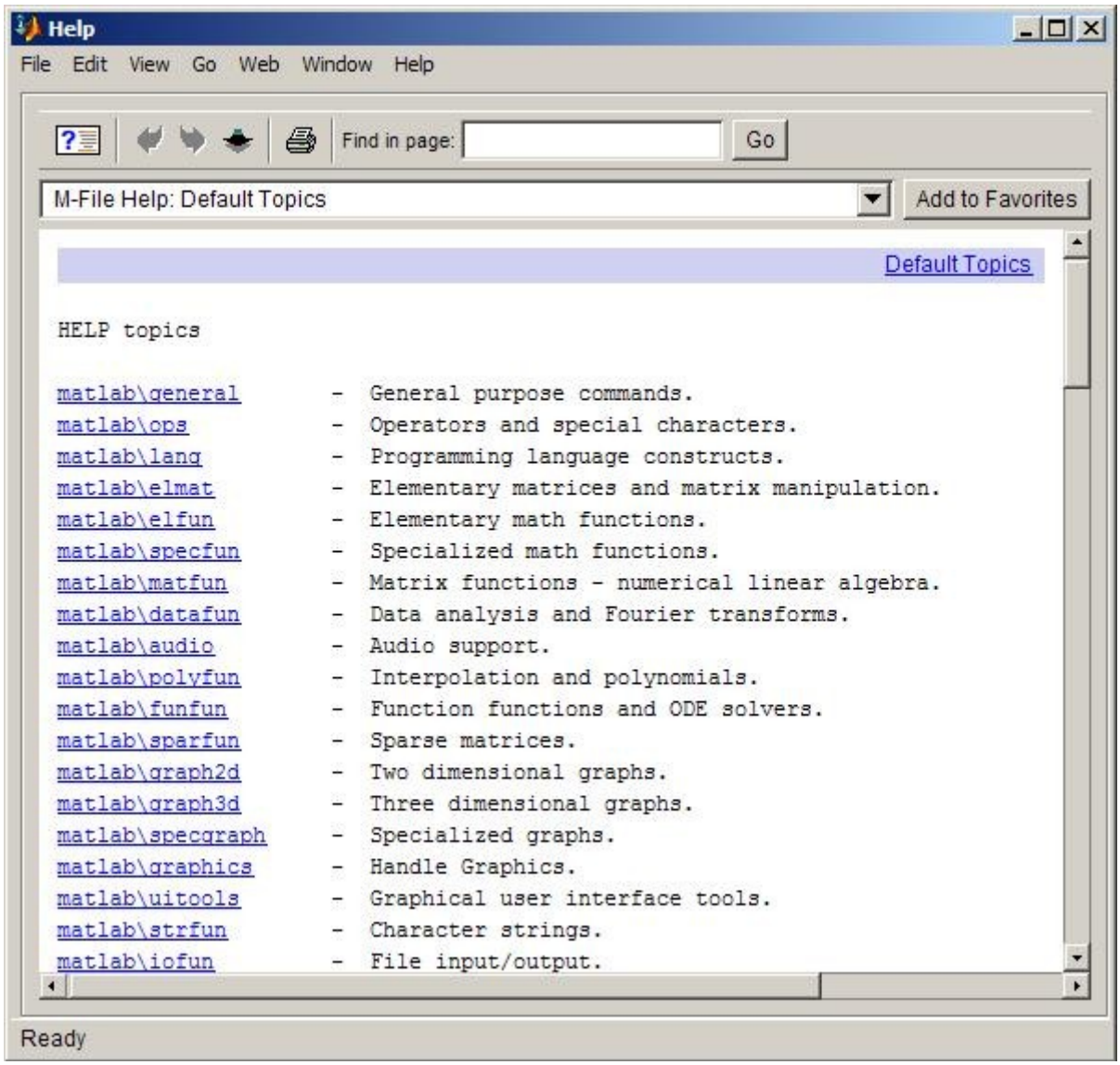
Exemple

```
>> lookfor determinant
DET Determinant.DET
```

Fonction `helpwin`

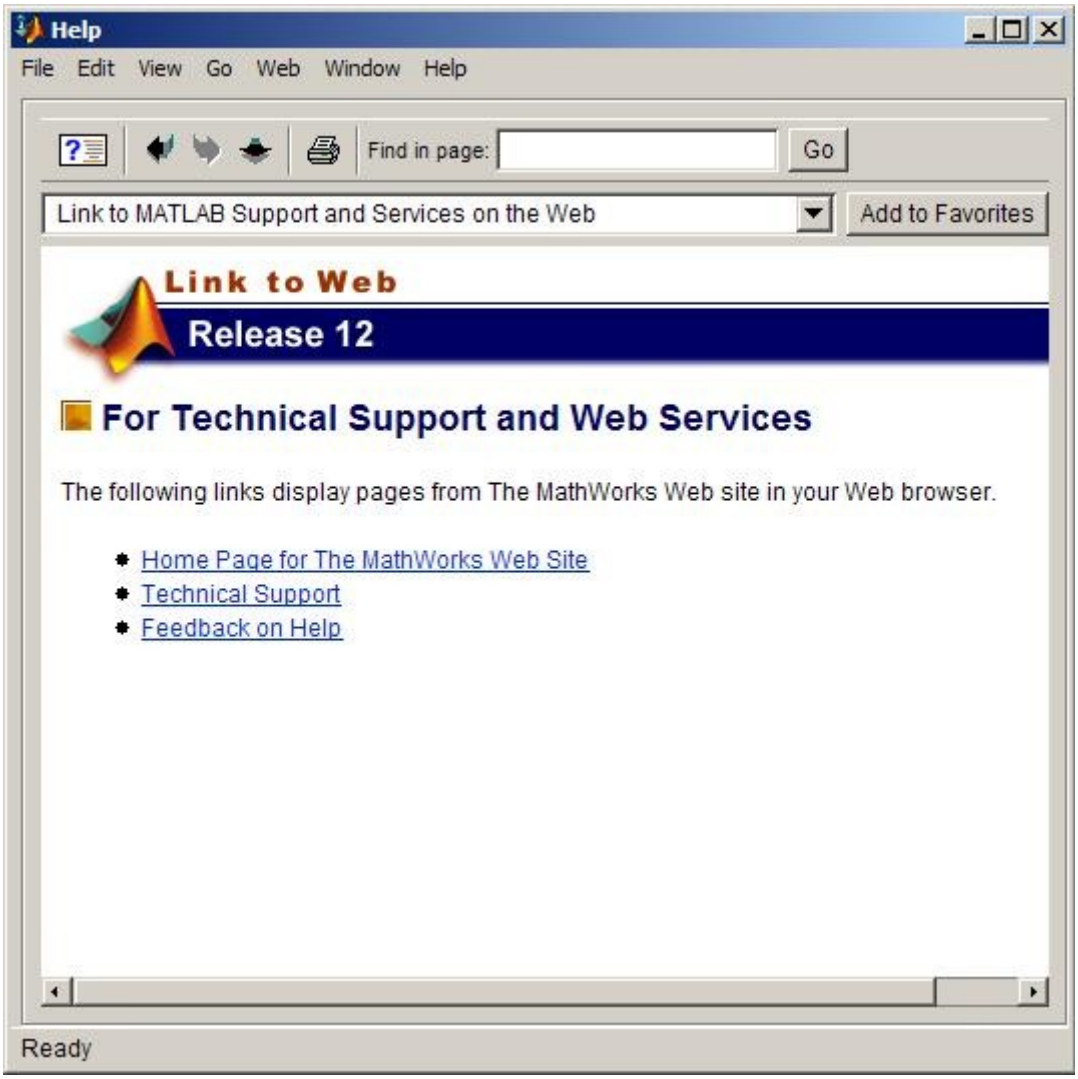
La fonction `helpwin` fait apparaître une fenêtre qui vous permet de naviguer à travers l'aide des fonctions de Matlab. On écrira dans Matlab:

```
>> helpwin
```



Fonction `doc`

Matlab dispose d'une documentation très fournie sous la forme de pages HTML. Pour accéder à ces pages, tapez la commande: `doc` ou `helpdesk` . Si vous recherchez une information sur une commande précise : `doc nom_de_la_commande`

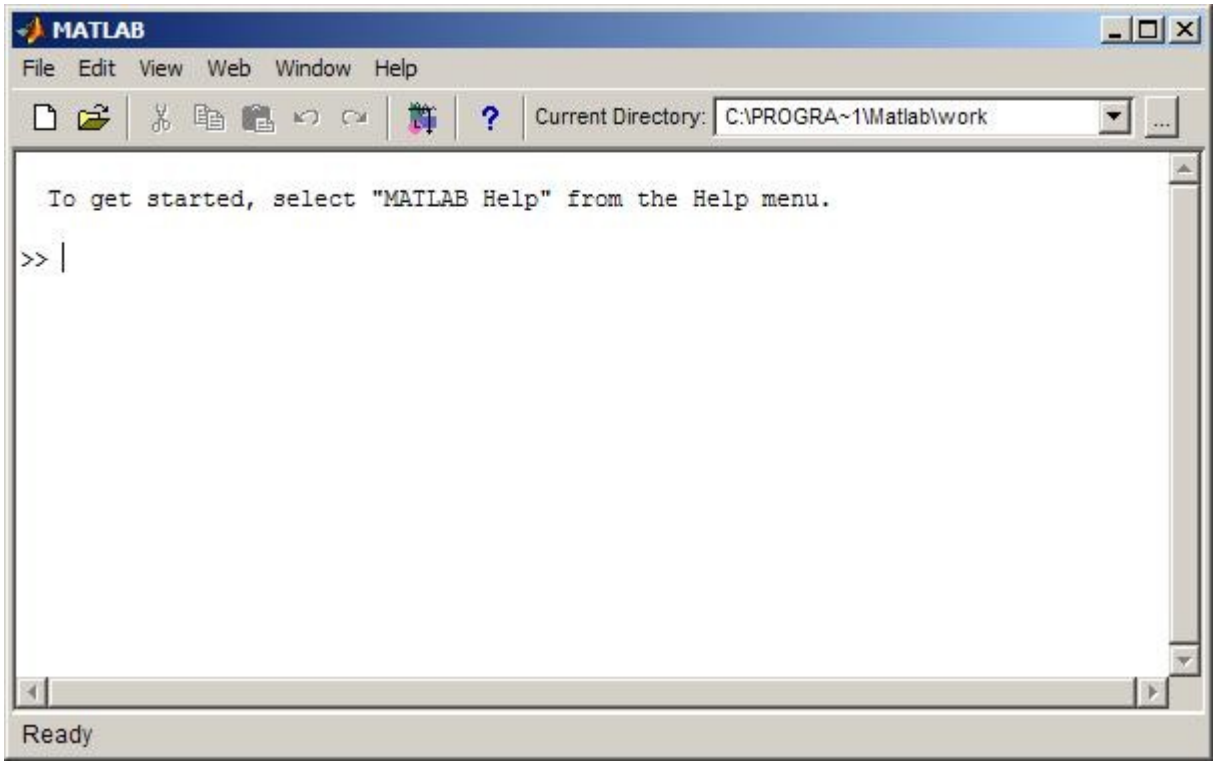


Introduction

Matlab, et non pas Mathlab est l'abréviation de MATRIX LABORATORY. C'est un logiciel de calcul numérique, utilisant des matrices comme objets de base. Ces matrices peuvent aussi être des vecteurs (ligne ou colonne), ainsi que des scalaires. Matlab manipule aussi bien les nombres réels que les nombres complexes.

Ce qui fait la popularité de Matlab, c'est la facilité avec laquelle on peut programmer des méthodes numériques, les tester et visualiser les résultats sous forme graphique. C'est un logiciel qui se destine plus à l'expérimentation numérique. Lorsqu'on doit résoudre un problème de très grande taille demandant une importante puissance de calcul, il vaut souvent mieux utiliser des logiciels spécifiques, ou programmer soi-même les méthodes numériques dans un langage tel que C, C++, Fortran, Java,....

Interface



Les menus et les boutons au sommet de la fenètre permettent d'accéder à certaines fonctionalités de Matlab sous Windows (éditeur de scripts, gestion des variables, gestion des chemins d'accès,...). Au centre de la fenètre, on trouve la ligne de commande de Matlab qui permet d'entrer des instructions et de lire les résultats. C'est là que nous allons travailler.

Créer une matrice

On peut créer une matrice `A` explicitement:

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

La matrice est délimitée par des crochets, on entre les éléments ligne par ligne, avec un point-virgule pour séparer les lignes de la matrice. De la mème façon, on peut créer un vecteur ligne:

```
>> B = [1 5]
B =
1 5
```

ou un vecteur colonne:

```
>> C = [3 ; 4 ; 5]
C =
3
4
5
```

Une autre façon de faire est de créer un vecteur colonne et de le transposer avec l'opérateur de transposition: l'apostrophe.

```
>> D = [1 2]'
D =
1
2
```

Les scalaires se déclarent indifféremment comme des matrices 1*1, soit directement comme des scalaires:

```
>> E = [5]
E =
5
>> F = 6
F =
6
```

On accède aux éléments d'une matrice en spécifiant entre parenthèses et séparés d'une virgule, la ligne et la colonne de l'élément désiré:

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

Pour les vecteurs colonne (resp. ligne), on peut choisir de ne pas spécifier la colonne (resp. ligne):

```
>> C(2,1)
ans =
4
```

Ou de façon équivalente:

```
>> C(2)
ans =
4
```

Pour les scalaires, on peut ne pas spécifier ni la ligne, ni la colonne:

```
>> F(1,1)
ans =
6
```

Ou encore:

```
>> F(1)
ans =
6
```

Ou encore:

```
>> F
F =
6
```

Pour connaître la signification du mot clé `ans`, on peut taper :

```
>> help ans

ANS Most recent answer.
ANS is the variable created automatically when expressions
are not assigned to anything else. ANSwer.
```

Pour une introduction à l'aide de Matlab, cliquez [ici]. Matlab possède aussi tout une série de fonctions pour créer des matrices spéciales essayez:

```
>> help elmat
```

Attention, la casse a de l'importance pour Matlab, `A` n'est pas équivalent à `a` ! On peut, par exemple, définir:

```
>> a = 2
a =
2
```

La variable `A` existe toujours:

```
>> A
A =
1 2 3
4 5 6
```

Modifier une matrice

Il est possible de modifier un des éléments d'une matrice :

```
>> A(1,3) = 2
A =
1 2 2
4 5 6
```

Si on modifie un élément inexistant d'une matrice, la matrice est agrandie jusqu'à ce que cet élément existe :

```
>> A(3,3) = 1 A =
1 2 2
4 5 6
0 0 1
```

Gestion de la mémoire

La facilité avec laquelle on crée de nouvelles variables dans Matlab fait que la mémoire risque vite d'être encombrée avec une multitude de variables dont on ne se sert plus. La commande `who` donne la liste des variables de l'espace de travail. La commande `whos` donne la mème liste, ainsi que des informations sur la nature et la taille de chaque variable.

```
>> who
Your variables are:
A C E a
B D F

>> whos
Name Size Bytes Class
A 3x3 72 double array
B 1x2 16 double array
C 3x1 24 double array
D 2x1 16 double array
E 1x1 8 double array
F 1x1 8 double array
a 1x1 8 double array

Grand total is 19 elements using 152 bytes
```

Pour effacer une variable et libérer la mémoire qu'elle occupait, il faut effectuer la commande:

`clear` *nom_de_la_variable*

```
>> clear F

>> who
Your variables are:
A C E
B D a
```

La commande `clear all` efface toutes les variables de l'espace de travail. Il n'est pas possible d'annuler (undo) la commande `clear`.

Affichage

Lorsqu'on souhaite que le résultat d'une commande ne s'affiche pas, il faut placer un point-virgule à la fin de celle-ci:

```
>> T = [1 2 3];
```

Aucun résultat n'est affiché, pourtant, la commande a bien été exécutée.

```
>> T
T =
1 2 3
```

Pour cette partie, on réutilisera les matrices qui ont été définies à la partie précédente [les bases].

Opérations sur les scalaires

Matlab permet d'effectuer les opérations classiques sur les scalaires :

```
>> toto = (E-a)^a*a/E+1
toto =
4.6000
```

Les fonctions classiques comme sin, cos, log, ... existent aussi. (essayez `helpwin ops`, ou `helpwin elfun`)

```
>> titi = sin(a)
titi =
0.9093
```

NB : Les fonctions trigonométriques travaillent en radians.

Valeurs particulières

Certaines valeurs particulières sont définies dans Matlab:

- `pi` : c'est la constante pi avec un maximum de chiffres après la virgule.
- `i` ou `j` : l'unité imaginaire. Si on a déjà utilisé i et j comme noms pour d'autres variables, ces constantes sont inaccessibles. On peut alors utiliser `sqrt(-1)`

```
>> tata = a + sqrt(-1) + 2*i;

>> tata = 2.0000 + 3.0000i;
```

On a un nombre complexe !

- `Inf` : L'infini. C'est le résultat renvoyé par Matlab quand un calcul devrait avoir un résultat infini.
- `NaN` : Not a Number. Lorsque le résultat d'un calcul est indéterminé.

```
>> b = -3/0
Warning: Divide by zero.

b = -Inf

>> c = 0/0
Warning: Divide by zero.

c = NaN
```

Opérations sur les matrices

Les opérations `+, - ,*, /` existent aussi pour les matrices. S'il n'y a pas d'ambiguïtés pour l'addition et la soustraction, il faut faire attention pour les opération de multiplication, division, puissance, etc.

Exemples:

```
>> a*A
ans =
2 4 4
8 10 12
0 0 2

>> A*A
ans =
9 12 16
24 33 44
0 0 1

>> A*C
ans =
21
62
5

>> T*A
ans =
9 12 17

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

Les matrices doivent avoir des dimensions compatibles.

Le signe `*` est soit l'opérateur de produit matriciel, soit l'opérateur de multiplication d'une matrice par un scalaire. Si on souhaite multiplier deux à deux les éléments de deux matrices, il faut utiliser l'opérateur `.*` .

```
>> C .* [2; 1; 2]
ans =
6
4
10
```

C'est la même chose pour l'opérateur `^`. Si `A^2` signifie `A*A` au sens matriciel, `A.^2` signifie qu'on prend tous les éléments de A élevés au carré. Pour plus de détail essayez : `helpwin ops`.

Résolution de systèmes linéaires

Une façon de résoudre un système linéaire `A*x = C`, est de calculer l'inverse de `A`:

```
>> x = inv(A)*C
x =
-5.6667
-0.6667
5.0000
```

Cette façon de procéder peut être très imprécise si la matrice `A` est mal conditionnée. En outre calculer l'inverse de la matrice demande beaucoup plus d'opérations qu'il n'en faut pour résoudre le système uniquement. Une meilleure façon de procéder est d'utiliser l'opérateur de division à gauche de Matlab (`\`). Matlab utilise alors un ensemble de méthodes mieux appropriées pour résoudre le système (Par exemple la factorisation LU de la matrice, suivie de la résolution de deux systèmes triangulaires).

```
>> x = A\C
x =
-5.6667
-0.6667
5.0000
```

Concaténation et extraction

Il est possible de créer de nouvelles matrices en concaténant des matrices déjà existantes. La syntaxe est similaire à celle de la création de matrices à partir de scalaires.

```
>> [A C;1 T]
ans =
1 2 2 3
4 5 6 4
0 0 1 5
1 1 2 3
```

Lorsqu'on écrit `A(1,3)`, on extrait en fait une sous-matrice de dimensions 1*1. Cette syntaxe est en fait extensible à des sous-matrices de plus grande taille.

```
>> A(2,[1 3])
ans =
4 6
```

Ici, on a extrait la sous matrice composée de la deuxième ligne de `A` et des colonnes 1 et 3.

```
>> A(2,[3 1 2])
ans =
6 4 5
```

Ici, on crée un matrice qui est constituée d'une permutation de la deuxième ligne de `A`.

```
>>            A([1            3            2],[3            1            2])
ans                                                                          =
2                                    1                                       2
1                                    0                                       0
6                        4                           5
```

Ici, on a simplement créé une matrice : `A`, dont on a permuté les lignes et les colonnes.

L'opérateur ":" (colon) est un opérateur permettant de générer des suites de nombres. Il est très utile pour extraire des sous-matrices, gérer les boucles d'itérations (voir la partie : structures de contrôle), etc.

Générer une suite de nombres

```
>> a = -2:3
a =
-2 -1 0 1 2 3

>> b = [-2:3]
b =
-2 -1 0 1 2 3

>> c = [-2:3]'
c =
-2
-1
0
1
2
3
```

La notation "a:b" signifie : tous les nombres de a à b par pas de 1.0 . Si ce n'est pas possible, on obtient un message d'erreur.

```
>> 3:-1
ans =
Empty matrix: 1-by-0
```

Modifier le pas

Si on avait voulu générer la liste de l'exemple précédent, mais en allant de 3 à 2, on aurait pu faire:

```
>> a = fliplr([-2:3])
a =
3 2 1 0 -1 -2
```

Mais il est plus simple d'écrire :

```
>> a = 3:-1:-2
a =
3 2 1 0 -1 -2
```

La notation "a : h : b" signifie : tous les nombres de a à b par pas de h. Le pas h peut être positif ou négatif. Pour générer tous les nombres pairs de 4 à 20, on écrit :

```
>> b = 4:2:20
b =
4 6 8 10 12 14 16 18 20
```

Sélectionner une sous-matrice

Soit la matrice A:

```
>> A = [1 2 3 4; 5 6 7 8 ; 9* ones(1,4); zeros(1,4)]
A =
1 2 3 4
5 6 7 8
9 9 9 9
0 0 0 0
```

Pour extraire la sous matrice 3*3 composée des 3 premières lignes et colonnes on écrit :

```
>> B = A(1:3,1:3)
B =
1 2 3
5 6 7
9 9 9
```

Pour extraire la dernière colonne :

```
>> A(1:4,4)
ans =
4
8
9
0
```

Il existe un raccourci qui permet de spécifier d'un coup toutes les lignes ou colonnes d'une matrice :

```
>> A(:,2)
ans =
2
6
9
0
```

Il faut lire : A(toutes les lignes, deuxième colonne). Cette notation est équivalente à :

```
>> A(1:size(A,1),2)
ans =
```

```
2
6
9
0
```

De la même façon, on peut écrire :

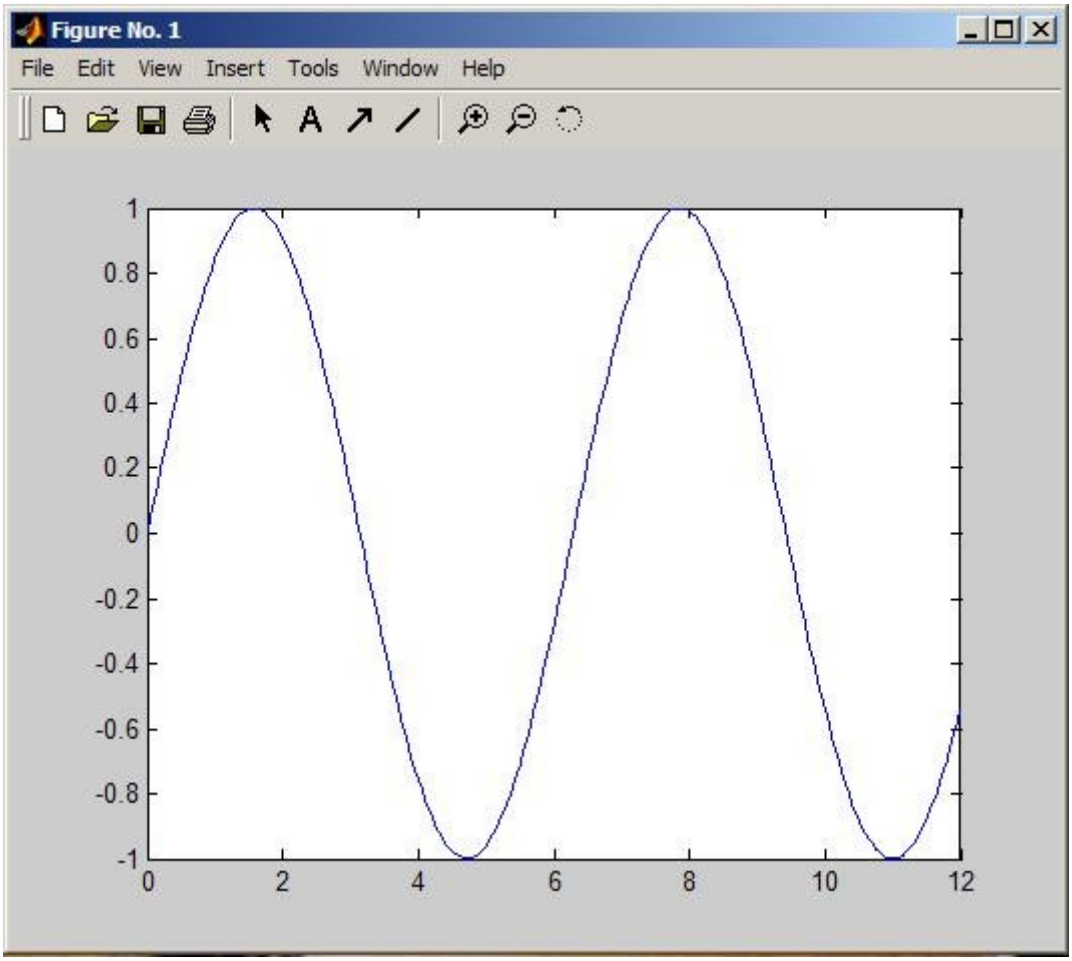
```
>> A(1,:)
ans =
1 2 3 4

>> A(1,1:size(A,2))
ans =
1 2 3 4
```

Tracer le graphe d'une fonction

Un moyen simple de visualiser le graphe d'une fonction de R dans R est d'évaluer cette fonction en un certain nombres de points, et de relier ces points par des segments de droite. Visualisons la fonction sin(x) pour x valant de 0 à 12.

```
>> x = [0:0.05:12];
>> y = sin(x);
>> plot(x,y)
```

A la première ligne, on génère des valeurs de 0 à 12, espacées de 0.05. Ensuite, on évalue la fonction sinus en ces points. Enfin on affiche ces points, reliés par des lignes. Le résultat est:



Note

Si vous voulez de l'aide sur l'opérateur ":", tapez `help colon` essayez aussi : `help linspace` , `help logspace`

Le langage de Matlab n'aurait pas beaucoup d'intérêt s'il n'était pas possible d'effectuer des tests, de programmer des boucles, etc., comme dans la majorité des langages de programmation. L'aide concernant les structures de contrôle peut être accédé via:

```
>> helpwin lang
```

Opérations booléennes

Les valeurs booléennes "vrai" et "faux" sont respectivement représentées dans Matlab par les valeurs 1 et 0. Les opérateurs booléens de Matlab sont décrits à la rubrique d'aide `relop (helpwin relop)`. On a les opérateurs : `>` , `>=` , `<` , `<=` , `==` (égal), `~=` (différent), `&` (et), `|` (ou), `~` (non), `xor` (ou exclusif).

Exemples:

```
>> a = 2; b = 3;
>> a>b
ans =
0

>> a>=a
ans =
1

>> a==a
ans =
1

>> a~=a
ans =
0

>> (b>a) & (a~=4)
ans =
1

>> (b<a) | (a==2)
ans =
1
```

Attention : Lorsque vous écrivez une expression qui combine des opérations arithmétiques et des opérations booléennes, veillez à bien utiliser les parenthèses.

Structure `if`

La structure if est similaire à celle de nombreux langages de programmation (C, C++, Java,...):

```
IF expression1
statements1
ELSEIF expression2
statements2
ELSE
statements3
END
```

Si l'expression `expression1` est vraie, alors les commandes `statements1` sont exécutées, sinon, si l'expression `expression2` est vraie alors les commandes `statements2` sont exécutées, enfin, si ni `expression1` ni `expression2` ne sont vraies, alors ce sont les commandes `statements3` qui sont exécutées. Les clauses `ELSEIF` et `ELSE` sont optionnelles.

Exemple :

```
>> if (a==3) [ENTER]
b = 2 [ENTER]
else [ENTER]
b = 4 [ENTER]
end [ENTER]
b =
4
```

Attention : Une erreur fréquente est d'écrire un THEN après `expression1`. THEN n'existe pas dans Matlab !

Structure `while`

La structure while est une structure puissante permettant de faire de répéter une séquence d'instructions, tant qu'une certaine condition est vérifiée:

```
WHILE expression
statements
END
```

Tant que `expression` est vraie, alors les commandes `statements` sont exécutées. A titre d'exemple, calculons la somme des carrés des entiers de 1 à 11:

```
>> i = 0;
>> somme = 0;
>> while(i<=10)
i=i+1;
somme = somme + (i*i);
end
>> somme
somme =
506
```

On aurait aussi pu se passer de l'utilisation d'une boucle:

```
>> sum([1:11].^2)
```

```
ans =
506
```

Il faut faire attention à ne pas créer de boucle sans fin. Exemple:

```
>> while (i>0)
i = i+1;
end
```

Cette boucle ne s'arrêtera jamais. On peut toujours stopper une commande Matlab en cours d'exécution en tapant : [CTRL] + C (combinaisons des touches Ctrl et C).

Structure `for`

```
FOR variable = expr
statement
END
```

Si `n` est la longueur du vecteur représenté par `expr` , les commandes `statement` seront exécutées `n` fois, lors de la première exécution, `variable` aura la valeur `expr(1)`, ... et ainsi de suite jusqu'à la dernière fois où `variable` aura la valeur `expr(n)`. A titre d'exemple, calculons le produit des entiers impairs de 1 à 11:

```
>> p = 1;
>> for i=[1:2:11]
p = p*i;

end >> p p =
10395
```

Sans utiliser de boucle, on aurait fait:

```
>> prod([1:2:11])
ans =
10395
```

Commande `break`

La commande `break` permet de sortir d'une boucle (for ou while). Exemple :

```
>> i = 0;
>> while (i>=0)
i = i+1;
if (i==10)
break;
end
end
>> i
i =
10
```

Matlab aurait une utilité fort limitée s'il n'était pas possible d'écrire des scripts et des fonctions:

- Script: Un ensemble d'instructions, écrites dans un fichier qu'on peut exécuter en une fois.

- Fonction: Un ensemble d'instructions, prenant certains arguments en entrée et renvoyant un ou plusieurs résultats.

Scripts

Les scripts sont stockés dans des fichiers dont l'extension est ".m". Ils contiennent une suite d'instructions Matlab, écrites comme si on les avait entrées dans la fenêtre de Matlab. Pour exécuter un script, il faut juste taper son nom (sans l'extension ".m") dans la fenêtre de commande de Matlab.

Exemple: monscript.m. A l'exécution, on obtient le résultat suivant:

```
>> monscript
Bonjour, ceci est un script.
Appuyez sur [Espace] pour continuer

A =
1  2  3
4  5  6

B =
3
5
7

A =
1  2  3
4  5  6
0  0  1

C = A*B ,.... voici le résultat

C =
34
79
7
```

Les scripts peuvent accéder aux variables qui ont été définies avant son lancement, et les variables qui ont été définies durant le script restent accessible après par l'utilisateur. Pour plus d'informations : `helpwin script`

Fonctions

Les fonctions sont aussi stockées dans des fichiers ".m" (mfiles). Le fichier doit porter le même nom que la fonction. La syntaxe pour écrire une fonction est la suivante:

```
function [Sortie1, ... , SortieN] = nom_de_la_fonction(Argument1, ... ,
ArgumentP)
instructions
```

Les instructions peuvent utiliser les variables `Argument1, ... ,ArgumentP`, et doivent attribuer une valeur aux variables `Sortie1, ... ,SortieN`.

Exemple : mafonction.m. A l'exécution, on obtient:

```
>> [r1 r2 r3] = mafonction(5,9);
>> r1
r1 =
14
>> r2
r2 =
45
>> r3
r3 =
5
```

```
>> help mafonction

Ceci                              servira                              d'aide.
Si   on   tape   help   mafonction,   ce   sont   les   premières   lignes
de commentaires de la fonction qui sont affichées.

[somme produit minimum] = mafonction(a,b)
renvoie la somme le produit et le minimum de a et b
```

Pour plus d'informations : `helpwin function`

Problèmes de chemin d'accès

Matlab ne pourra exécuter un script ou une fonction que si il sait dans quel répertoires il doit rechercher les mfiles correspondants. Par défaut, Matlab recherche dans certains de ses répertoires, ainsi que dans le répertoire courant. Pour savoir quel est le répertoire courant, utilisez la commande `pwd` (print working directory).

```
>> pwd
ans =
/home/schubert.user1
```

Pour changer de répertoire, utilisez la commande `cd`:

```
>> cd mfiles (pour descendre dans le sous répertoire mfiles)
```

Si vous ne souhaitez pas changer de répertoire, mais ajouter un répertoire à la liste des répertoire dans lesquels Matlab recherche les scripts et fonctions, utilisez la commande `addpath`:

```
>> addpath('./utils') (pour ajouter le sous-répertoire utils)
```

Attention

Il y a un bug (une erreur) dans certaines versions de Matlab: Matlab oublie de rechercher les scripts et fonctions dans le répertoire courant. Pour résoudre ce problème, taper la commande `addpath('./')` pour ajouter explicitement le répertoire courant dans la liste des répertoires explorés. Pour plus d'informations : `helpwin path, helpwin what, ...`

Thèmes choisis

Nous vous conseillons d'ouvrir une fenêtre Matlab et de reproduire les exemples donnés dans chacun des points suivants.

- **Equations différentielles ordinaires**
  Utiliser Matlab pour résoudre numériquement des systèmes d équations différentielles ordinaires.
  Cliquez [ici]
- **Les polynômes**
  Représentation et manipulation des expressions polynômiales
  Cliquez [ici]
- **La vectorisation**
  Pourquoi utiliser des vecteurs plutôt que des boucles
  Cliquez [ici]
- **Les graphes 2D**
  Présentation des fonctions de base permettant de réaliser des graphes en 2 dimensions
  Cliquez [ici]

Introduction

Les modèles mathématiques des sciences et techniques se présentent très souvent sous la forme de systèmes d'équations différentielles ordinaires, qui lient des fonctions (inconnues) du temps à leurs dérivées temporelles. En ajoutant des conditions aux limites à ces équations, on peut alors calculer ces fonctions inconnues. Par exemple, la loi de Newton permet de trouver les équations qui régissent le mouvement en chute libre d'un corps représenté par une masse ponctuelle, dans un champ de gravité constant. Si on connaît la position et la vitesse initiale du corps, on peut alors résoudre les équations du mouvement et obtenir la position et la vitesse du corps comme des fonctions du temps.

Dans de nombreux cas, ces équations sont trop complexes pour pouvoir être résolues de façon analytique et l'unique possibilité est d'essayer d'approximer les fonctions inconnues au moyen de méthodes numériques. L'idée de base consiste à ne chercher la valeur des fonctions qu'en un certain nombre de points : le problème est discrétisé.

Un certain nombre de ces méthodes numériques ont été implémentées dans le logiciel Matlab et permettent de résoudre relativement facilement un grand nombre de problèmes. Ce document constitue une introduction à l'utilisation des méthodes implémentées dans Matlab. Une utilisation efficace et avertie de ces méthodes suppose un minimum de connaissance des concepts sur lesquelles elles se basent.

Solution numérique d'EDO avec Matlab

Pour résoudre numériquement une EDO avec Matlab (ou un système d'EDOs) il faut d'abord faire un peu d'algèbre pour ramener l'équation ou le système d'équations différentiel sous la forme:

$$\frac{du}{dt} = g(t,u)$$

où **u(t)** est le vecteur des fonctions inconnues. Sa dérivée temporelle est exprimée par la fonction g, qui est une fonction du temps et de **u(t)**.

On peut ensuite lancer ODE45 pour trouver la valeur de **u(t)**. Il faut pour cela lui fournir 3 éléments:

- La fonction g(t,u) qui donne la valeur de la dérivée de **u(t)** en fonction de t et de **u**. Cette fonction sera décrite dans un fichier .m
- Un intervalle de temps sur lequel on souhaite connaître la valeur de la fonction inconnue
- Une condition initiale, c'est à dire un point de passage de la fonction inconnue.

Le pourquoi du comment

Cette partie décrit le fonctionnement interne de ODE45. Tout ceci se passe automatiquement à l'intérieur de ODE45. Si ce fonctionnement ne vous intéresse pas, passez cette partie et allez directement voir la partie "Mais encore" ci-dessous.

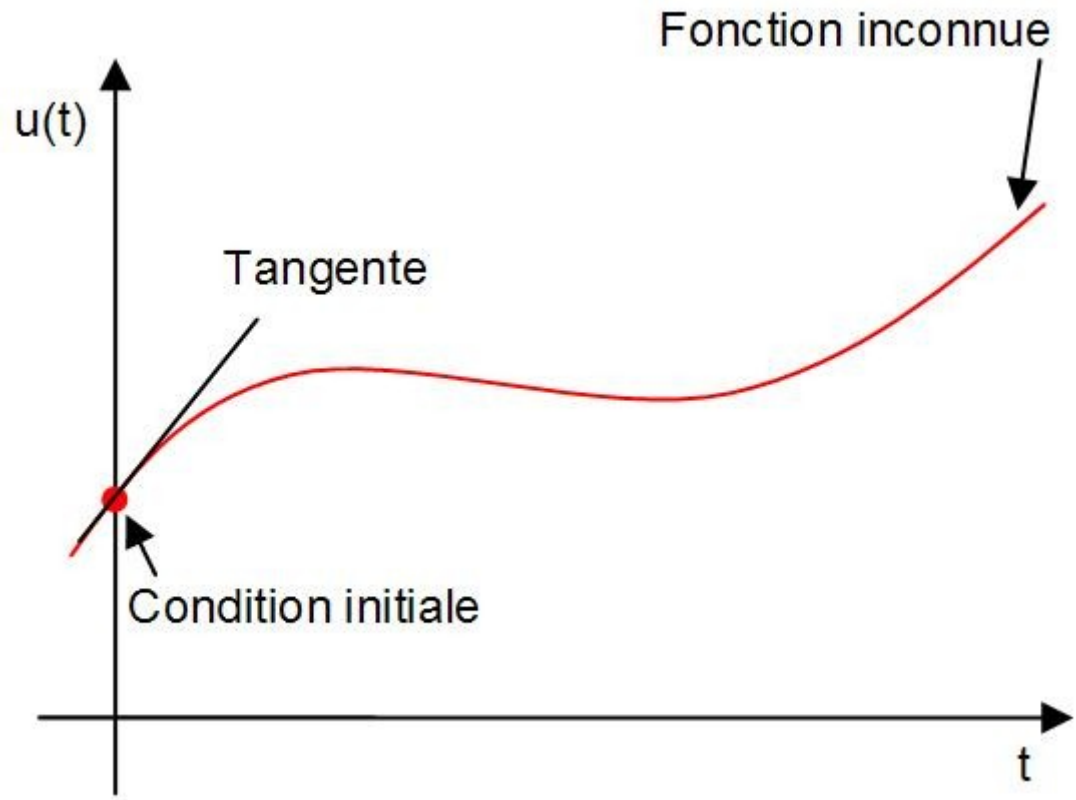
ODE45 va utiliser les éléments que vous lui fournissez pour reconstruire, pas à pas, la fonction inconnue. ODE45 va commencer par découper en morceaux l'intervalle de temps sur lequel vous lui avez demandé la valeur de la fonction **u(t)** (l'explication de ceci va apparaître par la suite). Soit $[T_0\ T_1\ ...\ T_N]$ cet intervalle. Notez que vous ne pouvez contrôler la manière dont ODE45 découpe l'intervalle. Vous ne pouvez que lui fournir les valeurs extrèmes: T_0 et T_N.

L'information qui sera ensuite utilisée par ODE45 est la condition initiale que vous lui avez fournie, c'est-à-dire la valeur **u(0)** de la fonction inconnue au temps $t=T_0$. Connaissant T_0 et $\mathbf{u(t=T_0)}$, et connaissant la fonction g, ODE45 peut calculer la valeur de **u'(t)** au temps $t=T_0$.

Autour de la condition initiale, on peut approximer la fonction inconnue par sa tangente, obtenue à l'aide du point de passage (condition initiale) et de la valeur de la dérivée au point de passage:

$$u'(t=T_0)=g(T_0,u(t=T_0))$$

Ces étapes sont résumées sur la figure suivantes:



La suite de l'histoire est assez simple. ODE45 va calculer la valeur de la fonction inconnue en $t=T_1$ en utilisant 2 approximations différentes de cette fonction inconnue (même principe que celui de la tangente ci-dessus). Si les 2 estimations sont proches, il garde la plus précise. Si ce n'est pas le cas, il choisit un T_1 plus proche de T_0 (on se rappelera que plus on se rapproche de T_0, plus la fonction colle à sa tangente).

Une fois la valeur de **$u(t=T_1)$** connue, on peut itérer. A l'aide de g, on calcule:

$$u'(t=T_1)=g(T_1,u(t=T_1))$$

Ceci nous donne la direction de la fonction inconnue au point de passage connu (T_1,**$u(T_1)$**) et permet de choisir T_2 et ainsi de suite.

Mais encore

On peut se poser toute une série de questions à propos de ODE45. En voici 2 importantes:

- Que fait-on si l'équation à résoudre est d'ordre supérieur à 1 ?
- Quelles sont les sorties fournies par ODE45 ?

Lorsque l'équation à résoudre est d'ordre supérieur à 1, on la transforme en un système de plusieurs équations chacune étant d'ordre 1.

Par exemple :

On désire résoudre numériquement l'équation différentielle décrivant le mouvement d'un pendule amorti (voir cours de mécanique). L'équation différentielle qui donne la position x(t) du pendule au cours du temps est:

$$\ddot{x} = -\frac{\eta}{m}\dot{x} - \frac{k}{m}x$$

Pour rappel, m est la masse du pendule et k le coéfficient de frottement. Cette équation est équivalente au système d'équations suivant:

$$\begin{pmatrix} \dot{u}_1 \\ \dot{u}_2 \end{pmatrix} = \begin{pmatrix} u_2 \\ -\frac{\eta}{m}u_2 - \frac{k}{m}u_1 \end{pmatrix}$$

où on a posé:

$$u = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

Il est à noter que dans ce cas, la condition initiale à fournir sera bien entendu un vecteur puisque nous avons maintenant **2** fonctions inconnues à retrouver. Matlab ne peut savoir qu'elles sont liées entre-elles par une relation de dérivée.

Quelles sont les sorties fournies par ODE45 ?

ODE45 fournit 2 sorties: U et T.

U est la valeur de la fonction inconnue et T est le vecteur contenant les instants T_i auxquels la fonction a été évaluée. T est indispensable puisque les instants T_i ne sont pas équidistants. T est un vecteur colonne.

Il est évident que U sera une matrice dans le cas où on résoud un système plutôt qu'une équation seule. Dans ce cas, la matrice U contient autant de ligne que le vecteur T et chaque colonne correspond aux valeurs d'une des fonctions inconnues.

Un exemple en détail

Reprennons l'exemple du pendule amorti. Le système d'équations à résoudre est décrit ci-dessus. Avant d'exécuter ODE45, il faut écrire la fonction g qui lie u'(t), u(t) et t. C'est chose faite dans ce fichier.

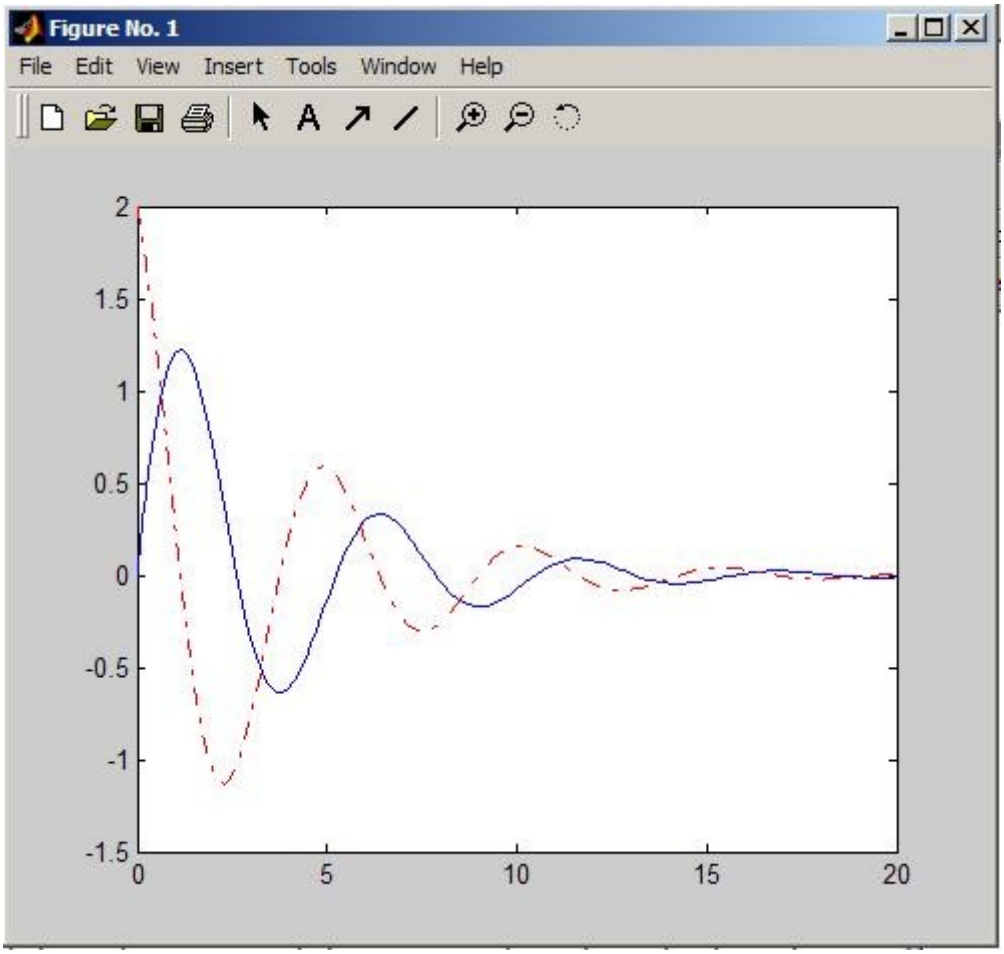
Comme conditions initiales, on va poser qu'en t=0s, le pendule est à sa position d'équilibre (x(0)=0) mais possède une vitesse de 2 m/s. On a donc: `u0 = [0 2]`.

Pour résoudre l'équation différentielle entre de t=0s jusque t=20s on va lancer par exemple:

```
>> [t u] = ode45('oscillateur',[0 20],[0 2]);
```

Si on affiche la position et la vitesse du pendule en fonction du temps, on obtient:

```
>> plot(t,u(:,1));
>> hold on;
>> plot(t,u(:,2),'r-.');
>> xlabel('temps (s)');
>> title('Pendule amorti');
>> legend('Position (m)','Vitesse (m/s)');
```



Beaucoup d'autres détails en utilisant `help ODE45`.

TUTORIEL MATLAB

Auteur: Adrien Leygue - leygue@mema.ucl.ac.be

Introduction

Dans matlab, un polynome est représenté par le vecteur de ses coefficients, commençant par le degré le plus élevé.

Exemple:

Soit le polynome: P(x) = 3x^2-5x+2, il sera représenté par le vecteur P:

```
>> P = [3 -5 2]
P = 3 -5 2
```

Pour évaluer le polynome, on utilise la fonction `polyval`. Pour calculer P(5) :

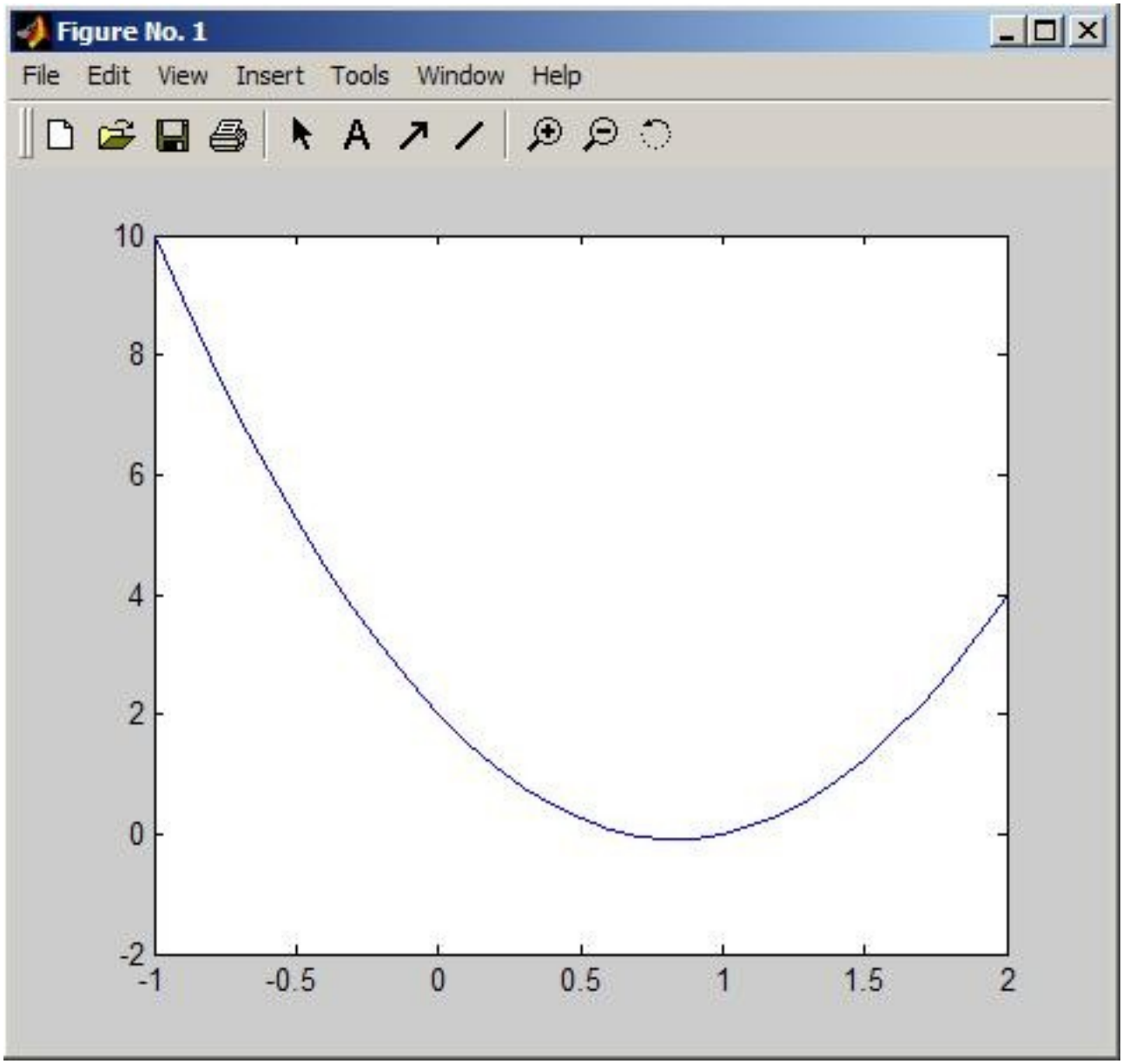
```
>> polyval(P,5)
ans = 52
```

`polyval` accepte aussi des vecteurs de points à évaluer. Dans ce cas, elle renvoit une vecteur de taille identique contenant la valeur du polynome pour chaque valeur du vecteur d'entrée:

```
>> x = [-1:0.1:2];
>> y = polyval(P,x);
```

Cela permet de faire des graphes facilement:

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



On aurait pu écrire directement:

```
>> plot([-1:0.1:2],polyval(p,[-1:0.1:2]))
```

Fonctions avancées

La commande `roots` permet de retrouver les racines du polynome (quelles soient réelles ou complexes):

```
>> racines = roots(p)
racines =
1.0000
0.6667
```

A l'inverse, la fonction `poly` permet de créer un polynome à partir de ces racines:

```
>> P3 = poly([1 -1])
P3 = 1 0 -1
```

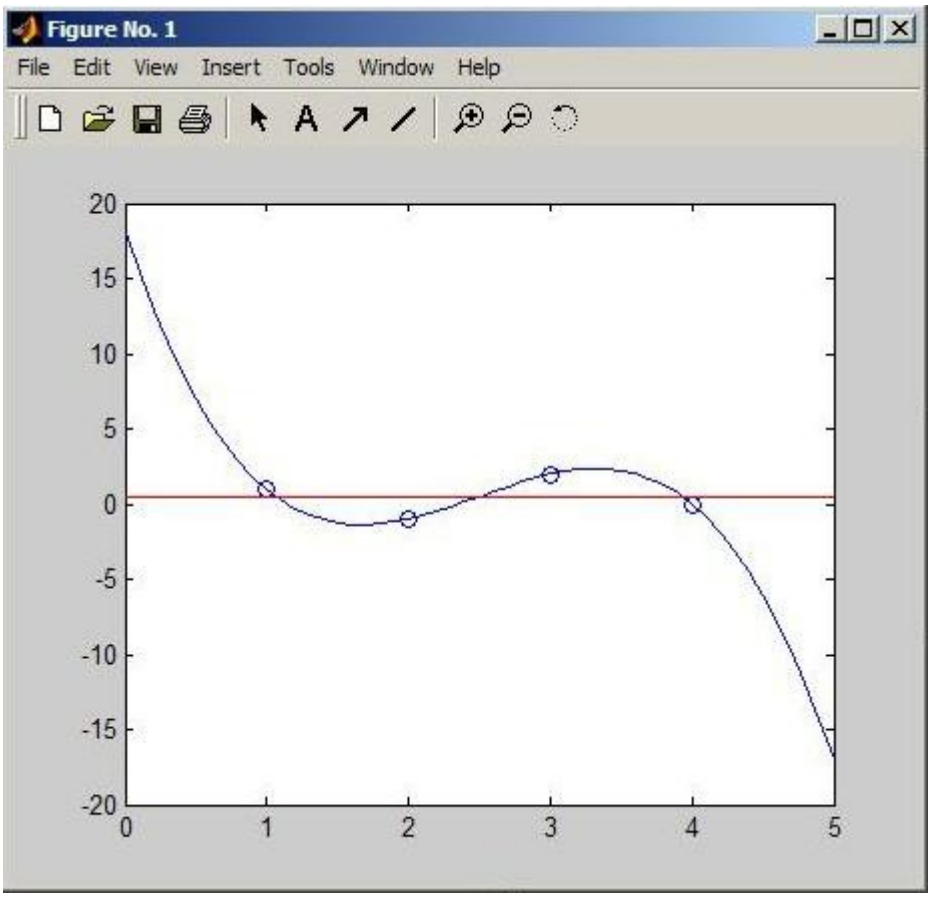
La fonction `polyder` calcule la dérivée d'un polynome. Les fonctions `conv` et `deconv` multiplient ou divisent deux polynomes.

La fonction `polyfit` permet de calculer le polynome d'interpolation passant par un ensemble de points, ou bien de calculer le polynome d'approximation au sens des moindres carrés. On calcule le polynome de degré 3 passant exactement par 4 points définis par x1 et y1:

```
>> x1 = [1 2 3 4];
>> y1 = [1 -1 2 0];
>> P4 = polyfit(x1,y1,3)
P4 = -1.6667 12.5000 -27.8333 18.0000
```

On peut également calculer la droite (polynome de degré 1) qui approxime le mieux les points (au sens des moindres carrés):

```
>> P5 = polyfit(x1,y1,1);
>> plot(x1,y1,'o');
>> hold on;
>> plot([0:0.1:5],polyval(P4,[0:0.1:5]));
>> hold on;
>> plot([0:0.1:5],polyval(P5,[0:0.1:5]),'r');
```



Pour plus d'informations:

```
>> help polyfun
>> helpwin polyfun
```

Avec Matlab, il est avantageusement possible d'éviter la programmation de nombreuses boucles en utilisant des notations et des fonctions vectorisées.

Exemple 1

On souhaite élever toutes les valeurs d'un vecteur `a = [1:20]` au carré.

Non vectorisé:

```
>> b1 = zeros(size(a));
>> for i=1:length(a), b1(i) = a(i)^2; end
```

Vectorisé:

```
>> b2 = a.^2;
```

La notation .^2 signifie qu'on applique l'opération ^2 élément par élément à la matrice/vecteur `a`. Non seulement c'est plus concis et plus lisible, mais cette approche est aussi plus rapide. En effet Matlab utilise des routines pré-compilées pour effectuer les opérations vectorisées. Mesurons le temps de calcul pour un gros vecteur.

Note: La commande `tic` lance le chronomètre, la commande `toc` le stoppe et renvoie le temps écoulé depuis tic.

Non vectorisé:

```
>> a = [1:100000];
>> b1 = zeros(size(a));
>> tic; for i=1:length(a), b1(i) = a(i)^2; end; toc

elapsed_time =

  1.0620
```

Vectorisé:

```
>> tic; b2 = a.^2; toc

elapsed_time =

  0.0600
```

La ligne `b1 = zeros(size(a))` n'est pas inutile: avec cette ligne on alloue a priori la mémoire dont on aura besoin pour stocker la matrice `b1`. De cette façon Matlab n'est pas obligé, à chaque itération, de modifier la taille de b1 quand on rajoute un nouvel élément. Si on n'avait pas alloué à priori la matrice `b1`, on aurait obtenu des performances encore moins bonnes:

```
>> a = [1:100000];
>> tic; for i=1:length(a), b3(i) = a(i)^2; end; toc

elapsed_time =

  400
```

Vous voilà convaincu (j'espère) qu'il vaut mieux éviter les boucles FOR (ou WHILE), avec Matlab.

Exemple 2

Faire la combinaison linéaire de 2 lignes d'une matrice `A = rand(10,20)`. On veut additionner 3 fois la première ligne à la deuxième ligne, sauf le premier et le dernier élément de la ligne.

Sans vectorisation
```
>> [l,c] = size(A);
>> for(i=2:c-1), A(2,i) = A(2,i)+3*A(1,i); end;
```

Avec vectorisation:
```
>> A(2,2:c-1) = A(2,2:c-1) + 3*A(1,2:c-1);
```

Exemple 3

A chaque élément d'un vecteur on additionne celui qui le précède dans le vecteur et celui sui le suit, sauf pour le premier élément pour lequel on ne considère que l'élément qui le suit, et le dernier element, pour lequel on ne considère que l'avant-dernier élément.

Exemple:
```
[1 2 3 4] --> [(1+2) (1+2+3) (2+3+4) (3+4)]
```

Non vectorisé (il vaut mieux placer les instructions dans un script):

```
a      =      [1      2      3      4      5      6      7      8      9];
prec                                  =                                 0;
for                  i                   =                  1:(length(a)-1)
  nouveau                           =                        a(i)+prec+a(i+1);
  prec                                 =                              a(i);
  a(i)                               =                            nouveau;
end
a(i+1)                              =                          a(i+1)+prec;
a
```

Avec vectorisation:
```
>>      a      =      [1      2      3      4      4      5      6      7      8];
>>    a    =    a    +    [a(2:lenght(a))    0]    +    [0    a(1:lenght(a)-1)]
```

La commande `plot` permet d'afficher un certain nombre de points en 2D. Suivant les options, ces points sont soit représentés par un symbole, soit reliés par une ligne. Il vous est proposé d'essayer les lignes de commande suivantes:

```
>> x = [1 2 2.5 2 3];
>> y = [2 1 3 4 0];
>> plot(x,y);
>> title('Trait plein');
>> figure
>> plot(x,y,'r:');
>> title('Trait pointillé rouge');
>> figure;
>> plot(x,y,'o');
>> title('Points isolés');
```

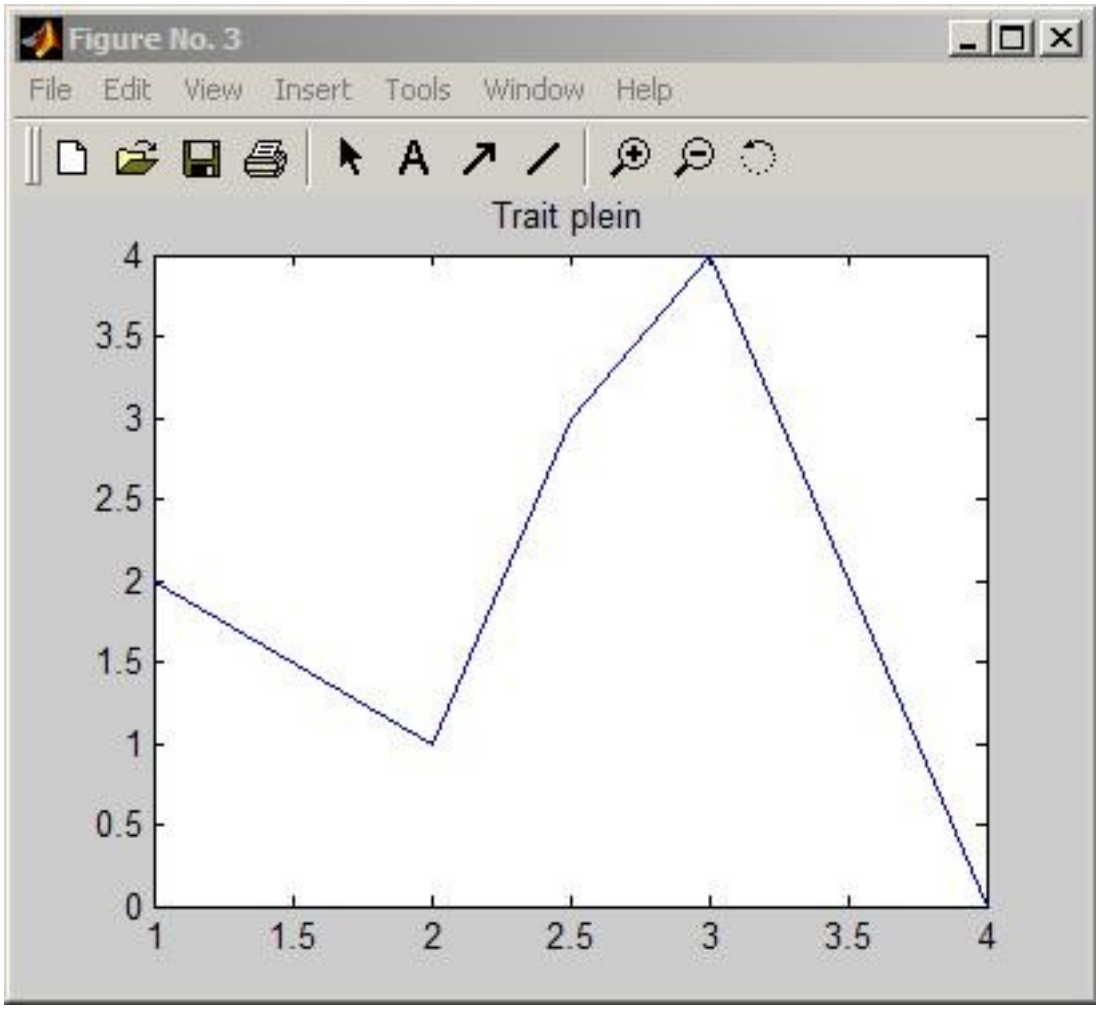


Figure No. 3
File Edit View Insert Tools Window Help
Trait plein
4
3.5
3
2.5
2
1.5
1
0.5
0
1
1.5
2
2.5
3
3.5
4

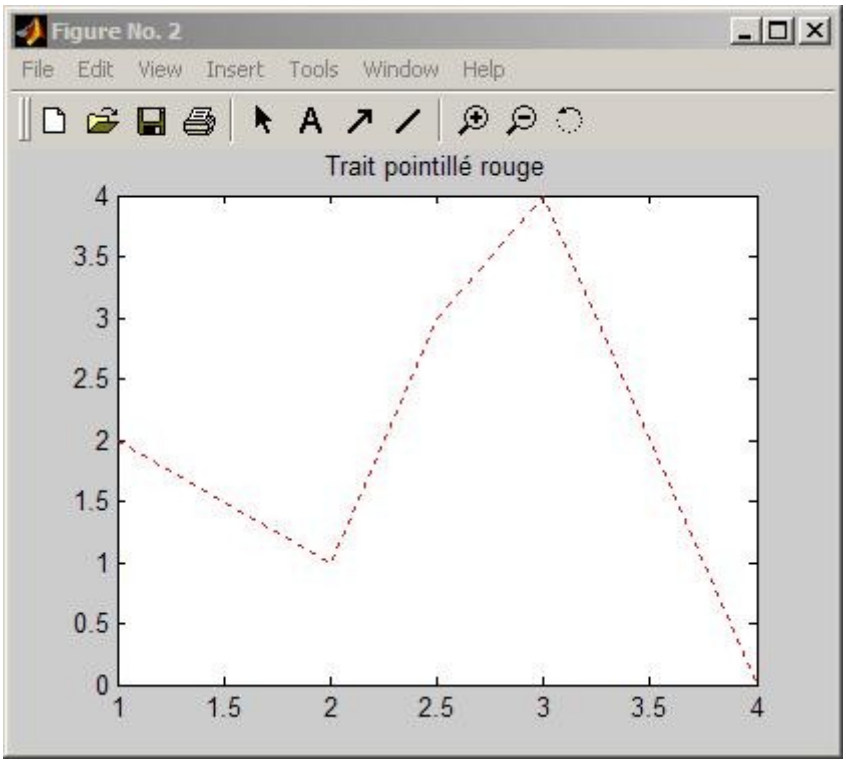
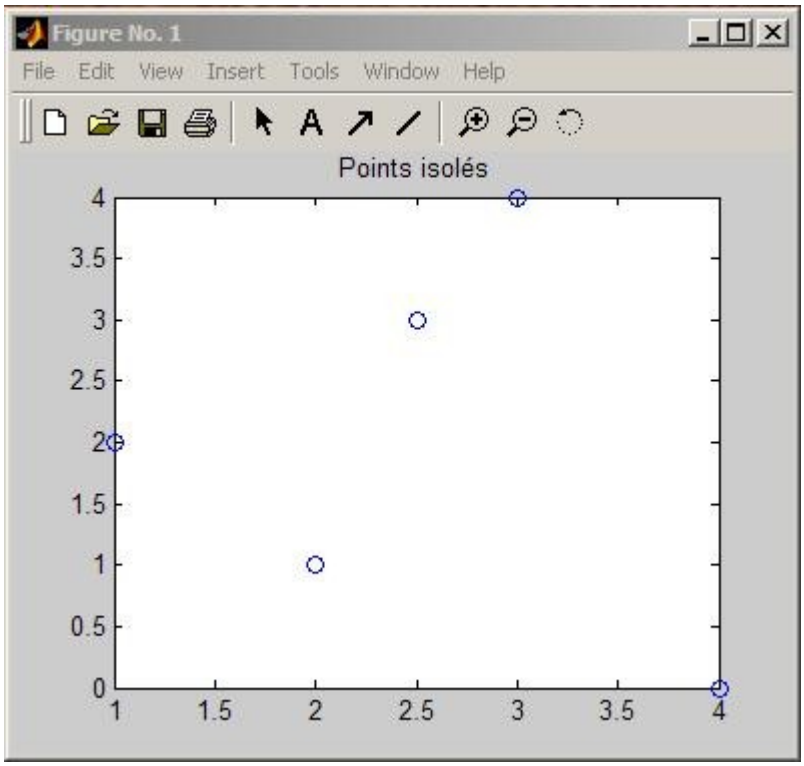


Figure No. 2
File Edit View Insert Tools Window Help
Trait pointillé rouge
4
3.5
3
2.5
2
1.5
1
0.5
0
1
1.5
2
2.5
3
3.5
4



Comme vous avez pu le constater, la commande `figure` fait apparaître une nouvelle fenêtre dans laquelle le prochain graphe sera dessiné. L'appel à la fonction `plot`, non précédé de cette commande, efface le graphe actuel pour dessiner le nouveau. Pour faire apparître 2 courbes sur le même graphe, il faut remplacer, dans la série de commandes ci-dessus, la commande `figure` par la commande `hold on`.

On peut controler les fenêtres en sauvant la valeur renvoyée par la commande figure:

```
>> f1 = figure;
>> plot(x,y,'o');
>> f2 = figure;
>> plot(x,y,'rx');
```

On peut alors facilement fermer la deuxième fenêtre:

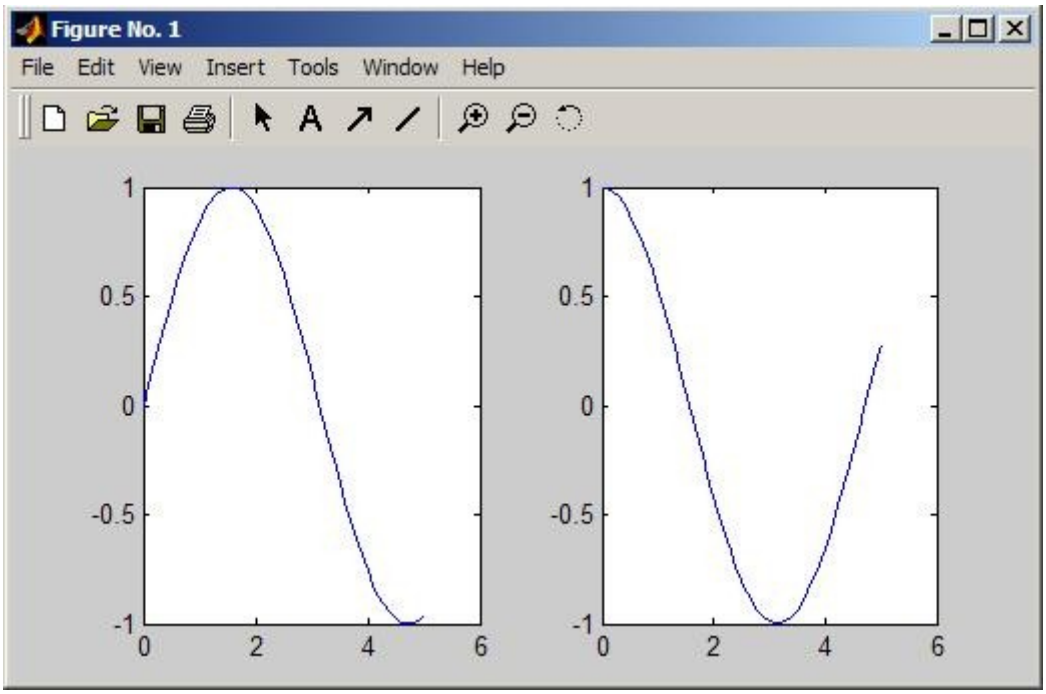
```
>> close(f2);
```

La commande `close all` ferme toutes les fenetres graphiques.

La commande `title` (voir ci-dessus), permet d'ajouter un titre au graphe. De même, les commandes `xlabel` et `ylabel` permettent d'ajouter des intitulés aux axes. On peut également modifier les limites des axes au moyen de `axis`.

La commande `subplot` permet d'afficher plusieurs graphes dans une même fenêtre.

```
>> x=[0:0.1:5];
>> subplot(1,2,1);
>> plot(x,sin(x);
>> subplot(1,2,2);
>> plot(x,cos(x));
```



On peut aussi réaliser des diagrammes logarithmiques `loglog` ou semi-logarithmiques `semilogx` et `semilogy`. On peut tracer des diagrammes de Bode `bode` ou de Nyquist `nyquist`.

Pour plus d'informations: `help plot`

Exercice 1

Écrire une fonction "monexp" qui calcule l'exponentielle d'un réel à partir du développement de Taylor de la fonction exponentielle.

Exercice 2

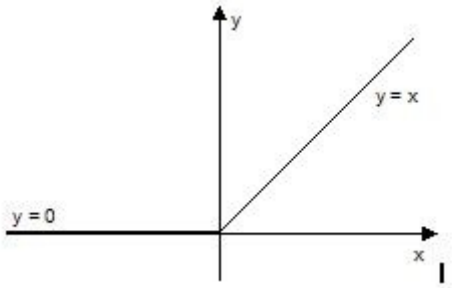
Ecrire une fonction "fibo" qui prend comme arguument un naturel "n" et qui calcule les n premiers termes de la suite de Fibonacci. Cette suite F(n) est définie comme: F(n) = F(n-1)+F(n-2), avec F(1)=F(2)=1.

Exercice 3

Ecrire une fonction "monsort" qui prend un vecteur de nombres réels comme argument et qui renvoie, comme résultat, ce même vecteur avec les éléments triés par ordre croissant. Comparez votre programme (résultats et performances) avec la fonction `sort` de Matlab.

Exercice 4

Construisez la fonction y = brol(x) dont le graphe est représenté ci-dessous.



Exercice 5

Construisez une fonction `y = ecrire(x,n)` qui sauve dans un fichier les n premières puissance de x. Le fichier doit contenir un élément par ligne.

Exercice 6

Construisez la fonction `[A,B,C] = test_divise(x,y)` telle que:

- A = 1 si x est divisible par y et 0 sinon
- B est le résultat de la division entière de y par x
- C est le reste de la division entière de y par x

Exemple:
[A,B,C] = test_divise(10,6) fourni:
A = 0 ; B = 1 ; C = 4

Exercice 7

Utilisez la fonction réalisée en 3. pour réaliser une fonction `[y] = banque(x)` qui vérifie si x est un numéro de compte bancaire valide. Un numéro de compte bancaire est valide si les 2 derniers chiffres sont le reste de la division entière des 10 premiers par 97.

Exercice 8

Ecrivez un script qui représente sur un même graphique, les fonctions sin(x), cos(x), $\sin^2(x)$, $\sin(x^2)$ dans des couleurs différentes.

Exercice 9

Ecrivez un script qui lit au clavier un nombre entier n et fourni une matrice n x n dont les éléments sont donnés par: a(i,j) = i + j.

Exercice 10

Ecrivez une fonction [y] = compare(x,y) qui prend 2 chaînes de caractères comme arguments et renvoi 1 sur x est avant y dans l'ordre alphabétique et 0 sinon.