Introduction à Matlab

Ce poly constitue une introduction à l'utilisation de la plate-forme Matlab.

Prérequis:
Savoir naviguer sur Internet, dans les dossiers gérés par le système d'exploitation Windows. Les étudiants devront se munir d'une disquette personnelle.

Objectifs:
- acquérir un certain "vocabulaire"
- acquérir une certaine "grammaire"
- acquérir une "méthodologie" dans la conception des programmes

Contrôle des Connaissances:
Un ou des exercices similaires à ceux énoncés dans la suite sera tiré au sort et proposé(s) à chaque étudiant dans le cadre des examen de sessions I et II.

note: ce poly a été écrit en utilisant la version 7 de MATLAB.

***

SOMMAIRE

MATLAB (MATrix LABoratory) comprend de nombreuses fonctions graphiques, un système puissant d'opérateurs s'appliquant à des matrices, des algorithmes numériques, ainsi qu'un langage de programmation extrêmement simple à utiliser. L'aspect modulaire est l'un des plus grands atouts de MATLAB : l'utilisateur peut lui-même définir ses propres fonctions, en regroupant des instructions MATLAB dans un fichier portant le suffixe ".m". La syntaxe est bien plus abordable que dans les langages classiques et devrait éliminer les réticences habituelles des programmeurs débutants pour écrire des fonctions. En termes de vitesse d'exécution, les performances sont inférieures à celles obtenues avec un langage de programmation classique. L'emploi de MATLAB devrait donc être restreinte à des problèmes peu gourmands en temps calcul, mais dans la plupart des cas, il présente une solution élégante et rapide à mettre en oeuvre.

1. Généralités et prise en main

1.1. Démarrage, quitter

Pour lancer le programme, double-cliquez l'icône **matlab** sur le bureau. Une fenêtre logo fait une brève apparition, puis dans la fenêtre de commande, le symbole ">>" apparaît : c'est l'invite de MATLAB qui attend vos commandes. Vous pourrez quitter la session avec la commande **quit**.

1.2. Aide, documentation en ligne, configuration du bureau

L'aide en ligne peut être obtenue directement dans la session en tapant **help nom de commande**.

1.3. Calculs élémentaires, ans, format

Commençons par les opérateurs les plus courants : +, -, *, /, ^. Le dernier signifie "puissance". Les parenthèses s'utilisent de manière classique. Nous avons tout pour effectuer un premier calcul : tapez une expression mathématique quelconque et appuyez sur <Entrée>. Par exemple :

>> **(3*2)/(5+3)**
ans =
0.7500

Le résultat est mis automatiquement dans une variable appelée **ans** (answer). Celle-ci peut être utilisée pour le calcul suivant, par exemple :

>> **ans*2**
ans =
1.5000

Ensuite, vous remarquerez que le résultat est affiché avec 5 chiffres significatifs, ce qui ne signifie pas que les calculs soient faits avec aussi peu de précision. Si vous voulez afficher les nombres avec plus de précision, tapez la commande **format long**. Pour revenir au comportement initial : **format short**.

1.4. Historique des commandes

Toutes les commandes que vous aurez tapé sous MATLAB peuvent être retrouvées et éditées grâce aux touches de direction. Appuyez sur ↑ pour remonter dans les commandes précédentes, ↓ pour redescendre et utilisez → et ← pour éditer une commande. Pour relancer une commande, inutile de remettre le curseur à la fin, vous appuyez directement sur la touche <Entrée>. Encore plus fort : vous pouvez retrouver toutes les commandes commençant par un groupe de lettres. Par exemple pour retrouver toutes les commandes commençant par <plot>, tapez plot, puis appuyez plusieurs fois sur ↑. Expérimenter cela dés maintenant.

1.5. Fichiers de commandes (scripts)

1.5.1. Principe général

Le principe est simple : regrouper dans un fichier une série de commandes MATLAB et les exécuter en bloc. Tout se passera comme si vous les tapiez au fur et à mesure dans une session MATLAB. Il est fortement conseillé de procéder comme ceci. Cela permet notamment de récupérer facilement votre travail de la veille.
Les fichiers de commandes peuvent porter un nom quelconque, mais doivent finir par l'extension ".m".
Pour créer un fichier de commandes, taper des commandes MATLAB à l'intérieur de l'éditeur intégré, sauver par exemple sous le nom "toto.m", puis sous MATLAB, tapez :
>> **toto**
Toutes les commandes du fichier seront exécutées en bloc.

1.5.2. Où doit se trouver mon fichier de commande ?

Créer votre répertoire personnel sur le disque dur avec le navigateur Windows. Utiliser votre nom pour le nom du répertoire. Pour que Matlab utilise les scripts qui y seront placés, modifier le répertoire courant de Matlab pour l'assigner à celui que vous venez de créer : barre des menus "current repertory". Tous les scripts qui seront écrits dans la suite devront être sauvés dans ce répertoire ou éventuellement sur une disquette avec un nom à votre convenance et suffisamment explicite.

1.5.3. Commentaires et auto-documentation

Les lignes blanches sont considérées comme des commentaires. Tout ce qui se trouve après le symbole "%" sera également considéré comme un commentaire.
Il est également possible de documenter ses fichiers de commande. Ainsi, définissez une série de lignes de commentaires ininterrompues au début de votre fichier **toto.m**. Lorsque vous taperez :
>> **help toto**
ces lignes de commentaires apparaîtront à l'écran. C'est ainsi que fonctionne l'aide en ligne de MATLAB. C'est tellement simple que ce serait dommage de s'en priver.

1.5.4. Suppression de l'affichage

Jusqu' à présent, nous avons vu que toutes les commandes tapées sous MATLAB affichaient le résultat. Pour certaines commandes (création de gros tableaux), cela peut s'avérer fastidieux. On peut donc placer le caractère "**;**" à la fin d'une ligne de commande pour indiquer à MATLAB qu'il ne doit pas afficher le résultat.

1.5.5. Pause dans l'exécution

Si vous entrez la commande **pause** dans un fichier de commandes, le programme s'arrêtera à cette ligne tant que vous n'avez pas tapé < Entrée>.

1. Premier programme sous Matlab
*Ecrire l'exemple suivant, le sauver sous le nom '**premierprog**' et l'exécuter à partir de la fenêtre de commande.*

```
% premier programme sous Matlab
disp('Ceci est mon premier programme')
disp(' Il affiche à l''écran les 10 premiers entiers')
for i=1:10
i
end
% fin du programme
```

*Modifier le programme en ajoutant un '**;**' aprés le '**i**'. Que donne maintenant son execution?*

2. *Input et format*

Ecrire un script qui invite l'utilisateur à entrer un nombre, l'affiche en format court simple puis court avec puissance de 10. Indices : ***input****,* ***format****. La précision dans un calcul avec chacun de ces nombres est-elle affectée?*

3. *Addpath (ajoute un chemin)*

Ecrire un programme simple et le sauver dans un répertoire qui ne soit pas le répertoire courant. Tenter de l'exécuter en l'appelant de la fenêtre de commandes. Utiliser la commande ***addpath*** *pour rendre l'appel effectif.*

2. Variables et fonctions prédéfinies

2.1. Variables

Le calcul effectué plus haut n'a guère d'intérêt en soi. Il est bien sûr possible de conserver un résultat de calcul et de le stocker dans des variables. Gros avantage sur les langages classiques : on ne déclare pas les variables. Leur type (entier, réel, complexe) s'affectera automatiquement en fonction du calcul effectué. Pour affecter une variable, on dit simplement à quoi elle est égale. Exemple :

>> **a=1.2**

a =

1.2000

On peut maintenant inclure cette variable dans de nouvelles expressions mathématiques, pour en définir une nouvelle :

>> **b = 5*a^2+a**

b =

8.4000

et ensuite utiliser ces deux variables :

>> **c = a^2 + b^3/2**

c =

297.7920

J'ai maintenant trois variables a, b et c. Ces variables ne sont pas affichées en permanence à l'écran. Mais pour voir le contenu d'une variable, rien de plus simple, on tape son nom :

>> **b**

b =

8.4000

ou on double-click sur son nom dans le workspace.

On peut aussi faire des calculs en complexe $\sqrt{-1}$ s'écrit indifféremment **i** ou **j**, donc pour définir un complexe :

>> **a+ b*i**

ans =

1.2000 + 8.4000i

Le symbole * peut être omis si la partie imaginaire est une constante numérique. Tous les opérateurs précédents fonctionnent en complexe. Par exemple :

>> **(a+b*i)^2**

ans =

-69.1200 +20.1600i

Un dernier point sur les variables :

- MATLAB fait la différence entre les minuscules et les majuscules.
- Les noms de variables peuvent avoir une longueur quelconque.
- Les noms de variables doivent commencer par une lettre.

2.2. Effacement des variables

La commande **clear** permet d'effacer une partie ou toutes les variables définies jusqu' à présent.
Syntaxe :
clear a b c . . .
Si aucune variable n'est spécifiée, toutes les variables seront effacées.

2.3. Variables prédéfinies

Il existe un certain nombre de variables pré-existantes. Nous avons déjà vu **ans** qui contient le dernier résultat de calcul, ainsi que **i** et **j** qui représentent $\sqrt{-1}$.
Il existe aussi **pi**, qui représente π, et quelques autres. Retenez que **eps**, nom que l'on a souvent tendance à utiliser est une variable prédéfinie. ATTENTION : ces variables ne sont pas protégées, donc si vous les affectez, elles ne gardent pas leur valeur initiale. C'est souvent le problème pour **i** et **j** que l'on utilise souvent spontanément comme indices de boucles, de telle sorte qu'on ne peut plus ensuite définir de complexe.
Quelle est la valeur de **eps** ?

2.4. Fonctions prédéfinies

Toutes les fonctions courantes et beaucoup parmi les moins courantes existent. La plupart d'entre elles fonctionnent en complexe. On retiendra que pour appliquer une fonction à une valeur, il faut mettre cette dernière entre parenthèses. Exemple :
>> **sin(pi/12)**
ans =
0.16589613269342
Voici une liste non exhaustive :
- fonctions trigonométriques et inverses : **sin, cos, tan, asin, acos, atan**
- fonctions hyperboliques (on rajoute "h") : **sinh, cosh, tanh, asinh, acosh, atanh**
- racine, logarithmes et exponentielles : **sqrt, log, log10, exp**
- fonctions erreur : **erf, erfc**
- fonctions de Bessel et Hankel : **besselj, bessely, besseli, besselk, besselh**.

4. <u>Angle et abs</u>
*Chercher la syntaxe utilisée pour les fonctions **angle** et **abs**. Déterminer alors, la phase et le module du nombre complexe 1+4*j.*

5. <u>Type</u>
*Essayer type **premierprog**, puis type **sin** et encore type **angle**. Que fait la commande **type**?*

3. Matrices et tableaux

En dépit du titre de cette section, MATLAB ne fait pas de différence entre les deux. Le concept de tableau est important car il est à la base du graphique : typiquement pour une courbe de n points, on définira un tableau de n abscisses et un tableau de n ordonnées. Mais on peut aussi définir des tableaux rectangulaires à deux indices pour définir des matrices au sens mathématique du terme, puis effectuer des opérations sur ces matrices.

3.1. Définition d'un tableau

On utilise les crochets [et] pour définir le début et la fin de la matrice. Ainsi pour définir une variable M contenant la matrice $\begin{bmatrix} 1 & 2 & 3 \\ 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$, on écrira :

>> **M = [1,2,3;11,12,13;21,22,23]**
M =
  1   2   3
11 12 13
21 22 23
On utilise le symbole "," qui sert de séparateur de colonne et ";" de séparateur de ligne. On peut aussi définir des vecteurs, ligne ou colonne, à partir de cette syntaxe (un espace est interprété comme une virgule). Par exemple :
>> **U = [1 2 3]**
U =
1 2 3
définit un vecteur ligne, alors que (une commande "entrée" est interprétée comme un poit-virgule):
>> **V = [11**
12
13]
V =
11
12
13
définit un vecteur colonne. Le passage d'une ligne de commande à la suivante s'effectuant par la frappe de la touche **<**Entrée**>**.
On aurait pu aussi définir ce dernier par :
>> **V=[11;12;13]**

3.2. Accès à un élément d'un tableau

Il suffit d'entrer le nom du tableau suivi entre parenthèses du ou des indices dont on veut lire ou écrire la valeur. Exemple si je veux la valeur de M, ligne 3, colonne 2 :
>> **M(3,2)**
ans =
22
Pour modifier seulement un élément d'un tableau, on utilise le même principe. Par exemple, je veux que M_{32} soit égal à 32 au lieu de 22 :
>> **M(3,2)=32**
M =
1   2   3
11 12 13
21 32 23
Vous remarquerez que MATLAB réaffiche du coup toute la matrice, en prenant en compte la modification. On peut se demander ce qui se passe si on affecte la composante d'une matrice qui n'existe pas encore. Exemple :
>> **P(2,3) = 3**
P =
0 0 0
0 0 3
Voilà la réponse : MATLAB construit automatiquement un tableau suffisamment grand pour arriver jusqu'aux indices spécifiés, et met des zéros partout sauf au terme considéré. Vous remarquerez que contrairement aux langages classiques, inutile de dimensionner les tableaux à l'avance : ils se construisent au fur et à mesure.

3.3. Extraction de sous-tableaux

Il est souvent utile d'extraire des blocs d'un tableau existant. Pour cela on utilise le caractère ":". 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 indices) :

tableau(début : fin, début : fin)

Ainsi pour extraire le bloc $\begin{bmatrix} 2 & 3 \\ 12 & 13 \end{bmatrix}$
on tapera :

>> **M(1:2,2:3)**
ans =
 2 3
12 13

On utilise le caractère ":" seul pour prendre tous les indices possibles. Exemple :

>> **M(1:2,:)**
ans =
 1 2 3
11 12 13

C'est bien pratique pour extraire des lignes ou des colonnes d'une matrice. Par exemple pour obtenir la deuxième ligne de M :

>> **M(2,:)**
ans =
11 12 13

3.4. Construction de tableaux par blocs

Vous connaissez ce principe en mathématiques. Par exemple, à partir des matrices et vecteurs précédemment définis, on veut définir la matrice

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

qui est une matrice 4x4. Pour faire cela sous MATLAB, on fait comme si les blocs étaient des scalaires, et on écrit tout simplement :

>> **N=[M V**
U 0]
N =
   1   2   3 11
  11 12 13 12
  21 32 23 13
   1   2   3   0

ou bien en utilisant les caractères "**,**" et "**;**"

>> **N=[M, V; U, 0]**

Cette syntaxe est très utilisée pour allonger des vecteurs ou des matrices, par exemple si je veux ajouter une colonne à M, constituée par V :

>> **M=[M V]**
M =
1   2   3 11
1 12 13 12
21 32 23 13

Si je veux lui ajouter une ligne, constituée par U :
>> **M = [M;U]**
M =
1 2 3
11 12 13
21 32 23
1 2 3

6. Matrice et extraction
Ecrire un script qui définisse la matrice M suivante et affiche la dernière ligne:

1	*2*	*3*	*4*	*5*
9	*10*	*11*	*12*	*13*
-1	*-2*	*-3*	*-4*	*-5*

3.5. Opérations sur les tableaux

3.5.1. Addition et soustraction

Les deux opérateurs sont les mêmes que pour les scalaires. A partir du moment où les deux tableaux concernés ont la même taille, le tableau résultant est obtenu en ajoutant ou soustrayant les termes de chaque tableau.

7. Matrice d'éléments aléatoires
Déterminer la syntaxe de la fonction **rand**, créer 2 matrices A et B de taille 5X6, à valeurs aléatoires comprises entre -10 et 10. Afficher C= A+B

3.5.2. Multiplication, division et puissance terme à terme

Ces opérateurs sont notés ".*", "./" et ".^" (attention à ne pas oublier le point). Ils sont prévus pour effectuer des opérations termes à terme sur deux tableaux de même taille. Ces symboles sont importants lorsque l'on veut tracer des courbes.

3.5.3. Multiplication, division et puissance au sens matriciel

Puisque l'on veut manipuler des matrices, il paraît intéressant de disposer d'un multiplication matricielle. Celle-ci se note simplement * et ne doit pas être confondue avec la multiplication terme à terme. Il va de soi que si l'on écrit A*B le nombre de colonnes de A doit être égal au nombre de lignes de B pour que la multiplication fonctionne.
La division a un sens vis- à-vis des inverses de matrices. Ainsi A/B représente A multiplié (au sens des matrices) par la matrice inverse de B. Il existe aussi une division à gauche qui se note \. Ainsi A\B signifie l'inverse de A multiplié par B. Ce symbole peut être aussi utilisé pour résoudre des systèmes linéaires : si v est un vecteur, A\v représente mathématiquement $A^{-1}v$, c'est- à-dire la solution du système linéaire Ax = v.
La puissance n-ième d'une matrice représente cette matrice multipliée n fois au sens des matrices par elle-même.
Pour bien montrer la différence entre les opérateurs ".*" et "*", un petit exemple faisant intervenir la matrice identité multipliée par la matrice $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
Voici la multiplication au sens des matrices :
>> **[1 0; 0 1] * [1 2; 3 4]**
ans =
1 2
3 4

et maintenant la multiplication terme à terme :
>> **[1 0; 0 1] .* [1 2; 3 4]**
ans =
1 0
0 4

8. Opérations sur une matrice
a=[1,4;6,8]; *Est-il possible d'appliquer la fonction sin à la matrice a ? Quel est le résultat ? Ecrire un script qui affiche les résultats successifs de sin(a).^n avec n variant de 1 à 10 et utilisant une boucle* ***for...end****.*

3.5.4. Transposition

L'opérateur transposition est le caractère ' (prime) et est souvent utilisé pour transformer des vecteurs lignes en vecteurs colonnes et inversement.

3.6. Longueurs de tableau

La fonction **size** appliquée à une matrice renvoie un tableau de deux entiers : le premier est le nombre de lignes, le second le nombre de colonnes. La commande fonctionne aussi sur les vecteurs et renvoie 1 pour le nombre de lignes (resp. colonnes) d'un vecteur ligne (resp colonne).
Pour les vecteurs, la commande **length** est plus pratique et renvoie le nombre de composantes du vecteur, qu'il soit ligne ou colonne.

3.7. Génération rapide de tableaux

3.7.1. Matrices classiques

On peut définir des matrices de taille donnée ne contenant que des 0 avec la fonction **zeros**, ou ne contenant que des 1 avec la fonction **ones**. Il faut spécifier le nombre de lignes et le nombre de colonnes. Voici deux exemples :
>> **ones(2,3)**
ans =
1 1 1
1 1 1
>> **zeros(1,3)**
ans =
0 0 0
L'identité est obtenue avec **eye**. On spécifie seulement la dimension de la matrice (qui est carrée. . .)
>> **eye(3)**
ans =
1 0 0
0 1 0
0 0 1
Il existe également une fonction **diag** permettant de créer des matrices diagonales (voir l'aide intégrée).

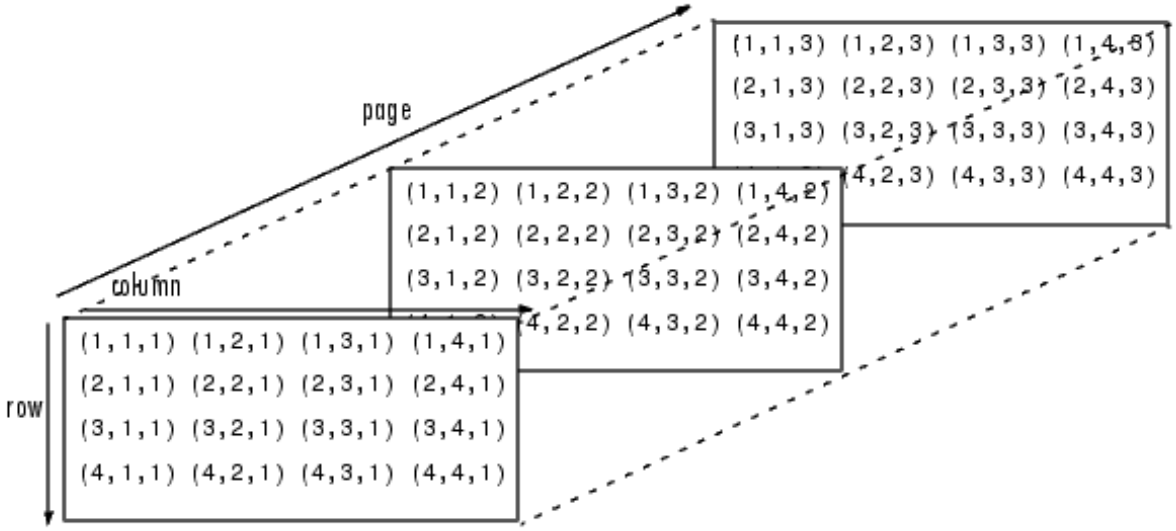
3.7.2. Listes de valeurs

Cette notion est capitale pour la construction de courbes. Il s'agit de générer dans un vecteur une liste de valeurs équidistantes entre deux valeurs extrêmes. La syntaxe générale est :
variable = valeur début: pas: valeur fin
Cette syntaxe crée toujours un vecteur ligne. Par exemple pour créer un vecteur x de valeurs équidistantes de 0.1 entre 0 et 1 :
>> **x = 0:0.1:1**

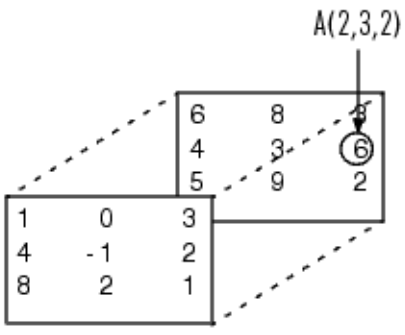
x =
Columns 1 through 7
0 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000
Columns 8 through 11
0.7000 0.8000 0.9000 1.0000
Il est conseillé de mettre un point-virgule à la fin de ce type d'instruction pour éviter l'affichage fastidieux du résultat.
Autre exemple pour créer 101 valeurs équi-réparties sur l'intervalle [0; 2π] :
>> **x = 0: 2*pi/100 : 2*pi;**
On peut aussi utiliser la fonction **linspace** :
>> **x=linspace(0,2*pi,101);**
Les valeurs peuvent être réparties de manière logarithmique avec la fonction **logspace**.

3.8. Matrices multidimensionnelles

Ci-dessous, tableau tri-dimensionnel, pour accéder à l'élément situé sur la seconde ligne, troisième colonne, quatrième page, utiliser les indices (2,3,4)



autre exemple, la matrice A :




```
A(:,:,1) =

     1     0     3
     4    -1     2
     8     2     1

A(:,:,2) =

     6     8     3
     4     3     6
     5     9     2
```

9. Résolution d'un système d'équations

Soit le système de 5 équations à 5 inconnues :

1=x+2y+3z+4u+5v

2=x+2y+3z+4u+v

3=x+2y+3z+u+v

4=-x-2y+3z+4u+v

5=x+y+z+u+v

Ecrire le script résolvant ce système

10. Génération rapide de matrices

*Soit a=ones(2,3). Est-il possible de calculer a-1, a+a, 2*a, a*a, a.*a ? Analyser les résultats.*

11. Génération rapide de vecteurs

En une ligne de commande, créer le vecteur colonne ayant comme valeurs (sans les écrire explicitement) :

$[10^{-60};10^{-59}; ...; 10^{59}; 10^{60}]$

indice : ***logspace***

12. Manipulation de matrices

Créer la matrice suivante en utilisant la commande reshape :

1	*11*	*21*	*31*	*41*	*51*	*61*	*71*	*81*	*91*
2	*12*	*22*	*32*	*42*	*52*	*62*	*72*	*82*	*92*
3	*13*	*23*	*33*	*43*	*53*	*63*	*73*	*83*	*93*
4	*14*	*24*	*34*	*44*	*54*	*64*	*74*	*84*	*94*
5	*15*	*25*	*35*	*45*	*55*	*65*	*75*	*85*	*95*
6	*16*	*26*	*36*	*46*	*56*	*66*	*76*	*86*	*96*
7	*17*	*27*	*37*	*47*	*57*	*67*	*77*	*87*	*97*
8	*18*	*28*	*38*	*48*	*58*	*68*	*78*	*88*	*98*
9	*19*	*29*	*39*	*49*	*59*	*69*	*79*	*89*	*99*
10	*20*	*30*	*40*	*50*	*60*	*70*	*80*	*90*	*100*

Utiliser triu pour créer :

1	*11*	*21*	*31*	*41*	*51*	*61*	*71*	*81*	*91*
0	*12*	*22*	*32*	*42*	*52*	*62*	*72*	*82*	*92*
0	*0*	*23*	*33*	*43*	*53*	*63*	*73*	*83*	*93*
0	*0*	*0*	*34*	*44*	*54*	*64*	*74*	*84*	*94*
0	*0*	*0*	*0*	*45*	*55*	*65*	*75*	*85*	*95*
0	*0*	*0*	*0*	*0*	*56*	*66*	*76*	*86*	*96*
0	*0*	*0*	*0*	*0*	*0*	*67*	*77*	*87*	*97*
0	*0*	*0*	*0*	*0*	*0*	*0*	*78*	*88*	*98*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*89*	*99*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*100*

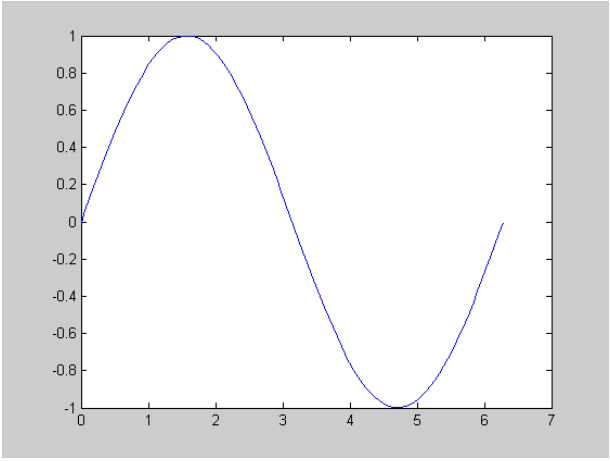
4. Graphique 2D

Une courbe 2D est représenté par une série d'abscisses et une série d'ordonnées. Le logiciel trace généralement des droites entre ces points. La fonction de MATLAB s'appelle **plot**.

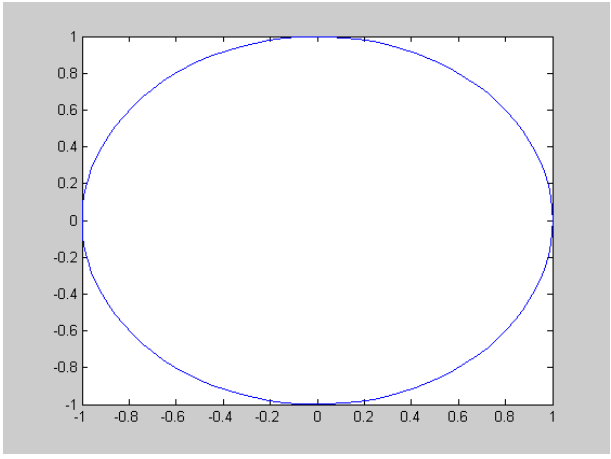
4.1. L'instruction plot

4.1.1. Tracer une courbe simple

L'utilisation la plus simple de l'instruction plot est la suivante.
plot (vecteur d'abscisses, vecteur d'ordonnées) ou encore,
plot ([x1 x2 ... xn], [y1 y2 ... yn])
Les vecteurs peuvent être indifféremment ligne ou colonne, pourvu qu'ils soient tous deux de même type. En général ils sont lignes car la génération de listes de valeurs vue précédemment fournit par défaut des vecteurs lignes.
Par exemple, si on veut tracer sin(x) sur l'intervalle [0; 2π], on commence par définir une série (de taille raisonnable, disons 100) de valeurs équidistantes sur cet intervalle :
>> **x = 0: 2*pi/100 : 2*pi;**
puis, comme la fonction **sin** peut s'appliquer terme à terme à un tableau :
>> **plot(x, sin(x))**
qui fournit le graphe suivant dans la fenêtre graphique :



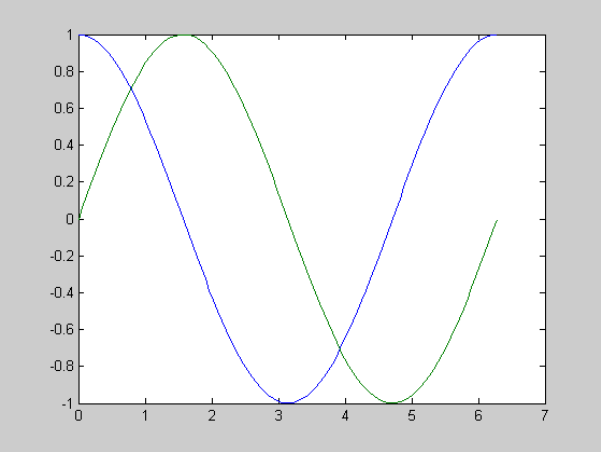
On voit que les axes s'adaptent automatiquement aux valeurs extrémales des abscisses et ordonnées.
On remarquera que tout ce que demande plot, c'est un vecteur d'abscisses et un vecteur d'ordonnées. Les abscisses peuvent donc être une fonction de x plutôt que x lui-même. En d'autres termes, il est donc possible de tracer des courbes paramétrées :
>> **plot(cos(x), sin(x))**



4.1.2. Superposer plusieurs courbes

Il suffit de spécifier autant de couples (abscisses, ordonnées) qu'il y a de courbes à tracer. Par exemple pour superposer sin et cos :

>> **plot(x, cos(x),x, sin(x))**



Les deux courbes étant en réalité dans des couleurs différentes. Cette méthode fonctionne même si les abscisses des deux courbes ne sont pas les mêmes.

4.1.3. Le piége classique

Vous tomberez rapidement dessus, et rassurez-vous, vous ne serez pas le premier. Essayez par exemple de tracer le graphe de la fonction x ==> x*sin (x) sur [0; 2π]. Fort des deux section précédentes, vous y allez franco :

>> **x = 0:0.1:2*pi;**

>> **plot(x, x*sin(x))**

??? Error using ==> *

Inner matrix dimensions must agree.

Pour MATLAB vous manipulez des tableaux : x en est un, et sin(x) en est un deuxième de même taille, et tous deux sont des vecteurs lignes. Est-ce que cela a un sens de multiplier deux vecteurs lignes au sens des matrices ? Non. C'est pourtant ce que vous avez fait, puisque vous avez utilisé le symbole * . Vous avez compris : il fallait utiliser la multiplication terme à terme .* soit :

>> **plot(x, x.*sin(x))**

4.1.4. Attributs de courbes

MATLAB attribue des couleurs par défaut aux courbes. Il est possible de modifier la couleur, le style du trait et celui des points, en spécifiant après chaque couple (abscisse, ordonnée) une chaîne de caractères (entre primes) pouvant contenir les codes suivants (obtenus en tapant help plot) :

Specifier	Line Style
`-`	solid line (default)
`--`	dashed line
`:`	dotted line
`-.`	dash-dot line

Specifier	Color
`r`	red
`g`	green
`b`	blue
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

Specifier	Marker Type
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square
d	diamond
^	upward pointing triangle
v	downward pointing triangle
>	right pointing triangle
<	left pointing triangle
p	five-pointed star (pentagram)
h	six-pointed star (hexagram)

Lorsque l'on utilise seulement un style de points, MATLAB ne trace plus de droites entre les points successifs, mais seulement les points eux-même. Ceci peut être pratique par exemple pour présenter des résultats expérimentaux.
Les codes peuvent être combinés entre eux. Par exemple :
>> **plot(x,sin(x),':',x,cos(x),'r-.')**

4.2. Décoration des graphiques

4.2.1. Titre

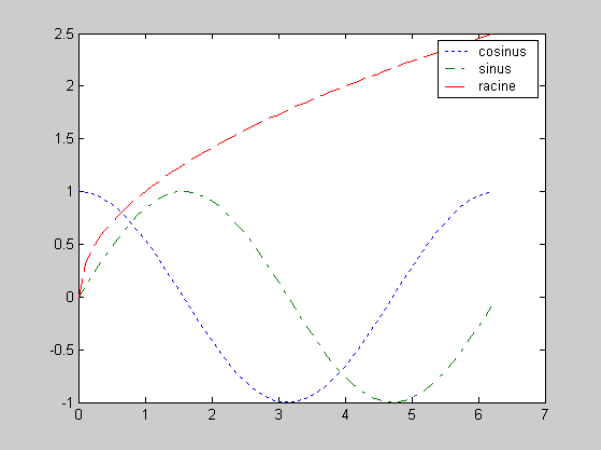
C'est l'instruction **title** à laquelle il faut fournir une chaîne de caractères. Le titre apparaît en haut de la fenêtre graphique :
>> **plot(x,cos(x),x,sin(x))**
>> **title('Fonctions sin et cos')**

4.2.2. Labels

Il s'agit d'afficher quelque chose sous les abscisses et à coté de l'axe des ordonnées :
>> **plot(x,cos(x))**
>> **xlabel('Abscisse')**
>> **ylabel('Ordonnée')**

4.2.3. Légendes

Utilisation de l'instruction **legend**. Il faut lui communiquer autant de chaînes de caractères que de courbes tracées à l'écran. Un cadre est alors tracé sur le graphique, qui affiche en face du style de chaque courbe, le texte correspondant. Par exemple :
>> **plot(x,cos(x),':',x,sin(x),'-.',x,sqrt(x),'--')**
>> **legend('cosinus','sinus','racine')**



4.2.4. Tracer un quadrillage

C'est l'instruction **grid**, qui utilisé après une instruction plot affiche un quadrillage sur la courbe. Si on tape à nouveau **grid**, le quadrillage disparaît.

4.3. Afficher plusieurs graphiques (subplot)

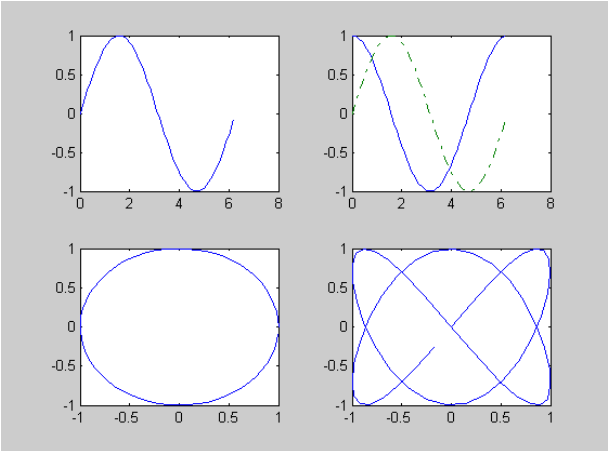
Voilà une fonctionnalité très utile pour présenter sur une même page graphique plusieurs résultats. L'idée générale est de découper la fenêtre graphique en pavés de même taille, et d'afficher un graphe dans chaque pavé. On utilise l'instruction **subplot** en lui spécifiant le nombre de pavés sur la hauteur, le nombre de pavés sur la largeur, et le numéro du pavé dans lequel on va tracer :

subplot (Nbre pavés sur hauteur, Nbre pavés sur largeur, Numéro pavé)

La virgule peut être omise. Les pavés sont numérotés dans le sens de la lecture d'un texte :
de gauche à droite et de haut en bas :

Une fois que l'on a tapé une commande **subplot**, toutes les commandes graphiques suivantes seront exécutées dans le pavé spécifié. Ainsi, le graphique suivant a été obtenu à partir de la suite d'instructions :

```
>> subplot(221)
>> plot(x,sin(x))
>> subplot(222)
>> plot(x,cos(x),x,sin(x),'-.')
>> subplot(223)
>> plot(cos(x),sin(x))
>> subplot(224)
>> plot(sin(2*x),sin(3*x))
```



4.4. Axes et zoom

Il y a deux manières de modifier les valeurs extrêmes sur les axes, autrement dit de faire du zooming sur les courbes. La plus simple est d'utiliser le bouton de la fenêtre graphique. Vous pouvez alors :

- encadrer une zone à zoomer avec le bouton de gauche de la souris, ou
- cliquer sur un point avec le bouton de gauche. Le point cliqué sera le centre du zoom, ce dernier étant effectué avec un facteur arbitraire
- cliquer sur un point avec le bouton de droite pour dézoomer

Pour faire des zooms plus précis, utiliser la commande axis (voir l'aide intégrée).

4.5. Instructions graphiques diverses

4.5.1. Maintien du graphique

Par défaut une instruction **plot** efface systématiquement le graphique précédent. Il est parfois utile de le conserver et de venir le surcharger avec une nouvelle courbe. Pour cela on utilise la commande **hold**. Pour démarrer le mode surcharge, taper **hold on**, pour revenir en mode normal, **hold off**.
Il est conseillé de ne pas abuser de cette commande.

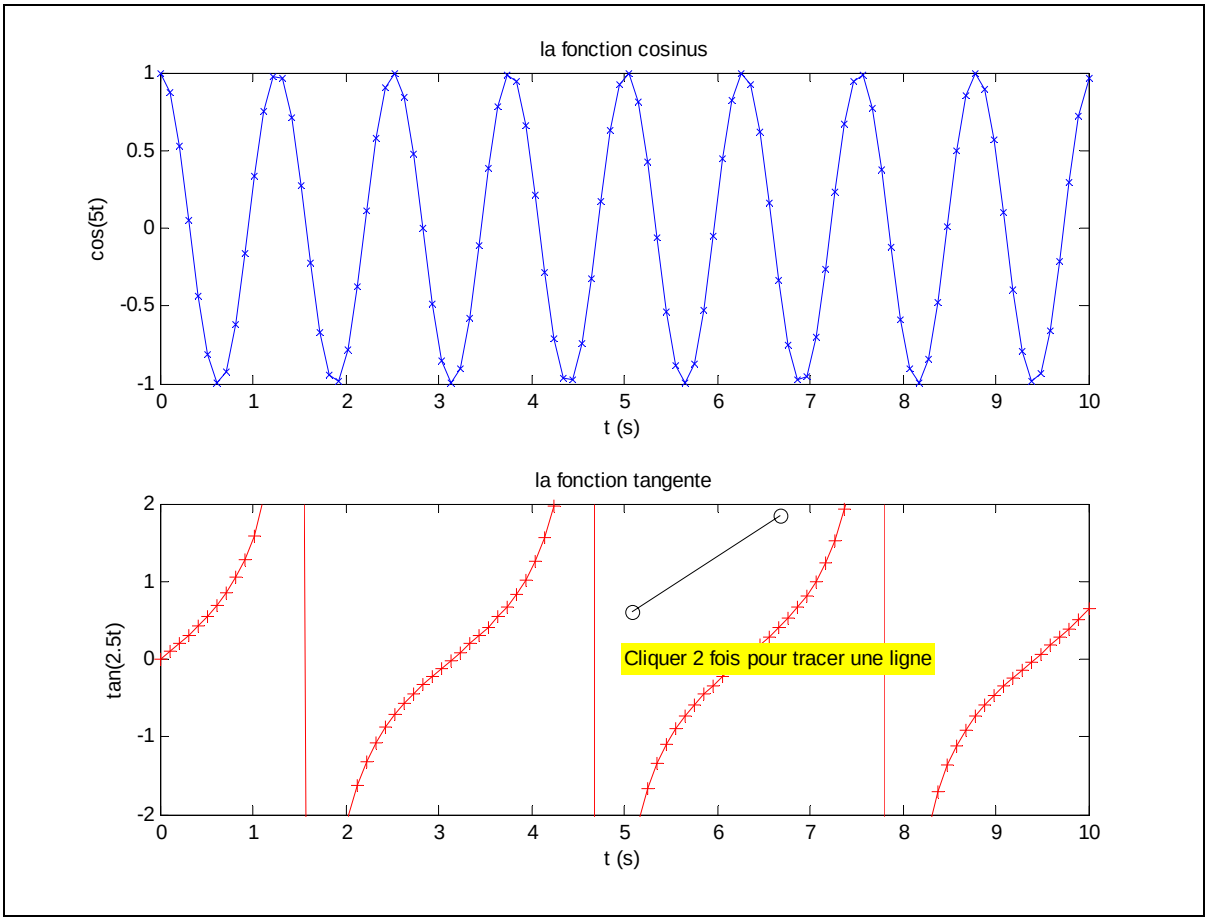
4.5.2. Effacement de la fenêtre graphique

Tapez simplement **clf**. Cette commande annule également toutes les commandes **subplot** et **hold** passées.

4.5.3. Saisie d'un point à la souris

La commande **ginput(N)** permet de cliquer N points dans la fenêtre graphique. La commande renvoie alors deux vecteurs, l'un contenant les abscisses, l'autre les ordonnées. Utilisée sans le paramètre N, la commande tourne en boucle jusqu à ce que la touche "Entrée" soit tapée.

13. Subplot et ginput
Ecrire un script qui représente les fonctions cos(5t) (ligne bleue et des 'X') et tan(t) (ligne rouge et des '+') dans l'intervalle t [0,10] sur 2 graphes d'une même figure, placer titres et labels. Utiliser la fonction ***text*** *pour inviter l'utilisateur à cliquer 2 fois dans l'un des graphes. Le script comportera la fonction* ***ginput*** *pour récupérer le résultat des 2 clics et tracera une ligne entre les 2 points ainsi définis. Obtenir quelque chose qui ressemble à la figure ci-dessous:*



5. Programmation MATLAB

5.1. Fonctions

Nous avons vu un certain nombre de fonctions prédéfinies. Il est possible de définir ses propres fonctions. La première méthode permet de définir des fonctions simples sur une ligne de commande. La seconde, beaucoup plus générale permet de définir des fonctions très évoluées en la définissant dans un fichier.

5.1.1. Fonctions inline

Admettons que je veuille définir une nouvelle fonction que j'appelle **sincos** définie mathématiquement par :
sincos(x) = sin (x) - x cos (x)
On écrira :
>> **sincos = inline('sin(x)-x*cos(x)', 'x')**
sincos =
Inline function:
*sincos(x) = sin(x)-x*cos(x)*
C'est très simple. N'oubliez pas les *primes*, qui définissent dans MATLAB des chaînes de caractères. On peut maintenant utiliser cette nouvelle fonction :
>> **sincos(pi/12)**
ans =
0.0059
Essayons maintenant d'appliquer cette fonction à un tableau de valeurs :
>> **sincos(0:pi/3:pi)**
??? Error using ==> inline/subsref
*Error in inline expression ==> sin(x)-x*cos(x)*
*??? Error using ==> **
Inner matrix dimensions must agree.
Cela ne marche pas. Vous avez compris pourquoi ? (c'est le même problème que pour l'instruction plot(x,x*sin(x)) vue précédemment) MATLAB essaye de multiplier x vecteur ligne par cos(x) aussi vecteur ligne au sens de la multiplication de matrice ! Par conséquent il faut bien utiliser une multiplication terme à terme .* dans la définition de la fonction :
>> **sincos = inline('sin(x)-x.*cos(x)')**
sincos =
Inline function:
*sincos(x) = sin(x)-x.*cos(x)*

>> **sincos(0:pi/3:pi)**
ans =
0 0.3424 1.9132 3.1416
On peut donc énoncer la règle suivante :
Lorsque l'on définit une fonction, il est préférable d'utiliser systématiquement les opérateurs terme à terme .*, ./ et .^ au lieu de * / et ^ si l'on veut que cette fonction puisse s'appliquer à des tableaux.
On peut de même définir des fonctions de plusieurs variables :
>> **sincos = inline('sin(x)-y.*cos(x)', 'x', 'y')**
sincos =
Inline function:
*sincos(x,y) = sin(x)-y.*cos(x)*
L'ordre des variables (affiché à l'écran) est celui dans lequel elles apparaissent dans la définition de la fonction. Essayer :
>> sincos(1,2)

5.1.2. Fonctions définies dans un fichier

Dans les mathématiques on écrit souvent les fonctions sous la forme :
y = f(x) ou z = g(x, y) ou ....
Nous verrons que c'est la même chose sous MATLAB. Les noms de fonctions doivent être plus explicites que f ou g. Par exemple une fonction renvoyant le carré d'un argument x pourra être nommée **carre** (on évite les accents):
y = carre(x)
Dans le fichier définissant la fonction, cette syntaxe est aussi précédée du mot clé **function** :
Forme générale d'une fonction dans un fichier **nomfonction.m** :

```
function Resultat = nomfonction(Parametres_Passes_a_la_Fonction)
% lignes de commentaires (aide descriptive apparaissant dans la fenêtre de commandes % si on tape : help nomfonction )
lignes de commandes
...
Resultat=...
```

Commençons par reprendre l'exemple précédent (sincos). L'ordre des opérations est le suivant :
1. Editer un nouveau fichier appelé sincos.m
2. Taper les lignes suivantes :

```
function s = sincos(x)
% sincos calcule sin(x)-x.*cos(x)
s = sin(x)-x.*cos(x);
```

3. Sauvegarder
Le résultat est le même que précédemment :

```
>> sincos(pi/12)
ans =
0.0059
```

On remarquera plusieurs choses :
- l'utilisation de .* pour que la fonction soit applicable à des tableaux.
- la variable s n'est là que pour spécifier la sortie de la fonction.
- l'emploi de **;** à la fin de la ligne de calcul, sans quoi MATLAB afficherait deux fois le résultat de la fonction.

Donc encore une règle :
Lorsque l'on définit une fonction dans un fichier, il est préférable de mettre un **;** à la fin de chaque commande constituant la fonction. Attention cependant à ne pas en mettre sur la première ligne.
Un autre point important : le nom du fichier doit porter l'extension .m et le nom du fichier sans suffixe doit être exactement le nom de la fonction (apparaissant après le mot-clé **function**)
En ce qui concerne l'endroit de l'arborescence où le fichier doit être placé, la règle est la même que pour les scripts.

Voyons maintenant comment définir une fonction comportant plusieurs sorties. On veut réaliser une fonction appelée **cart2pol** qui convertit des coordonnées cartésiennes (x; y) (entrées de la fonction) en coordonnées polaires (r; fi) (sorties de la fonction). Voilà le contenu du fichier **cart2pol.m** :

```
function [r, theta] = cart2pol (x, y)
% cart2pol convertit des coordonnées cartésiennes (x; y) (entrées de la fonction)
% en coordonnées polaires (r; fi) (sorties de la fonction)
```

```
% sqrt = racine carrée (fonction interne de Matlab)
% atan = fonction réciproque de tangente (fonction interne de Matlab)
r = sqrt(x.^2 + y.^2);
theta = atan (y./x);
```

On remarque que les deux variables de sortie sont mises entre crochets, et séparées par une virgule.
Pour utiliser cette fonction, on écrira par exemple :

```
>> [module,phase] = cart2pol(1,1)
module =
1.4142
phase =
0.7854
```

On affecte donc simultanément deux variables *module* et *phase* avec les deux sorties de la fonction, en mettant ces deux variables dans des crochets, et séparés par une virgule.
Il est possible de ne récupérer que la première sortie de la fonction. MATLAB utilise souvent ce principe pour définir des fonctions ayant une sortie principale et des sorties optionnelles.
Ainsi, pour notre fonction, si une seule variable de sortie est spécifiée, seule la valeur du rayon polaire est renvoyée. Si la fonction est appelée sans variable de sortie, c'est **ans** qui prend la valeur du rayon polaire :

```
>> cart2pol(1,1)
ans =
1.4142
```

14. Créer une fonction: vec2col
*Ecrire la fonction **vec2col.m** qui transforme tout vecteur (ligne ou colonne) passé en argument en vecteur colonne. Un traitement d'erreur (fonction **error**) testera que la variable passée à la fonction est bien un vecteur et non une matrice (utiliser **size** et l'instruction conditionnelle **if** décrite plus bas).*

15. Equations du 1^{ier} et du second degré
Compléter les 2 fonctions dont les 2 premières lignes sont respectivement :

```
function x=equ1(a,b)
% resout a*x+b=0
```

et

```
function x=equ2(a,b,c)
% resout a*x^2+b*x+c=0
```

*qui résolvent les équations du premier et du second degré. Utiliser la fonction **nargin** pour que l'appel à **equ2(a,b, c)** soit valide. Vérifier que les solutions sont correctes.*

16. Nargin, nargout
*Chercher avec l'aide la syntaxe et l'utilisation de ces commandes. Les appliquer à **equa1** et **equa2**.*

5.1.3. Portée des variables

A l'intérieur des fonctions comme celle que nous venons d'écrire, vous avez le droit de manipuler trois types de variables :
- Les variables d'entrée de la fonction. Vous ne pouvez pas modifier leurs valeurs.
- Les variables de sortie de la fonction. Vous devez leur affecter une valeur.
- Les variables locales. Ce sont des variables temporaires pour découper des calculs par exemple. Elles n'ont de sens qu' à l'intérieur de la fonction et ne seront pas "vues" de l'extérieur. Et C'EST TOUT!

Imaginons maintenant que vous écriviez un fichier de commande, et une fonction (dans deux fichiers différents bien sûr), le fichier de commande se servant de la fonction. Si vous voulez passer une valeur A du

fichier de commandes à la fonction, vous devez normalement passer par une entrée de la fonction. Cependant, on aimerait bien parfois que la variable A du fichier de commandes soit utilisable directement par la fonction.
Pour cela on utilise la directive **global**. Prenons un exemple : vous disposez d'une relation donnant le Cp d'un gaz en fonction de la température T :
$Cp(T) = AT^3 + BT^2 + CT + D$
et vous voulez en faire une fonction. On pourrait faire une fonction à 5 entrées, pour T, A, B, C, D, mais il est plus naturel que cette fonction ait seulement T comme entrée.
Comment passer A,B,C,D qui sont des constantes ? Réponse : on déclare ces variables en **global** tant dans le fichier de commandes que dans la fonction, et on les affecte dans le fichier de commandes.
Fichier de commandes :

```
global A B C D
A = 9.9400e-8;
B = -4.02e-4;
C = 0.616;
D = -28.3;
T = 300:100:1000; % Température en Kelvins
plot(T, cp(T)) % On trace le CP en fonction de T
```

La fonction :

```
function y = cp(T)
global A B C D
y = A*T.^3 + B*T.^2 +C*T + D;
```

17. Matrice de Vandemonde
Ecrire la fonction v=vnd(x) qui à partir du vecteur de composante (x_0,...,x_n) construit la matrice suivante dite de Vandermonde :

$$v = \begin{bmatrix} x_0^n & \cdots & x_0^2 & x_0 & 1 \\ x_1^n & \vdots & x_1^2 & x_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^n & \cdots & x_n^2 & x_n & 1 \end{bmatrix}$$

18. Variable locale
*Ecrire un script de nom **base** contenant la ligne unique :*

a=12345;

*Exécuter **base**, taper ensuite dans la fenêtre de commande :*

a ↵

*Taper dans l'éditeur la fonction de nom **local** contenant les 2 lignes :*

```
function local
a=0
```

*Exécuter la fonction **local**, taper ensuite dans la fenêtre de commande :*

a ↵

Expliquer les résultats affichés.

19. Variable globale
*Ecrire une fonction de nom **globa** contenant les lignes :*

function globa
% fonction illustrant la notion de variable globale
% voir aussi globb
global A
A=101010;
Ecrire une fonction de nom ***globb*** *contenant les lignes :*
function globb
% fonction illustrant la notion de variable globale
% voir aussi globa
global A
disp(A)
Executer ***globa****, puis* ***globb****. Taper* ***A↵*** *dans la fenêtre de commandes. Taper maintenant* ***global A*** *dans la fenêtre de commandes, puis* ***A↵****. Expliquer les résultats affichés. Expliquer la différence entre les commandes* ***clear A*** *et* ***clear global A****.*

5.2. Structures de contrôle

On va parler ici de tests et de boucles. Commençons par les opérateurs de comparaison et les opérateurs logiques

5.2.1. Opérateurs de comparaison et logiques

Notons tout d'abord le point important suivant, inspiré du langage C :
MATLAB représente la constante logique "FAUX" par 0 et la constante "VRAIE" par 1. Ceci est particulièrement utile, par exemple pour définir des fonctions par morceaux. Ensuite, les deux tableaux suivants comportent l'essentiel de ce qu'il faut savoir :
Opérateur Syntaxe MATLAB

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
= =	Equal to
~=	Not equal to

Operator	Description
&	AND
|	OR
~	NOT

On peut se demander ce qui se passe quand on applique un opérateur de comparaison entre deux tableaux, ou entre un tableau et un scalaire. L'opérateur est appliqué terme à terme. Ainsi :
>> **A=[1 4 ; 3 2]**
A =
1 4
3 2
>> **A> 2**
ans =
0 1
1 0
Les termes de A supérieur à 2 donnent 1 (vrai), les autres 0 (faux).

Une application importante des opérateurs logiques concerne le traitement automatique des grandes matrices. Créons une matrice (2X5, petite pour l'exemple):

>> **A=[10, 12, 8, 11, 9; 10, 10, 128, 11, 9];**

A =

10	12	8	11	9
10	10	128	11	9

L'élément 128 s'écarte fortement de toutes les autres. Il peut s'agir par exemple d'un point aberrant apparu lors d'une mesure expérimentale et on désire remplacer les valeurs supérieure à 20 par la valeur 10. Pour une petite matrice telle que A, cela est facilement réalisable, il suffit de taper: **A(2, 3)=10;** Mais cette méthode est irréaliste si on a à faire avec une matrice 1000X1000 ! Utilisons plutôt:

>> **A(A>20)=10;**

Et le tour est joué:

A =

10	12	8	11	9
10	10	10	11	9

On peut aussi construire des fonctions par morceaux. Imaginons que l'on veuille définir la fonction suivante : f(x) = sin(x) si x > 0 sinon f(x) = sin(2x)

Voilà comment écrire la fonction:

>> **f = inline('sin(x).*(x>0) + sin(2*x).*(~(x>0))', 'x')**

f =

Inline function:

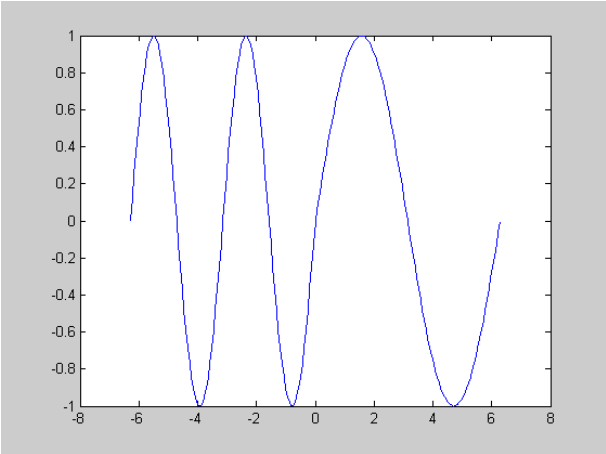
f(x) = sin(x).(x>0) + sin(2*x).*(~(x>0))*

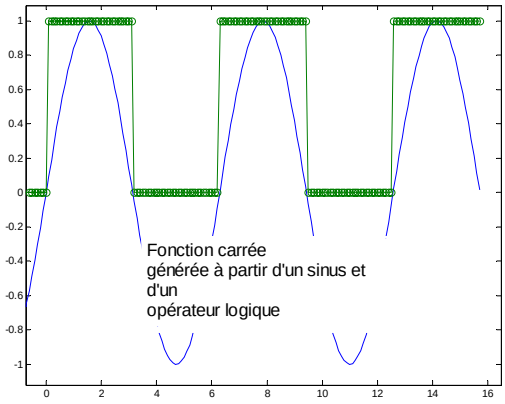
On ajoute les deux expressions sin x et sin 2x en les pondérant par la condition logique définissant leurs domaines de validité. Simple mais efficace ! Représentons la fonction ainsi définie :

>> **x=-2*pi:2*pi/100:2*pi;**

>> **plot(x,f(x))**

donne la courbe suivante à gauche:





20. <u>Utilisation d'opérateurs logiques</u>

A partir d'une fonction sinus, créer une fonction carrée telle que celle, ci-dessus à droite.

5.2.2. La commande find

La commande **find** est utile pour extraire simplement des éléments d'un tableau selon un critère logique donné.

>> **k = find (x)**

renvoie dans la variable k la liste des indices du tableau x dont les éléments sont non nuls.

Sachant que "FAUX" et 0 sont identiques dans MATLAB, cela permet de trouver tous les indices d'un tableau respectant un critère logique donné :

>> **x = [-1.2, 0, 3.1, 6.2, -3.3, -2.1]**

x =

-1.2000 0 3.1000 6.2000 -3.3000 -2.1000

>> **inds = find (x < 0)**

inds =

1 5 6

On peut ensuite facilement extraire le sous-tableau ne contenant que les éléments négatifs de x en écrivant simplement :

>> **y = x(inds)**

y =

-1.2000 -3.3000 -2.1000

Retenez bien cet exemple. Ce type de séquence s'avère très utile dans la pratique car ici encore, il économise un test (**if**). Notons enfin que la commande **find** s'applique aussi aux tableaux 2D.

21. Reshape et find

Expliquer précisément ce que fait la commande suivante :

>>a=reshape(fix((rand(1,400)-0.5)*100),20,20);

Trouver tous les éléments de valeur égale à 7.

Transformer la matrice a en un vecteur colonne b, trier b par ordre croissant. Combien d'éléments ont pour valeur 7?

*Indices : **find**, **sort**.*

ATTENTION : à moins d'ajouter un commentaire explicite, ce genre d'expression est à proscrire car il rend la compréhension du code difficile.

5.2.3. Instructions conditionnelles if

La syntaxe est la suivante :

if condition logique
instructions
elseif condition logique
instructions
...
else
instructions
end

On remarquera qu'il n'y a pas besoin de parenthèses autour de la condition logique. Notons qu'il est souvent possible d'éviter les blocs de ce type en exploitant directement les opérateurs de comparaison sur les tableaux (voir la fonction carrée et la fonction par morceaux définie dans la section précédente), ainsi que la commande **find**.

*22. Fonction avec if et is**

*Un certain nombre de fonctions détecte l'état des entités MATLAB. Créer une fonction **parite(x)** qui renvoie la parité du nombre x ('x est pair' ou 'x est impair'), précise s'il s'agit d'un nombre premier ('x est premier'), traite le cas où x n'est pas un nombre et celui où x n'est pas entier (en utilisant la fonction **estentier**). La fonction devra utiliser les fonctions **isnumeric**, **isprime** et les commandes de contrôles **if...elseif...end**.*

5.2.4. Boucles for

for variable = valeur début: pas: valeur fin
instructions
end

Exemple:

```
for i = 1:m
  for j = 1:n
    H(i,j) = 1/(i+j);
  end
end
```

23. Boucle conditionnelle, break

Combien de fois cette boucle est-elle évaluée ? Quelles sont les valeurs de i affichées à l'écran ?

```
for i=12:-2:1;
   disp(i)
   if i==8
      break
   end
end
```

*24. **Tic** et **toc***

*Ecrire un script utilisant les fonctions **tic** et **toc** et une (ou des) boucle(s) telle(s) que le temps écoulé soit de l'ordre de la seconde. Le résultat est-il toujours le même ?*

5.2.5. Boucles while

while condition logique
instructions
end

25. Boucle tant que

Que fait le script suivant ?

```
eps = 1;
while (1+eps) > 1
   eps = eps/2;
end
eps = eps*2
```

5.2.6. return

Un retour à la fonction appelante ou à la fenêtre de commande, est provoqué lorsque MATLAB rencontre la commande **return**.

5.2.7. Switch, case, otherwise

26. Switch

Que fait le script ci-dessous ?

```
clc
couleur = input('Entrer une couleur : ', 's');
couleur=lower(couleur);

switch couleur
```

```
  case 'rouge'
    chaine='La couleur choisie est : rouge';
  case 'vert'
    chaine='La couleur choisie est : vert';
  case 'bleu'
    chaine='La couleur choisie est : bleu';
  otherwise
    chaine='Couleur inconnue';
end
disp(chaine)
```

27. Boucles implicites

Ecrire explicitement le vecteur u défini par des boucles implicites :

u(1:2:10)= (1:2:10).^2;

u(2:2:10)= (2:2:10).^3;

Expliquer les résultats.

5.3. Types de données (data types)

5.3.1. Types numériques

Les **entiers** sont codés sur 8, 16 32 ou 64 bits, il peut-être avantageux de les utiliser pour économiser de la mémoire. C'est le cas pour les images dont les niveaux d'intensité des pixels sont représentés en entiers non-signés sur 8 bits.

Data Type	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	`int8`
Signed 16-bit integer	-2^{15} to $2^{15}-1$	`int16`
Signed 32-bit integer	-2^{31} to $2^{31}-1$	`int32`
Signed 64-bit integer	-2^{63} to $2^{63}-1$	`int64`
Unsigned 8-bit integer	0 to 2^8-1	`uint8`
Unsigned 16-bit integer	0 to $2^{16}-1$	`uint16`
Unsigned 32-bit integer	0 to $2^{32}-1$	`uint32`
Unsigned 64-bit integer	0 to $2^{64}-1$	`uint64`

Essayer et interpréter ceci:

```
r=2^7*rand(100,100,3);
r=uint8(r);
image(r);
```

Les **nombres à virgules flottantes** double-précision (codés sur 64 bits) sont les nombres utilisés par défaut par MATLAB. En simple-précision, ils sont codés sur 32 bits.

MATLAB manipule les **nombres complexes**.

inf = infini et **NaN** = Not a Number sont 2 valeurs spéciales utiles, représentant par exemple 1/0 et 0/0 (ou encore inf/inf) respectivement.

5.3.2. Type logique

Les variables logiques ont déjà étés abordées plus haut. Elles ne prennent que 2 valeurs 0 ou 1, "faux" ou "vrai" logique codé sur 1 octet. Essayer :

```
>> [30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

Les opérateurs logiques suivants retournent un type logique :

& (and), | (or), ~ (not), xor, any, all

Ils peuvent être utilisés lorsque l'on veut exécuter une portion de code conditionnel , extraire des portions de matrices (voir exercice correspondant).

5.3.3. Caractères et chaînes

Les chaînes sont traités comme des tableaux.

5.3.4. Structures

Les structures (bases de données) sont des tableaux avec des "conteneurs de données" appelés "champs". Les champs peuvent être de n'importe quel type. Par exemple, un champ peut contenir une chaîne de caractère représentant un nom, un autre contenir un nombre représentant l'âge, un troisième une matrice de notes, ...
Exemple :

```
etudiant.nom = 'Jean Durand';
etudiant.age = 21;
etudiant.notes = [12; 13; 10; 14; 14];
```

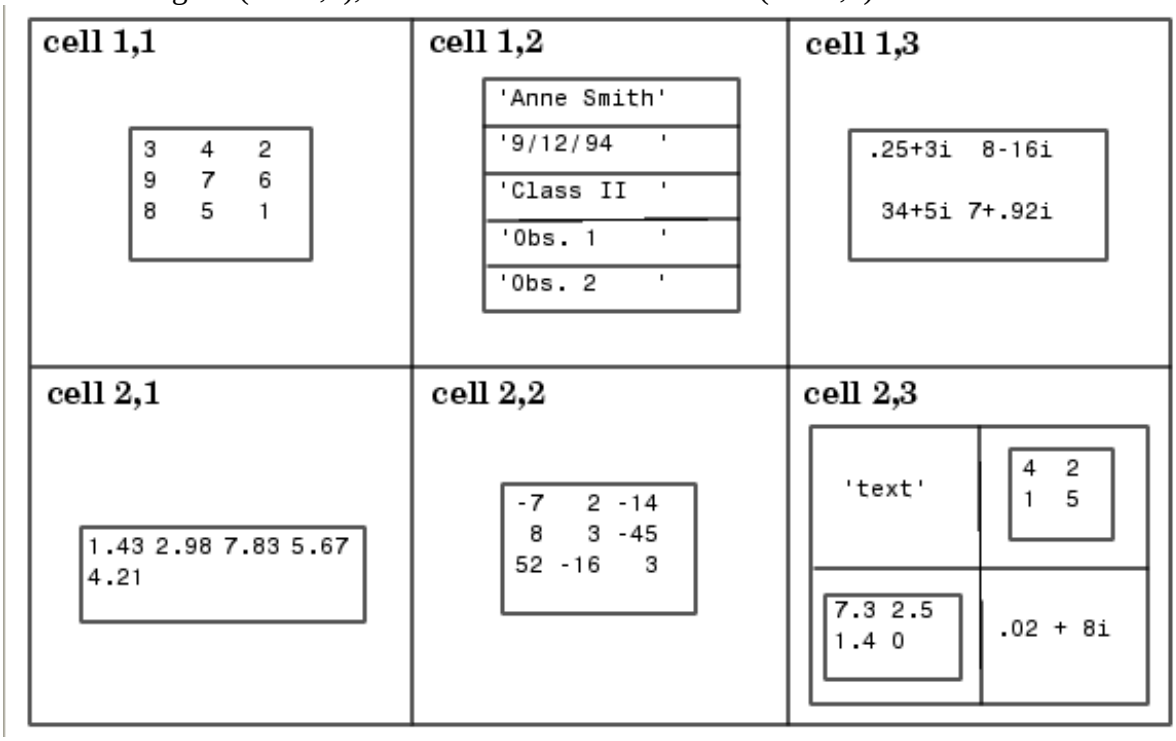
etudiant est une structure à 3 champs ne comportant pour l'instant qu'un seul élément. Ajoutons en un second:

```
etudiant(2).nom = 'Gabriel Dupont';
etudiant(2).age = 22;
etudiant(2).notes = [19; 18; 18; 17; 19];
```

Les structures sont particulièrement utiles pour passer en une seule variable un grand nombre de paramètres.

5.3.5. Cellules

Un tableau de cellules permet de stocker des données de type différents. La figure ci-dessous montre un tableau A (2 X 3) de cellules contenant: un tableau d'entiers non signés (cell 1,1), un tableau de caractères (cell 1,2), un tableau de nombres complexes (cell 1,3), un tableau de nombres à virgules flottantes (cell 2,1), un tableau d'entiers signés (cell 2,2), et un autre tableau de cellules (cell 2,3).



Pour accéder aux données d'un tableau de cellules, utiliser la même méthode d'indexation que pour les autres matrices MATLAB avec des accolades { } à la place des parenthèses ( ). Par exemple A{1, 3} donne le contenu de la cellule placée sur la première ligne, à la troisième colonne, ici le tableau de nombre complexes.
Pour **créer la cellule** (1, 3), on peut écrire:
A(1, 3)={0.25+3i, 8-16i; 34+5i, 7+0.92i}; ou
A{1, 3}=[0.25+3i, 8-16i; 34+5i, 7+0.92i];

Les tableaux de cellules peuvent être une alternative aux structures.

28. Fonctions signenombre et signematrice
*En utilisant la structure de contrôle **if...elseif...else...end**, écrire la fonction **signenombre** renvoyant la valeur -1, 0 ou 1 suivant que le nombre passé en entrée est négatif, nul ou positif respectivement.*
*Ecrire maintenant une fonction **signematrice** s'appliquant aux matrices et renvoyant une matrice de -1, 0*

29. Ecrire une fonction dont l'en-tête est :
function i=recherche(v,x)
recherchant dans un vecteur v la position du dernier élément égal à une valeur x donnée (ou 0 si il est absent).

30. Fonction estentier
*Ecrire une fonction { **function r=estentier(x)** } utilisant **fix** pour tester si un nombre x est entier (r=1) ou non (r=0). La fonction doit pouvoir s'appliquer à des matrices.*

31. Types numériques
Dans la fenêtre de commandes, taper
a=65;
b=int8(65);
puis comparer la taille mémoire occupée par ces 2 variables, c'est-à dire le nombre de bytes (octets) utilisés pour les coder indiqués dans la fenêtre "Workspace".

32. Type logique
Que fait le script suivant :
```
a=input('Entrer un nombre entier ');
if ~isprime(a)
disp([num2str(a),' n''est pas premier'])
return
end
disp([num2str(a),' est premier'])
```
Celui-ci ?
```
A = 5:5:50
B = [1 3 6 7 10]
A(B)
```
Essayer et expliquer ceci :
```
A = fix(25*rand(5))
B = A>20
A(~B) = 0
```

33. Type caractères et chaînes de caractères
*Concaténer les 2 chaînes : **a='abc'** et **z='uvwxyz'** en ligne **chligne=[a,z]**, puis en colonne **chcolonne=[a;z]**. La commande char, ajoute des espaces pour adapter la différence de longueurs des chaînes. Essayer **char(abc, z)**.*

*Transformer **chligne** en entiers correspondant aux codes ASCII de ses caractères : **asc=uint8(chligne)**.*

*Revenir aux caractères avec la commande **char(ans)***

*Rechercher la position du caractère 'w' : **position = findstr('w', chligne)***

*Que fait cette ligne de commande : **dec2bin(a(findstr('w', a)))***

34. Types structures et cellules
Création d'une structure fictive etudiant, comportant les champs : nom, prenom, age, notes et moyenne.

*1/ Ecrire une fonction **enAlea=abaleatoire(a, b)** qui renvoie un entier aléatoire compris entre a et b.*
Cette fonction servira à déterminer de façon aléatoire
- la longueur de chaînes de caractères représentant les noms et les prénoms d'une liste de personnes fictives, cette longueur devra être comprise entre 5 et 12
- l'âge qui devra être compris entre 16 et 21

*2/ Ecrire une fonction **chAlea=chainealeatoire(n)** qui renvoie une chaîne de caractères aléatoires de longueur n. Les codes ASCII des lettres a, b, c, ...,z s'étendent de 97 à 122. Les noms et prénoms des étudiants seront générés par cette fonction.*

*3/ Ecrire une fonction **noAlea=notesaleatoires** qui renvoie une matrice aléatoire (5X1) de nombre compris entre 0 et 20 et définis à une décimale à prés.*

*4/ Ecrire la fonction principale (principale dans le sens où elle utilise les fonctions précédentes) **etStruct=generationetudiant(n)** qui génère la structure etudiant comportant n éléments (donc n étudiants).*

*5/ Ecrire la fonction **afficheunetudiant(s)** qui indique à l'utilisateur le nombre d'éléments de la structure s, l'invite à entrer un numéro d'élément et affiche à l'écran le nom et le contenu des champs de cet élément.*

6. Conseils dans la programmation

Conventions largement adoptés par les programmeurs, même dans d'autres langages. Elles ne sont évidemment pas obligatoires mais permettent une meilleure lisibilité et assurent l'entretien des programmes.

6.1. Conventions d'écriture

* Les noms de variables devraient documenter leur sens ou usage, commencer par une minuscule avec éventuellement, des majuscules isolées pour assurer leur lisibilité (pas d'accentuation):
masse2
poidsVehicule

* La longueur d'un nom est lié à la portée de la variable. Des variables entières comme i, j, k, n, ... ou réelles comme x, y, u, ... ont une portée de quelques lignes de code.

* Le préfixe n devrait être réservé pour indiquer des nombres :
nPoints
nFichiersOuverts

* Les itérateurs peuvent être préfixés par i :

```
for iFichier=1:nFichier
:
end
```

* Les noms de structures devraient commencer par une majuscule:
Segment.longueur
Segment.epaisseur

* Les noms de fonctions devraient être intégralement en minuscules avec éventuellement des "underscores":
calcule_moyenne(.)
triangule(.)
extraitcolonne(.)

* Les constantes devraient être en majuscules:

```
BOLTZMANN=1.38e-23; % constante de Boltzmann
LUMIERE=3e8;% vitesse de la lumière
```

6.2. Fichiers, organisation, amélioration des performances

* Le meilleur moyen d'écrire un gros programme est de l'assembler à partir de petites pièces, généralement des fonctions spécialisées dans une tâche spécifique.

* Une fonction interagit avec les programmes à partir de ses entrées, de ses sorties et des variables globales. Les structures permettent d'éviter de longues listes de variables d'entrée/sortie.

* Vectoriser les boucles, c'est à dire les convertir en opérations sur des vecteurs ou des matrices. L'exemple suivant calcule le sinus de 1001 valeurs comprises entre 0 et 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

La version vectorisée du même programme est:

```
t = 0:0.01:10;
y = sin(t);
```

Comparer le temps d'exécution avec **tic** et **toc**.

* Pré-allouer de l'espace mémoire pour les matrices. Dans les boucles for ou while qui augmente la taille d'une structure à chaque boucle, MATLAB perd du temps à chaque fois à rechercher des blocs contigus de mémoire de plus en plus grands. La rapidité du programme sera amélioré si la matrice est initialisée dés le début à sa taille maximale. Voici une version dispendieuse en temps d'exécution:

```
x = 0;
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

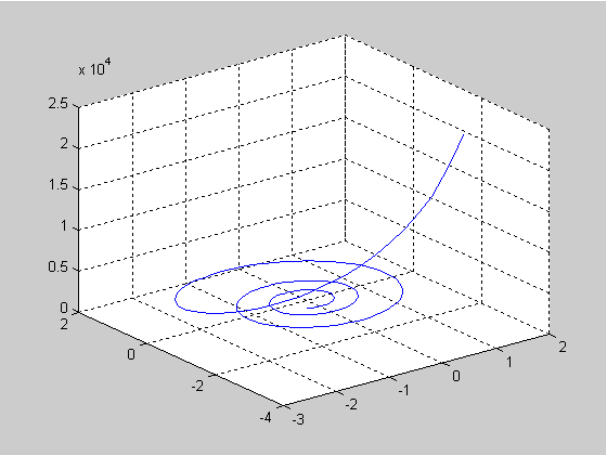
Changer la première ligne en allouant un bloc de mémoire de 1 X 1000 pour x initialisé à zéro produit un sérieux gain de temps machine:

```
x = zeros(1, 1000);
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

7. Graphiques 3D

7.1. Courbes en 3D

Une courbe en 2D est définie par une liste de doublets (x; y) et une courbe en 3D par une liste de triplets (x; y; z). Puisque l'instruction plot attendait deux arguments, un pour les x, un pour les y, l'instruction **plot3** en attend trois : les x, les y et les z.
Voici un exemple de courbe paramétrée :
>> **t=-10:0.1:10;**
>> **plot3(exp(-t/10).*sin(t), exp(-t/10).*cos(t), exp(-t))**
>> **grid**



7.2. Surfaces

7.2.1. Génération des points (meshgrid)

Pour définir une surface, il faut un ensemble de triplets (x; y; z). En général les points (x; y) tracent dans le plan un maillage régulier mais ce n'est pas une obligation. La seule contrainte est que le nombre de points soit le produit de deux entiers m×n (on comprendra pourquoi très bientôt).
Si on a en tout m×n points, cela signifie que l'on a m×n valeurs de x, m×n valeurs de y et m×n valeurs de z. Il apparaît donc que abscisses, ordonnées et cotes des points de la surface peuvent être stockées dans des tableaux de taille m×n.
Toutes les instructions de tracé du surface, par exemple **surf** auront donc la syntaxe suivante:
surf (tableau d'abscisses, tableau d'ordonnées, tableau de cotes)

$$\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \vdots & x_{2n} \\ \vdots & \vdots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \quad \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \vdots & y_{2n} \\ \vdots & \vdots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix} \quad \begin{bmatrix} z_{11} & \cdots & z_{1n} \\ z_{21} & \vdots & z_{2n} \\ \vdots & \vdots & \vdots \\ z_{m1} & \cdots & z_{mn} \end{bmatrix}$$

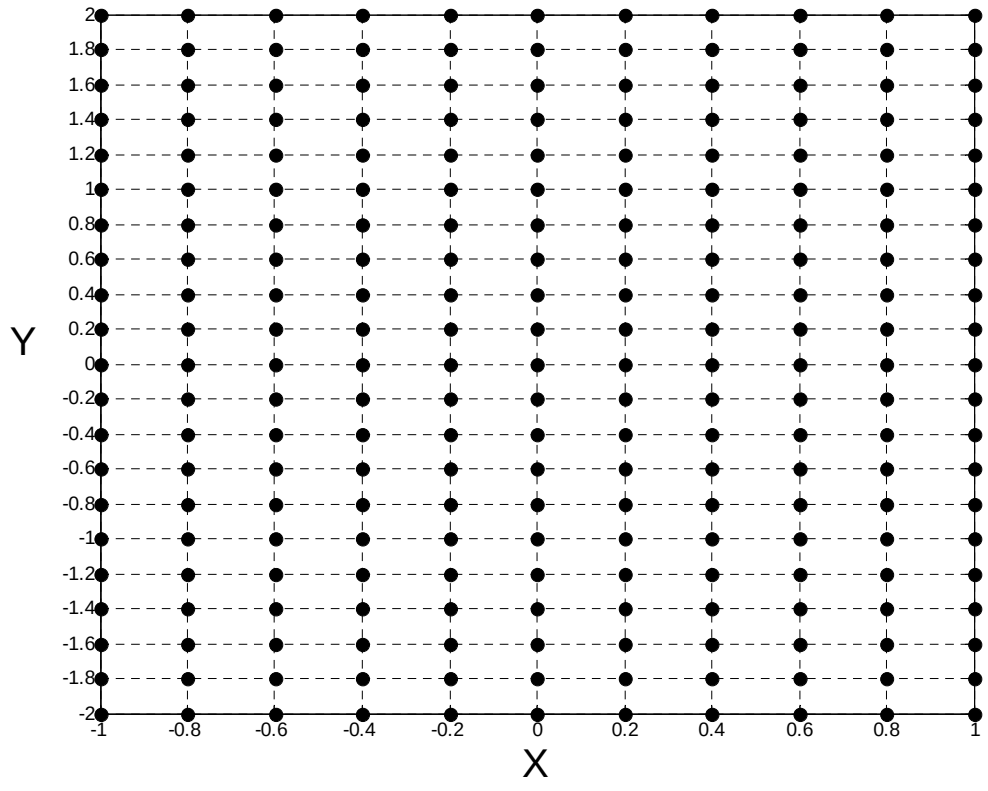
Il reste maintenant à construire ces tableaux. Prenons l'exemple de la surface définie par $z = x^2 + y^2$ dont on veut tracer la surface représentative sur [-1; 1] × [-2; 2].
Pour définir un quadrillage de ce rectangle, il faut d'abord définir une suite de valeurs x_1; ... ; x_m pour x et une suite de valeurs y_1; ... ; y_n pour y, par exemple :
>> **x = -1:0.2:1**

x =
Columns 1 through 7
-1.0000 -0.8000 -0.6000 -0.4000 -0.2000 0 0.2000
Columns 8 through 11
0.4000 0.6000 0.8000 1.0000
>> **y = -2:0.2:2**
y =
Columns 1 through 7
-2.0000 -1.8000 -1.6000 -1.4000 -1.2000 -1.0000 -0.8000
Columns 8 through 14
-0.6000 -0.4000 -0.2000 0.2000 0.4000 0.6000
Columns 15 through 21
0.8000 1.0000 1.2000 1.4000 1.6000 1.8000 2.0000

En combinant toutes ces valeurs de x et y, on obtient m×n points dans le plan (x; y). Il faut maintenant construire deux tableaux, l'un contenant les 11×21 abscisses de ces points l'autre les 11×21 ordonnées, soit:

$$X = \begin{bmatrix} -1 & -0.8 & \cdots & 1 \\ -1 & -0.8 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ -1 & -0.8 & \cdots & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} -2 & -2 & \cdots & -2 \\ -1.8 & -1.8 & \cdots & -1.8 \\ \vdots & \vdots & \vdots & \vdots \\ 2 & 2 & \cdots & 2 \end{bmatrix}$$

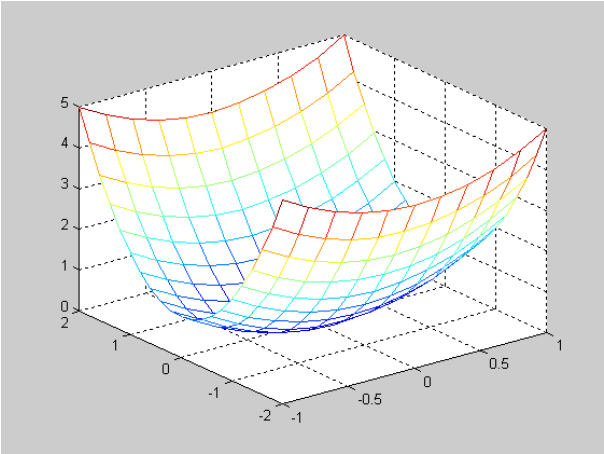
Ainsi, si on considère par exemple la première colonne de ces matrices, on obtient tous les couples (x, y) pour x=-1: (-1, -2), (-1, -1.8), (-1, -1.6), (-1, -1.4), ..., (-1, 1.8) et (-1, 2). Les autres colonnes couvrent le reste de la portion du plan xy. Ci-dessous, le maillage correspondant:



Rassurez-vous pas besoin de boucles, la fonction **meshgrid** se charge de générer ces 2 matrices:
>> **[X,Y] = meshgrid(x, y);**
Il reste maintenant à calculer les z = f(x, y) correspondants. C'est là que les calculs terme à terme sur les matrices montrent encore leur efficacité : on applique directement la formule aux tableaux X et Y, sans oublier de mettre un point devant les opérateurs *, / et ^
>> **Z = X .^2 + Y.^2;**

7.2.2. Tracé de la surface

Ensuite on peut utiliser toutes les fonctions de tracé de surface, par exemple **mesh** :
>> **mesh(X,Y,Z)**

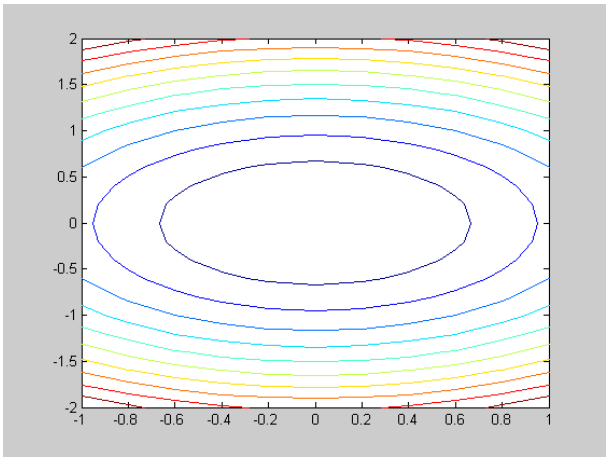


Les instructions les plus courantes sont
- **mesh** qui trace une série de lignes entre les points de la surface
- **meshc** qui fonctionne comme **mesh** mais ajoute les courbes de niveau dans le plan (x; y)
- **surf** qui "peint" la surface avec une couleur fonction de la cote
- **surfl** qui "peint" la surface comme si elle était éclairée.
- **surfc** qui fonctionne comme **mesh** mais ajoute les courbes de niveau dans le plan (x, y)

Notons enfin qu'il existe des fonctions de conversion entre les coordonnées cartésiennes, cylindriques et sphériques, permettant de tracer facilement des courbes définies dans l'un de ces systèmes de coordonnées. On regardera par exemple la documentation de **cart2pol**.

7.2.3. Courbes de contour

On utilise la fonction **contour**. Elle s'utilise comme les instructions précédentes, mais fournit un graphe 2D dans le plan (x; y). Plusieurs paramètres optionnels peuvent être spécifiés, notamment le nombre de courbes de contours à afficher :
>> **contour(X,Y,Z, 10)**



7.2.4. Contrôle de l'angle de vue

Il existe la commande **view**, mais le plus simple est de travailler à partir de la fenêtre graphique, cliquer sur  .
Cliquez ensuite avec le bouton gauche de la souris pour faire tourner la figure. ATTENTION : tout clic de la souris provoque un réaffichage, et si votre surface contient beaucoup de points, elle met beaucoup de temps se réafficher.

35. Tracé d'une surface en 3D
Tracer la surface $z=\sin(x^2 + y^2)$ pour $0<x, y<4\sqrt{\pi}$.

8. Echanges entre MATLAB et l'extérieur

8.1. Sauvegarde de données

Commandes **save** et **load**. La première permet d'écrire toute ou une partie des variables dans un fichier binaire dont l'extension est ".mat". La seconde permet de recharger les variables contenues dans le fichier. Syntaxe :

save nom fichier var1 var2 var3

Les variables sont simplement séparées par des blancs. Si aucune variable n'est spécifiée, toutes les variables sont sauvegardées dans nom_fichier.mat. La syntaxe de **load** est identique. Les données sont dans un format binaire.

Noter que des options de ces commandes permettent de sauver et de charger des fichiers de données dans le format ASCII (format texte ==> voir l'aide).

8.2. Importer des tableaux

MATLAB est souvent utilisé comme logiciel d'exploitation de résultats. Exemple classique: on a un programme FORTRAN ou C ou ... qui calcule des séries de valeurs, et on souhaite tracer une série en fonction de l'autre.

Le moyen le plus simple est que votre programme écrive un fichier texte organisé sous forme de n colonnes, séparées par des blancs (espaces), contenant chacune m lignes:

x_1 y_1 z_1 : : :

x_2 y_2 z_2 : : :

x_3 y_3 z_3 : : :

: : :

x_m y_m z_m : : :

Cette structure est obligatoire pour que la méthode simple exposée ici fonctionne correctement. Ce fichier doit porter une extension, qui peut être quelconque, mais différente de ".m" ou ".mat". Admettons qu'il soit nommé toto.res, tapez dans MATLAB :

>> **load toto.res**

Ce faisant, vous créez un nouveau tableau MATLAB, qui porte le nom du fichier sans extension, soit toto. Vous pouvez maintenant extraire les colonnes de ce tableau (voir section 3.3), et les mettre dans des vecteurs, par exemple :

>> **x = toto(:,1)**

>> **y = toto(:,2)**

>> **z = toto(:,3)**

et vous n'avez plus qu' à tracer l'un de ces vecteurs en fonction d'un autre.

8.3. Inclure des courbes MATLAB dans un document

Dans la fenêtre graphique, choisissez File -> Export. Vous pouvez alors créer toutes sortes de fichier graphiques avec le contenu de la fenêtre graphique.

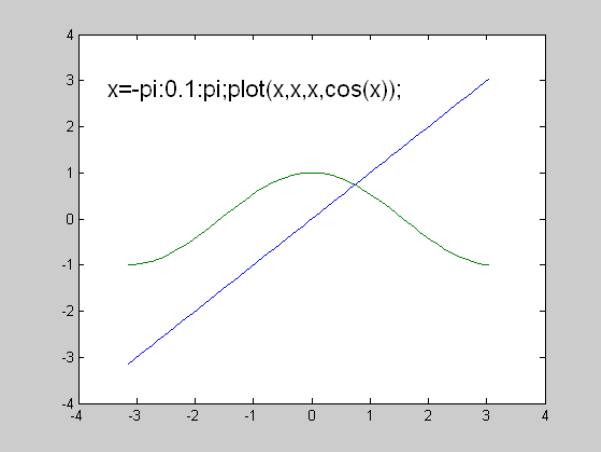
Pour les documents du pack office de Windows, vous pouvez aussi de façon assez simple, cliquer dans la fenêtre graphique sur edit, copy figure, puis coller dans votre document comme cela a été réalisé dans celui-ci.

9. Calcul numérique avec MATLAB

L'objectif de ce chapitre est de présenter brièvement des routines de calcul numérique fournies dans la version de base de MATLAB. La liste des problèmes fournie dans ce polycopié est non exhaustive et on trouvera des informations complémentaires dans les différentes documentations en ligne MATLAB. De même, les concepts théoriques sont supposés connus par le lecteur et il ne sera pas fait mention des méthodes utilisées.

9.1. Recherche des zéros d'une fonction

Problème : On cherche x_0 tel que $f(x_0) = 0$.
La fonction **fzero** permet de résoudre ce problème. Il faut fournir d'une part la fonction f elle-même, et d'autre part une estimation (valeur initiale) de x_0. L'efficacité de l'algorithme est comme toujours dépendante de la valeur initiale choisie. La fonction f peut-être définie par une directive **inline** ou bien écrite dans une fichier.
Par exemple on cherche le zéro de :
$f(x) = \cos(x) - x$
Une approche graphique permet souvent de trouver une estimation de x_0. La figure suivante montre ainsi que les fonctions $x \rightarrow x$ et $x \rightarrow \cos(x)$ se coupent en un point sur $[-\pi; \pi]$. Une valeur raisonnable pour l'estimation de x_0 est par exemple 0.



On peut donc écrire, en utilisant inline :
>> **f = inline('x-cos(x)')**
>> **fzero(f,0)**
ans =
0.7391
On remarquera que la variable f envoyée à la fonction **fzero** est elle-même une fonction. Toutes les routines de calcul numérique de MATLAB nécessitant l'écriture d'une fonction par l'utilisateur fonctionnent selon ce principe.
On peut également écrire la fonction dans un fichier f.m :
function y = f(x)
y = x-cos(x);
et ensuite on écrira :
>> **fzero('f',0)**
ans =
0.7391
ATTENTION! On remarquera ici que le symbole f est mis entre primes. Cela vient du fait qu'ici, la définition de la fonction f est faite dans un fichier. Il est préférable en général de définir les fonctions dans des fichiers car cela offre une plus grande souplesse.
Un dernier point important : comment faire lorsque la définition de la fonction dépend en plus d'un paramètre ? Par exemple, on veut chercher le zéro de la fonction $f(x) = \cos(mx) - x$
où m est un paramètre susceptible de varier entre deux exécutions. On ne peut pas rajouter un argument à la définition de notre fonction f car **fzero** impose que f ne dépende que d'une seule variable. Une solution serait d'affecter m dans le corps de la fonction mais elle est mauvaise car :
- lorsque l'on veut changer la valeur de m il faut modifier la fonction,
- cette fonction sera peut-être appelée des dizaines voire des centaines de fois par **fzero**, et on répétera à chaque fois la même instruction d'affectation de m.

La bonne solution est d'utiliser la directive **global** vue pécédemment. La fonction f s'écrira donc :

```
function y = f(x)
global m
y = x-cos(m*x);
```

et on cherchera son zéro en écrivant :

```
>> global m
>> m = 1;
>> fzero(f,0)
```

Notons enfin que l'on aura tout intérêt à mettre les trois lignes ci-dessus dans un fichier de commandes, et à lancer ce fichier de commandes en bloc.

9.2. Interpolation

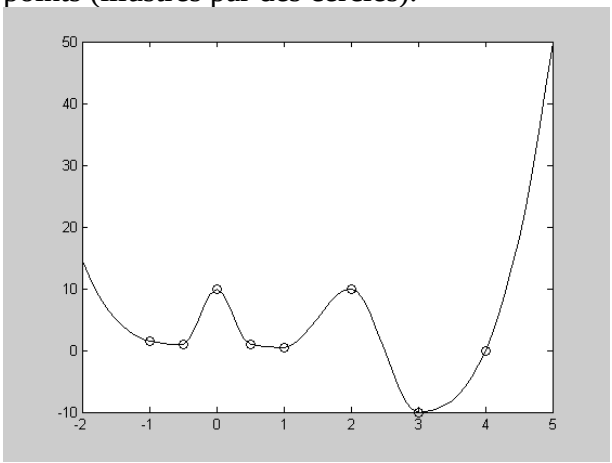
Le problème : connaissant des points tabulés (x_i; y_i), construire une fonction polynomiale passant par ces points. En termes plus simples, c'est ce que fait la fonction **plot** quand elle "relie" les points que vous lui donnez : elle fait passer un polynôme de degré 1 entre deux points consécutifs.

La fonction **interp1** résout le problème. Il suffit de lui donner les valeurs tabulées ainsi qu'une série d'abscisses où l'on souhaite connaître les ordonnées interpolées. Sur chaque intervalle on peut interpoler par un polynôme de degré 0 (option 'nearest'), de degré 1, qui est le comportement par défaut (option 'linear') ou de degré 3 (option 'cubic').

Les lignes suivantes fournissent un exemple générique: à partir de 8 points (xi, yi), on cherche une interpolation f(x) entre ces 8 points.

```
clear; close all;
xi = [-1 -1/2 0 1/2 1 2 3 4];
yi = [1.5 1 10 1 1/2 10 -10 0];
x = -2:0.1:5;
ycubic = interp1 (xi, yi, x, 'cubic');
plot(xi,yi,'ok', x,ycubic,'-k')
```

La figure suivante illustre l'interpolation (en traits pleins) cubiques obtenues par **interp1** sur ces 8 points (illustrés par des cercles).



9.3. Approximation (estimation de paramètres ou "fitting" ou "ajustement")

Problème : faire passer une courbe d'équation connue au milieu d'un nuage de points de telle sorte que la courbe soit "la plus proche possible" de l'ensemble des points. Comme on cherche à minimiser une distance entre deux fonctions, on parle en général d'approximation aux moindres carrés ou encore de régression.

9.3.1. Linéaire

Illustrons la méthode par un exemple: on cherche à ajuster des points expérimentaux (xi; yi) par une fonction de la forme :

$$y = f(x) = a_1 + \frac{a_2}{x} + \frac{a_3}{x^2}$$

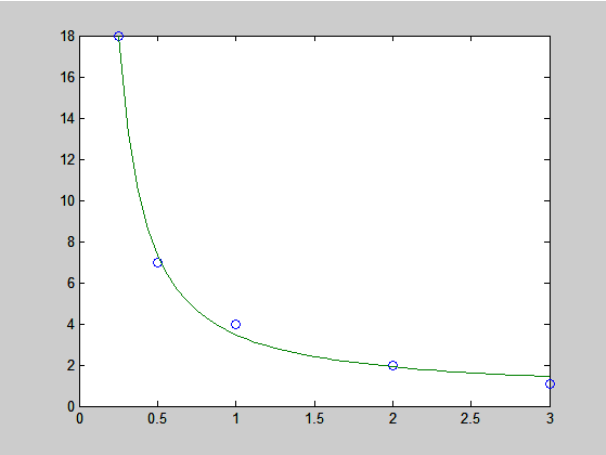
Ou encore de façon plus générale, la forme suivante : $y = a_1 f_1(x) + a_2 f_2(x) + ... + a_m f_m(x)$.
Les 5 points expérimentaux sont donnés par : xi = [0.25 1/2 1 2 3] et yi = [18 7 4 2 1.1].
On a donc un système de 5 équations à 3 inconnues (le système est dit "surcontraint", c'est à dire qu'il y a plus d'équations que d'inconnues) :

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{x} & \frac{1}{x^2} \\ 1 & \frac{1}{x} & \frac{1}{x^2} \\ 1 & \frac{1}{x} & \frac{1}{x^2} \\ 1 & \frac{1}{x} & \frac{1}{x^2} \\ 1 & \frac{1}{x} & \frac{1}{x^2} \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

de la forme $Y = F \times A$. On trouve la matrice des paramètres par inversion : $F^{-1} \times Y = A$ qui s'écrit sous MATLAB par F \ Y = A.
La séquence d'instructions est la suivante :

```
>> xi = [1/4 1/2 1 2 3]; xi = xi'; % on définit les abscisses et ordonnées
>> yi = [18 7 4 2 1.1]; yi = yi'; % en colonnes
>> F = [ones(size(xi)) 1./xi 1./(xi.^2)];
>> A = F \ yi
>> x = 1/4:1/16:3; % On trace la courbe obtenue
>> plot(xi,yi,'o', x, A(1) + A(2)./x + A(3)./(x.^2))
```

Ce programme trace la courbe ajustée ainsi que les points expérimentaux :



Cette méthode marche dés que la fonction cherchée dépend linéairement des coefficients inconnus (le cas opposé dépasse le cadre d'une simple introduction). Cela fonctionne en particulier pour faire de l'approximation polynomiale, mais dans ce cas on peut utiliser directement la fonction **polyfit**, qui en plus des coefficients du polynôme renvoie des informations sur la qualité de l'approximation réalisée. La fonction **polyfit** attend en entrée simplement les xi, les yi et l'ordre du polynôme recherché. La fonction **polyval** peut - être utilisée pour recalculer le polynôme d'approximation en tout point.
L'exemple suivant calcule deux polynômes d'approximation, un d'ordre 3, l'autre d'ordre 4 et trace le résultat.

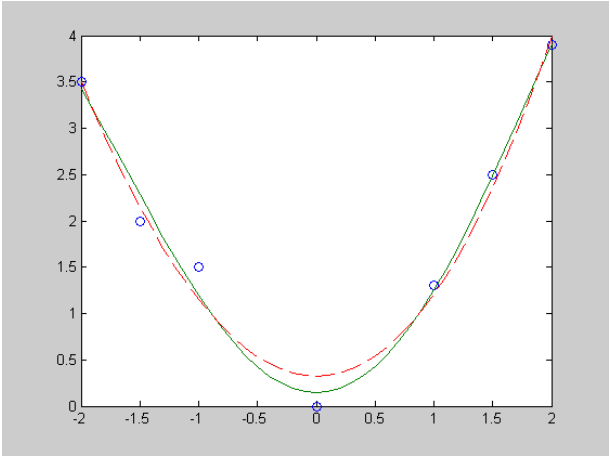
```
>> xi = [-2 -1.5 -1 0 1 1.5 2];
>> yi = [3.5 2 1.5 0 1.3 2.5 3.9];
>> A4 = polyfit (xi, yi, 4)
>> A3 = polyfit (xi, yi, 3)
>> x = -2:0.1:2;
>> plot(xi, yi, 'o', x, polyval(A4,x), x, polyval(A3,x), '--' )
```

A4 =

-0.0652 0.0313 1.1411 -0.0022 0.1473

A3 =

0.0313 0.8570 -0.0022 0.3247



36. <u>Simulation d'une exponentielle décroissante bruitée</u>

On désire simuler la mesure d'une grandeur expérimentale pseudo-périodique.

*a/ Ecrire une fonction { **function s=singau(param, t)** } qui renvoie une sinusoïdale de la forme générale :*

$$s(t) = a * \exp(-b * (t - td)^2) * \cos(2 * \pi * f * t + phi)$$

param sera un vecteur avec la correspondance :

param(1)= a
param(2)= b
param(3)= td
param(4) = f
param(5) = phi

t est un vecteur temps de N points (t = 0, δt, 2δt, 3δt, ..., (N-1)δt) créé préalablement, a l'amplitude, b un coefficient d'amortissement, td un décalage temporel, f la fréquence et phi la phase initiale. La fonction sera correctement documentée.

*b/ Ecrire une fonction { **function yb=bruit(y, sig)** } qui additionne un bruit normal d'écart type sig à un vecteur quelconque y. Ce bruit simulera celui toujours présent lors des mesures expérimentales.*

*c/ Appliquer la fonction **bruit** à s (Une valeur sig = 0.1*max(s) paraît raisonnable et signifie qu'on commet une erreur accidentelle de 10% du calibre (=max(s)) sur les mesures), comparer dans le même graphe les 2 courbes s et bruit(s, 0.1*max(s)). Ci-dessous une partie de la figure qu'on peut obtenir.*