Initiation à MATLAB

Olivier LOUISNARD

12 octobre 2001

Sommaire

Préambule

Ce document est issu d'une première version écrite en 1993, qui avait l'avantage de tenir sur 10 pages. Au fil des années il est apparu que beaucoup de concepts importants étaient passés sous silence, ce qui obligeait les enseignants à combler ces lacunes au fur et à mesure.

Le présent document reprend donc ces concepts plus en détail, notamment les fonctions et le graphique 3D. Il n'a pas vocation à se substituer à la documentation MATLAB, très bien faite, illustrée de nombreux exemples, et surtout consultable sur le WEB. Son objectif principal est de démystifier l'idée couramment répandue que MATLAB est un logiciel difficile à utiliser, et devrait permettre à n'importe quel utilisateur sachant se servir d'une calculette de prendre en main le logiciel en moins d'une heure.

Il est à mon sens beaucoup plus simple et souple d'utilisation que n'importe quel tableur (aucun nom ne sera cité ...), même pour des applications simples de visualisation de données. La philosophie est toute différente d'un tableur, en ce sens que toutes les données ne sont pas visibles directement à l'écran.

Toutes les fonctions ne seront bien sûr pas détaillées dans ce document (il en existe plusieurs centaines) : seules les fonctionnalités à mon sens utiles pour un travail scientifique quotidien seront abordées, avec un fort accent sur les fonctions graphiques 2D ou 3D.

La version 2 de ce polycopié contenait un nouveau chapitre présentant brièvement quelques problèmes de calcul numériques abordables simplement avec des fonctions de base MATLAB, notamment les problèmes de régression linéaire et de résolution d'équations différentielles ordinaires.

Cette version 3 propose de plus dans ce dernier chapitre des problèmes d'identification de paramètres non-linéaires, problèmes récurrents dans l'analyse de données expérimentales, notamment spectrométriques.

Enfin, les nouvelles fonctionnalités de la version 6 n'ont pas été prises en compte dans cette version. Cela est peu important, les améliorations effectuées depuis la version 5 portant essentiellement sur l'interface utilisateur. Ces améliorations seront prises en compte dans une version ultérieure.

Introduction

MATLAB (MATrix LABoratory) comprend de nombreuses fonctions graphiques, un système puissant d'opérateurs s'appliquant à des matrices, des algorithmes numériques (EDOs, zeros d'une fonction, intégration, interpolation), ainsi qu'un langage de programmation extrêmement simple à utiliser.

Il fut conçu initialement (au milieu des années 1980) pour manipuler aisément des matrices à l'aide de fonctions pré-programmées (addition, multiplication, inversion, décompositions, déterminants ...), en s'affranchissant des contraintes des langages de proprammation classique :
– Plus de déclarations de variables.
– Plus de phase d'édition-compilation-exécution.
Cette orientation calcul matriciel a depuis évolué vers un outil pouvant être vu comme une super-calculatrice graphique et regroupant dans la version de base la quasi-majorité des problèmes numériques (hormis les EDP qui sont aussi diversifiées que délicates à résoudre).

Plusieurs extensions plus «pointues» ont été conçues sous la forme de «TOOLBOXes», qui sont des paquets (payants) de fonctions supplémentaires dédiées à un domaine particulier :

– CONTROL pour l'automatique
– SIGNAL pour le traitement du signal
– OPTIMIZATION pour l'optimisation
– NEURAL NETWORK pour les réseaux de neurones

Cet 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.

Notons enfin que MATLAB est disponible sur tous types de plates-formes (toutes les stations sous UNIX y compris LINUX, Windows 9x et Macintosh).

Chapitre 1

Généralités et prise en main

1.1 Démarrage, quitter

Pour lancer le programme, tapez **matlab** dans une fenêtre de commandes. Une fenêtre logo fait une brêve 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

L'aide en ligne peut être obtenue directement dans la session en tapant **help** *nom de commande*. La commande **help** toute seule vous indiquera les différents thèmes abordés dans la documentation.

Il y a mieux : toute la documentation papier peut être consultée sur le WEB avec un navigateur quelconque. C'est très pratique car il existe des hyper-liens de la documentation d'une fonction à une autre, ainsi que des exemples prêts à fonctionner. Adresse :

`http ://intranet.enstimac.fr/docinfo/doc_matlab_5.3/helpdesk.html`

Vous pouvez y arriver également à partir de la page d'accueil de l'école par `Intranet -> La documentation informatique de l'EMAC` (sauvegardez ensuite l'adresse dans vos signets). La rubrique la plus utile est «Matlab functions» qui décrit l'intégralité des fonctions disponibles. Elles sont accessibles soit par thèmes (utile quand on cherche une fonction dont on ne connaît pas le nom) ou par index.

1.3 Calculs élémentaires

Commençons par les opérateurs les plus courants : `+, -, *, /, ^`. Le dernier signifie «puissance», et on retiendra qu'il est différent de celui du FORTRAN. 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 vaiable 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 sont faits avec aussi peu de précision.* La précision utilisée par MATLAB pour stocker les réels est celle du double precision FORTRAN. Si vous voulez afficher les nombres avec plus de précision, tapez la commande **format long**. Pour revenir au comportament initial : **format short**.

1.4 Historique des commandes

C'est une fonctionnalité très utile, inspirée des shells UNIX modernes : toutes les commandes que vous aurez tapé sous MATLAB peuvent être retrouvées et éditées grâce aux touches de direction. Appuyez sur $\Uparrow$ pour remonter dans les commandes précédentes, $\Downarrow$ pour redescendre et utilisez $\Rightarrow$, $\Leftarrow$, et la touche «Backspace» 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 $\Uparrow$.

Chapitre 2

Variables et fonctions prédéfinies

2.1 Variables

Le calcul effectué plus haut n'a guère d'intérêt en soi. Il ets 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 affcter 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`. Comme indiqué dans le préambule ces variables ne sont pas affichéers 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
```

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 oipérateurs 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 et liste des variables

La commande **clear** permet d'effacer une partie ou toutes les variables définies jusqu'à présent. Il est conseillé de placer cette commande au début de vos fichiers de commandes, en particulier lorsque vous manipulez des tableaux. Syntaxe :

clear *var1 var2 var3 ...*

Si aucune variable n'est spécifiée, toutes les variables seront effacées.

La commande **who** affiche les noms de toutes les variables en cours.

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. 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!!

2.4 Fonctions prédéfinies

Toutes les fonctions courantes et 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**. Il faut deux paramètres : l'ordre de la fonction et l'argument lui-même. Ainsi $J_1(3)$ s'écrira `besselj(1,3)`

La notion de fonction est plus générale dans MATLAB, et certaines fonctions peuvent

avoir plusieurs entrées (comme **besselj** par exemple) mais aussi plusieurs sorties.

Chapitre 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[1] :

```
>> M = [1 2 3
11 12 13
21 22 23]

M =

     1     2     3
    11    12    13
    21    22    23
```

[1] Le choix de prendre des variables majuscules pour les matrices est personnel et n'est nullement imposé par MATLAB

Le passage d'une ligne de commande à la suivante s'effectuant par la frappe de la touche «Entrée».

On peut également utiliser le symbole **,** qui sert de séparateur de colonne et **;** de séparateur de ligne. Ainsi pour définir la matrice précédente on aurait pu taper :

>> M = [1,2,3;11,12,13;21,22,23]

ou bien, en remplaçant la virgule par des blancs :

>> M = [1 2 3;11 12 13;21 22 23]

On peut aussi définir des vecteurs, ligne ou colonne, à partir de cette syntaxe. Par exemple :

>> U = [1 2 3]

```
U =

     1     2     3
```

définit un vecteur ligne, alors que :

>> V = [11
12
13]

```
V =

    11
    12
    13
```

définit un vecteur colonne. On aurait pu aussi définir ce dernier par :

>> V=[11;12;13]

3.2 Accès à un élément d'un tableau

Il suffit d'entre 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_{32} :

```
>> 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 de demander se 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
```

Si on utilise le caractère `:` seul, ça veut dire 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 peut définir la matrice

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

qui est une matrice 4x4. Pour faire ça 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 le caractère ;

```
>> 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
    11    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
```

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.

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 tableau de même taille. Ces symboles sont fondamentaux lorsque l'on veut tracer des courbes.

3.5.3 Multiplication, division et puissance au sens matriciel

Puisque l'on peut manipuler des matrices, il paraît intéressant de disposer d'un multiplication matricielle. Celle-ci se note simplement `*` et ne doit pas être confondur 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) à la matrice inverse de B. *ATTENTION : même si la matrice B est régulière, elle peut être mal conditionnée, et la méthode numérique utilisée pour calculer son inverse peut renvoyer des résultats faux* (cf. cours systèmes linéaires).

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 ausii 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^{me} 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 à 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
```

3.5.4 Transposition

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

3.5.5 Synthèse

Le tableau suivant résume les différents opérateurs applicables aux matrices.

Syntaxe MATLAB	Ecriture mathématique	Terme général
`A`	A	A_{ij}
`B`	B	B_{ij}
`A+B`	$A+B$	$A_{ij}+B_{ij}$
`A-B`	$A-B$	$A_{ij}-B_{ij}$
`A.*B`		$A_{ij}B_{ij}$
`A./B`		A_{ij}/B_{ij}
`A.^B`		$A_{ij}^{B_{ij}}$
`A.^s`		A_{ij}^{s}
`A*B`	AB	$\sum_k A_{ik}B_{kj}$
`A/B`	AB^{-1}	
`A\B`	$A^{-1}B$	
`A^n`	A^n	
`A'`	A^T	A_{ji}

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. ATTENTION, pour engendrer des vecteurs lignes (resp. colonnes), il faut spécifier explicitement «1» pour le nombre de lignes (resp. 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 diagonale par diagonale (voir la doc).

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 (cf. section 5.1.4).

Autre exemple pour créer 101 valeurs équiréparties sur l'intervalle $[0, 2\pi]$:

```
>> x = 0: 2*pi/100 : 2*pi;
```

On peut aussi créer des valeurs réparties de manière logarithmique avec la fonction **logspace** (voir la doc).

Chapitre 4

Graphique 2D

Une courbe 2D est pour tout logiciel de tracé de courbes représenté par une série d'abscisses et une série d'ordonnées. Ensuite, le logiciel trace généralement des droites entre ces points. MATLAB n'échappe pas à la règle. La fonction 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*)
[x_1 x_2 ... x_n] [y_1 y_2 ... y_n]

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 à la fin du chapitre précédent fournit par défaut des vecteurs ligne.

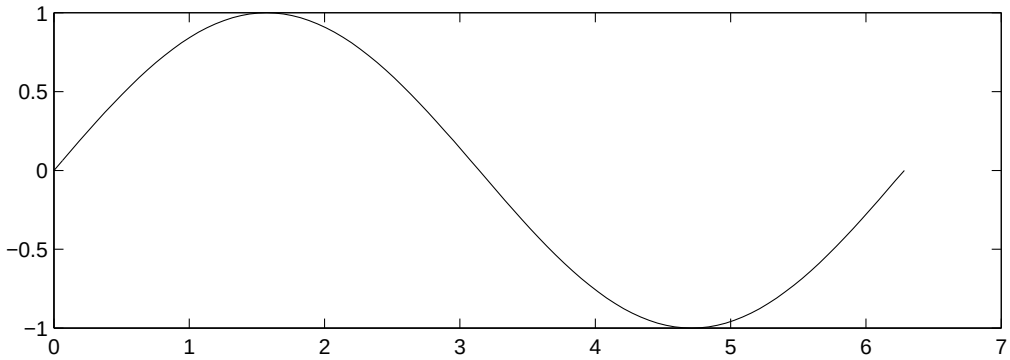
Par exemple, si on veut tracer $sin(x)$ sur l'intervalle $[0, 2\pi]$, on commence par définir une série (raisonnable, disons 100) de valeurs équidistantes sur cet intervalle :

>> 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[1] :

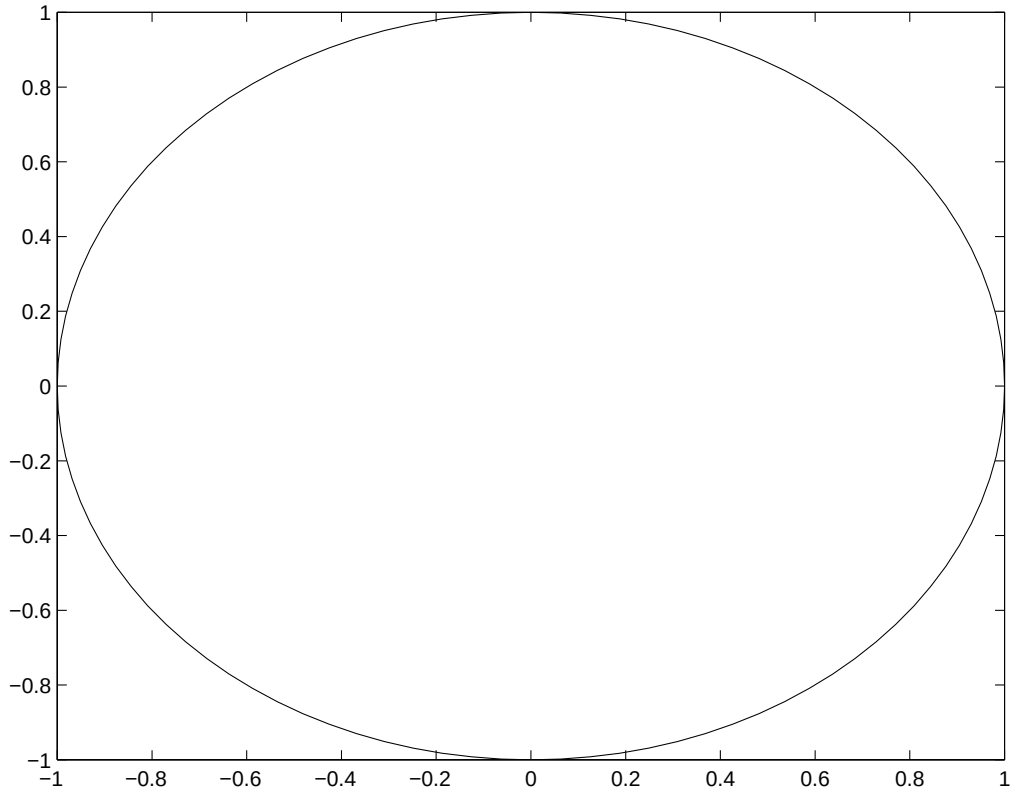


On voit que les axes s'adaptent automatiquement aux valeurs extrémales des abscisses et ordonnées.

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))

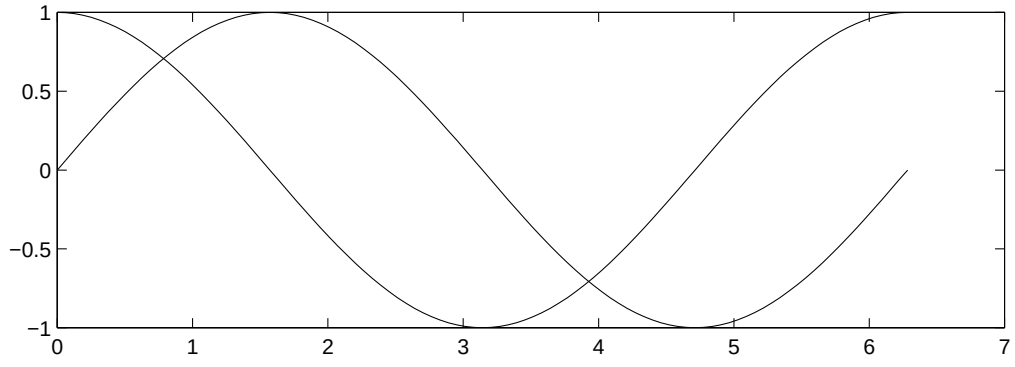
[1] On remarquera en réalité que le graphique a un autre rapport d'aspect que celui présenté. Nous avons volontairement réduit la hauteur de la courbe pour limiter le nombre de pages de ce document.



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.

Dans le cas plus fréquent où les abscisses sont les mêmes, il existe un autre moyen de superposer les courbes. On fournit toujours le vecteur des abscisses, commun aux deux courbes, et on fournit autant de vecteurs d'ordonnées qu'il y a de courbes. Tous ces vecteurs d'ordonnées sont regroupés dans un même tableau, chaque ligne du tableau représentant un vecteur d'ordonnées :

plot (*vecteur d'abscisses*, *tableau d'ordonnées*)

$$[\ x_1 \quad x_2 \quad \dots \quad x_n\ ] \qquad \begin{bmatrix} y_1^1 & y_2^1 & \dots & y_n^1 \\ y_1^2 & y_2^2 & \dots & y_n^2 \\ \vdots & \vdots & \dots & \vdots \\ y_1^m & y_2^m & \dots & y_n^m \end{bmatrix}$$

Première courbe
Deuxième courbe
$\mathrm{m}^{\text{ème}}$ courbe

Par exemple, pour superposer sin et cos, on devra fournir à **plot** les arguments suivants :

plot ($[\ x_1 \quad x_2 \quad \dots \quad x_n\ ]$, $\begin{bmatrix} \cos(x_1) & \cos(x_2) & \dots & \cos(x_n) \\ \sin(x_1) & \sin(x_2) & \dots & \sin(x_n) \end{bmatrix}$)

Le deuxième tableau se construit très facilement avec le point-virgule (voir section 3.1)

```
>> plot(x, [cos(x);sin(x)])
```

4.1.3 Le piège classique

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

```
>> x = 0:0.1:1;
>> plot(x, x*sin(x))
??? Error using ==> *
Inner matrix dimensions must agree.
```

Et voilà : vous vous faites insulter, et en plus on vous parle de matrices, alors que vous ne vouliez que tracer une bête courbe... Mais 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 ça a un sens de multiplier deux vecteurs lignes au sens des matrices ? Non, n'est-ce-pas ? 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))
```

et ça marche ! On peut donc énoncer la règle suivante :

> Lorsque l'on écrit une expression arithmétique dans les arguments de `plot`, il faut utiliser systématiquement les opérateurs terme à terme `.*` `./` et `.^` au lieu de `*` `/` et `^`

4.1.4 Attributs de courbes

Vous aurez remarqué que 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 chaine de caractères (entre quotes) pouvant contenir les codes suivants (obtenus en tapant `help plot` :

Couleurs		**Styles de points**		**Styles de lignes**	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	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-.')

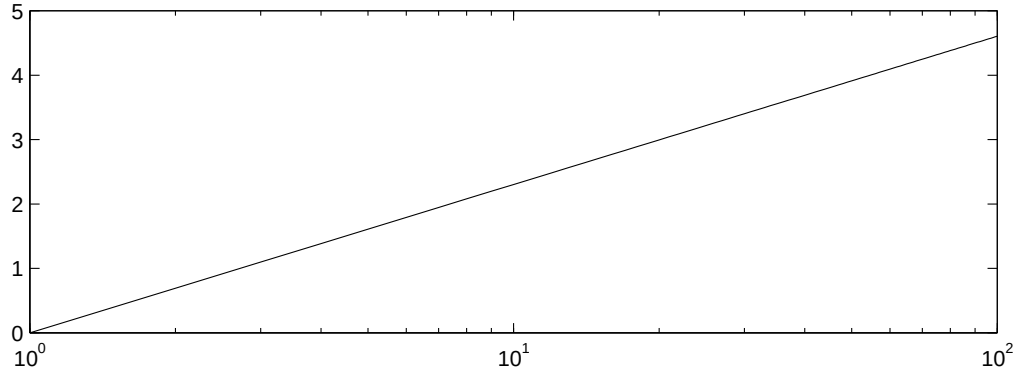
4.2 Echelles logarithmiques

On peut tracer des échelles log en abscisse, en ordonnée ou bien les deux. Les fonctions correspondantes s'appellent respectivement **semilogx**, **semilogy** et **loglog**. Elles s'utilisent exactement de la même manière que **plot**.

Par exemple :

```
>> x=1:100;
>> semilogx(x,log(x))
```

donne la courbe suivante :



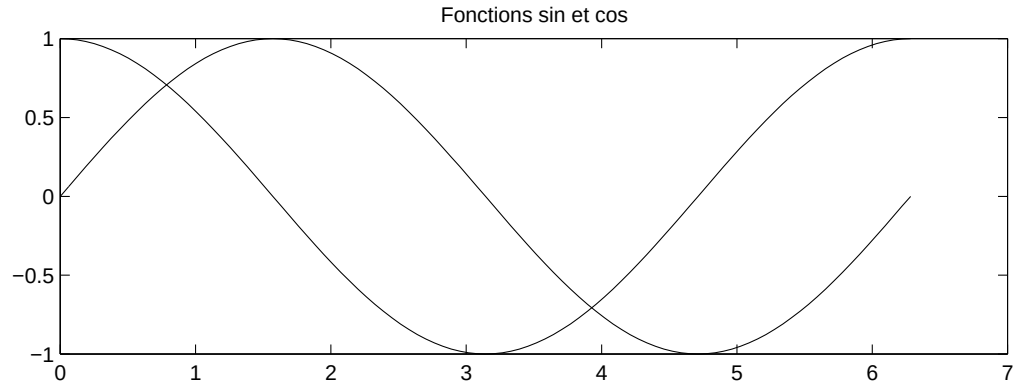
4.3 Décoration des graphiques

4.3.1 Titre

C'est l'instruction **title** à laquelle il faut fournir une chaine de caractères[2]. Le titre apparaît en haut de la fenêtre graphique :

```
>> plot(x,cos(x),x,sin(x))
>> title('Fonctions sin et cos')
```

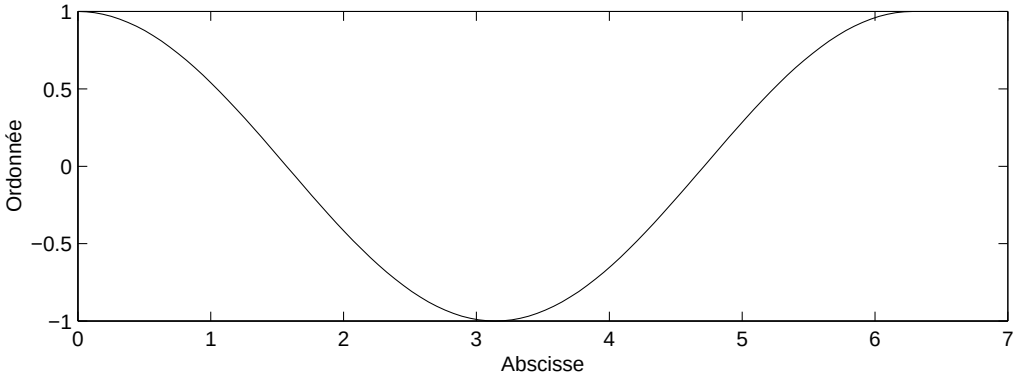
[2]Pour ceux qui connaissent LaTeX, MATLAB reconnaît une partie de la syntaxe LaTeXen ce qui concerne les formules mathématiques, notamment les indices et exposants ainsi que les lettres grecques.



4.3.2 Labels

Il s'agit d'afficher quelque chose sous les abscisses et à coté de l'axe des ordonnées :

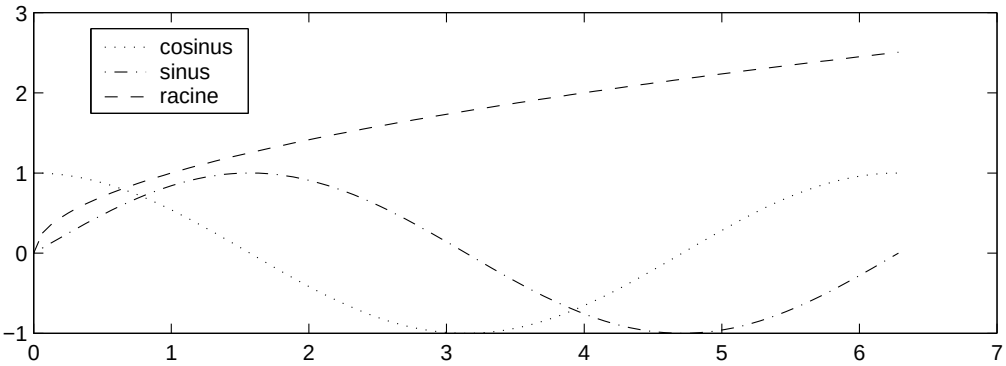
>> plot(x,cos(x))
>> xlabel('Abscisse')
>> ylabel('Ordonnée')



4.3.3 Légendes

C'est l'instruction **legend**. Il faut lui communiquer autant de chaines de caractères que de courbes tracées à l'écran. Un cadre est alors tracé au milieu du 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')

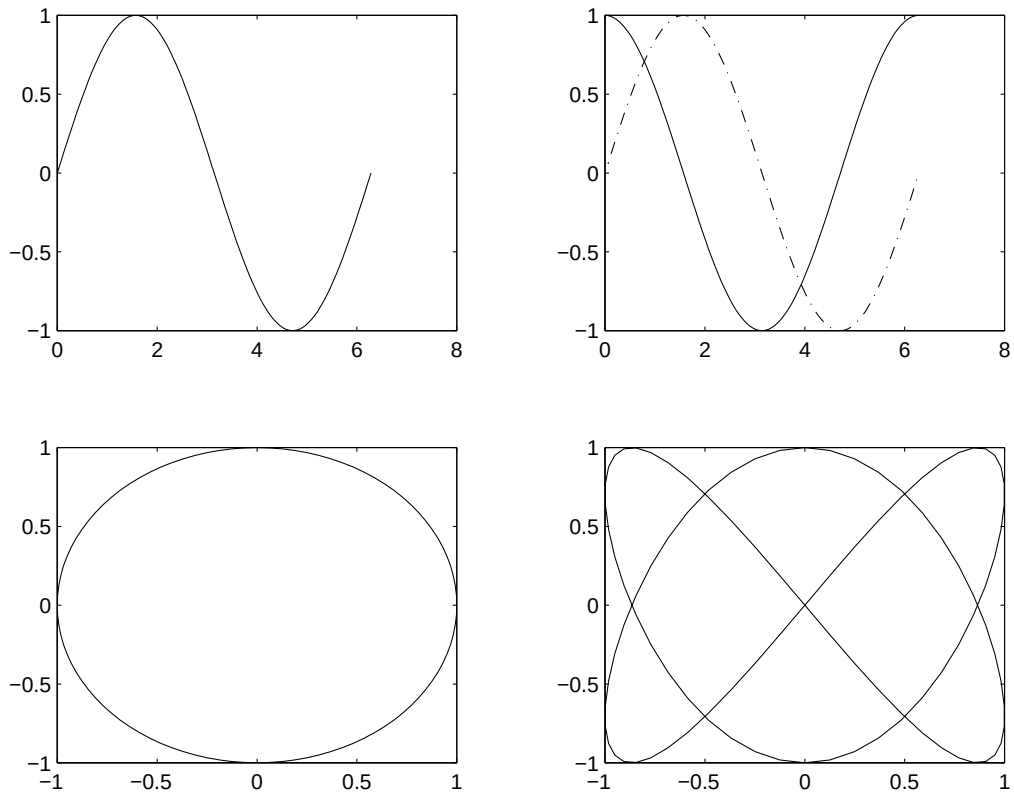


4.3.4 Tracer un quadrillage

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.

4.4 Afficher plusieurs graphiques (subplot)

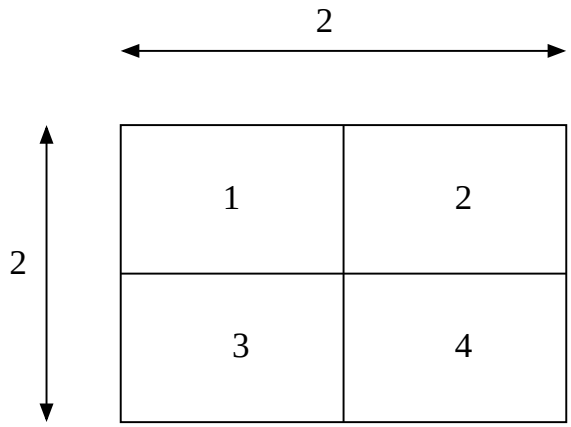
Voilà une fonctionnalité très utile pour présenter sur une même page graphique un grand nombre de résultats, par exemple :



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 précédent 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))
```

4.5 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  de la fenêtre graphique. Vous pouvez alors :

- encadrer une zone à zoomer avec le bouton de gauche de la souris
- 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 la doc).

4.6 Instructions graphiques diverses

4.6.1 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 la 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.

4.6.2 Effacement de la fenêtre graphique

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

4.6.3 Saisie d'un point à la souris

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

Chapitre 5

Programmation MATLAB

5.1 Fichiers de commandes

5.1.1 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 définir un fichier de commandes, prenez votre éditeur préféré[1], et ouvrez un fichier `toto.m`. Tapez des commandes MATLAB à l'intérieur, puis sous MATLAB, tapez :

```
>> toto
```

Toutes les commandes du fichier seront exécutées en bloc.

5.1.2 Ou doit se trouver mon fichier de commande ?

Le plus simple, c'est qu'il se trouve dans le directory courant (c'est-à-dire celui ou vous avez lancé MATLAB). Il peut se trouver aussi dans un directory référencé dans la variable

[1]Sans vouloir faire de pub (pour un logiciel domaine public :-), notez que Xemacs comporte un mode MATLAB très puissant, qui formatte automatiquement vos fichiers de commande, met des couleurs, etc. . ..

path MATLAB. Tapez cette commande pour en voir le contenu. Il est en général conseillé de se créer un directory `/home/mon_nom/matlab`.

Le path peut être modifié avec les commandes **path** et **addpath**. En général ces commandes sont placées dans le fichier `/home/mon_nom/matlab/startup.m`, qui est exécuté automatiquement au démarrage de MATLAB. Voici par exemple à quoi ressemble le mien :

```
addpath('/usr/local/public/matlab');
addpath('/home/louisnar/these/matlab');
```

Ainsi tous les fichiers de commandes présents dans ces deux directories seront accessibles de n'importe où.

5.1.3 Commentaires et autodocumentation

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 d'autodocumenter 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 ça serait dommage de s'en priver.

5.1.4 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.

5.1.5 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».

5.1.6 Mode verbeux

Si vous souhaitez qu'au fur et à mesure de son exécution, MATLAB vous affiche les commandes qu'il est en train d'exécuter, vous pouvez taper :

```
>> echo on
```

Pour revenir au mode normal, taper simplement **echo off**. Ce mode peut-etre utilisé en combinaison avec **pause** pour que le programme vous affiche quelque chose du style «Appuyez sur une touche pour continuer» [2]. Il suffit d'écrire le message dans un commentaire :

```
echo on
pause % Allez appuyez un petit coup sur Entrée pour continuer !
echo off
```

5.2 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.

5.2.1 Fonctions inline

Admettons que je veuille définir une nouvelle fonction que j'appelle `sincos` définie mathématiquement par :

$$\mathrm{sincos}(x) = \sin x - x \cos x$$

On écrira :

```
>> sincos = inline('sin(x)-x*cos(x)')

sincos =

     Inline function:
     sincos(x) = sin(x)-x*cos(x)
```

Avouez qu'on peut difficilement faire plus simple ! N'oubliez cependant pas les quotes, qui définissent dans MATLAB des chaines de caractères. On peut maintenant utiliser cette nouvelle fonction :

[2] Merci à Jeanjosé Orteu pour cette suggestion...

```
>> 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.
```

Ca ne marche pas. Vous avez compris pourquoi ? (c'est le même problème que pour l'instruction `plot(x,x*sin(x))` vue au chapitre 4) 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)
```

et maintenant ça marche :

```
>> sincos(0:pi/3:pi)

ans =

         0    0.3424    1.9132    3.1416
```

On peut donc énoncer la règle approximative 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 aura reparqué que la commande **inline** reconnaît automatiquement que la variable est `x`. On peut de même définir des fonctions de plusieurs variables :

```
>> sincos = inline('sin(x)-y.*cos(x)')

sincos =

     Inline function:
     sincos(x,y) = sin(x)-y.*cos(x)
```

Ici encore, MATLAB a reconnu que les deux variables étaient `x` et `y`. L'ordre des variables (affiché à l'écran) est celui dans lequel elles apparaissent dans la définition de la fonction. On peut cependant spécifier explicitement l'ordre des variables (voir la doc).

5.2.2 Fonctions définies dans un fichier

C'est la méthode la plus généraliste, et elle permet notamment de réaliser des fonctions ayant plusieurs sorties. 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)
s = sin(x)-x.*cos(x);
```
3. Sauvegardez

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 méritant un cadre :

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 fichiers de commande (section 5.1.2).

Il est permis de comparer la concision de MATLAB pour la déclaration d'une fonction à la lourdeur du FORTRAN ou de tout autre langage, où les paramètres formels et toutes les variables locales doivent être déclarées.

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, θ) (sorties de la fonction). Voilà le contenu du fichier `cart2pol.m`[3] :

```
function [r, theta] = cart2pol (x, y)
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 :

```
>> [rr,tt] = cart2pol(1,1)

rr =

    1.4142


tt =

    0.7854
```

On affecte donc simultanément deux variables `rr` et `tt` avec les deux sorties de la fonction, en mettant ces deux variables dans des crochets, et séparés par une virgule. Les crochets ici n'ont bien sûr pas le même sens que pour les tableaux.

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

[3] Cette fonction est ici extrêmement mal écrite (trouvez pourquoi), mais de toutes façons elle existe déjà toute faite sous MATLAB...

«optionnelles»[4].

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
```

5.2.3 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 (disons l'équivalent du programme principal en FORTRAN), et une fonction (dans deux fichiers différents bien sûr), le fichier de commande se servant de la fonction. Si vous voulez passer une valeur 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 corrélation donnant le C_p d'un gaz en fonction de la température :

$$C_p(T) = AT^3 + BT^2 + CT + D$$

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 :

[4] C'est le cas par exemple de **eig** qui renvoie les valeurs propres d'une matrice et éventuellement la matrice de passage associée.

```
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;
```

Pouir ceux qui ne l'auraient pas encore compris, ce système est à rapprocher du `common` FORTRAN.

5.3 Structures de contrôle

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

5.3.1 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
égal à	==
différent de	~=
supérieur à	>
supérieur ou égal à	>=
inférieur à	<
inférieur ou égal à	<=

Opérateur	Syntaxe MATLAB
Négation	~
Ou	|
Et	&

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). Il est sans intérêt d'effectuer cela dans une instruction **if** (voir section suivante), en revanche c'est intéressant pour construire des fonctions par morceaux. Imaginons que l'on veuille définir la fonction suivante :

$$f(x) = \begin{cases} \sin x & \text{si} \quad x > 0 \\ \sin 2x & \text{sinon} \end{cases}$$

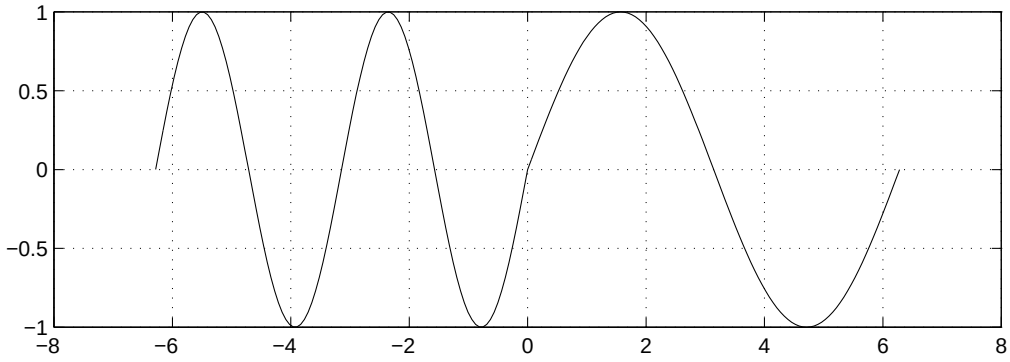
Voilà comment écrire la fonction

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

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 ! Pour vous prouver que ça marche :

>> x=-2*pi:2*pi/100:2*pi;
>> plot(x,f(x))

donne la courbe suivante :



5.3.2 La commande `find`

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

`k =` **find** `(` *tableau logique 1D*`)`

renvoie dans la variable `k` la liste des indices du tableau 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. On s'en convaincra en faisant la même chose en FORTRAN, ou bien à l'issue de la section suivante. Notons enfin que la commande **find** s'applique aussi aux tableaux 2D (voir la doc).

5.3.3 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 par morceaux définie dans la section précédente), ainsi que la commande **find**.

Il existe aussi une structure du type **switch**... **case** (voir la doc)

5.3.4 Boucles for

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

L'originalité réside dans le fait que la variable de boucle peut être réelle.

5.3.5 Boucles while

while *condition logique*
    `instructions`
end

Le fonctionnement est classique.

Chapitre 6

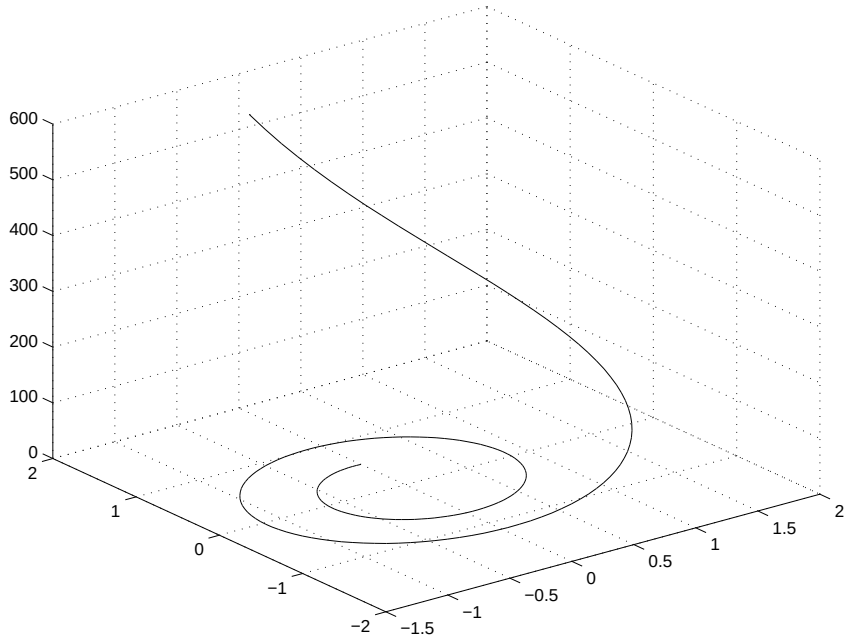
Graphique 3D et assimilés

6.1 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 :

```
>> plot3(exp(-t/10).*sin(t), exp(-t/10).*cos(t), exp(-t))
>> grid
```



6.2 Surfaces

6.2.1 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 $m \times n$ (on comprendra pourquoi très bientôt).

Si on a en tout $m \times n$ points, cela signifie que l'on a $m \times n$ valeurs de x, $m \times n$ valeurs de y et $m \times n$ valeurs de z. Il apparaît donc que abscisses, ordonnées et cotes des points de la surface peuvent être stockés dans des tableaux de taille $m \times n$.

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*)

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

Il reste maintenant à construire ces tableaux. Prenons tout d'abord le cas de la surface définie par $z = x^2 + y^2$ dont on veut tracer la surface représentative sur $[-1, 1] \times [-2, 2]$. Pour définir un quadrillage de ce rectangle, il faut définir une suite de valeurs $x_1, \ldots x_m$ pour x et une suite de valeurs $y_1, \ldots y_n$ 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    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 $m \times n$ points dans le plan (x, y). Il faut maintenant construire deux tableaux, l'un contenant les $m \times n$ abscisses de ces points l'autre les $m \times n$ ordonnées, soit :

$$X = \begin{bmatrix} x_1 & x_2 & \ldots & x_m \\ x_1 & x_2 & \ldots & x_m \\ \vdots & \vdots & & \vdots \\ x_1 & x_2 & \ldots & x_m \end{bmatrix} \qquad Y = \begin{bmatrix} y_1 & y_1 & \ldots & y_1 \\ y_2 & y_2 & \ldots & y_2 \\ \vdots & \vdots & & \vdots \\ y_n & y_n & \ldots & y_n \end{bmatrix}$$

Rassurez-vous pas besoin de boucle, la fonction **meshgrid** s'en charge :

>> [X,Y] = meshgrid(x,y);

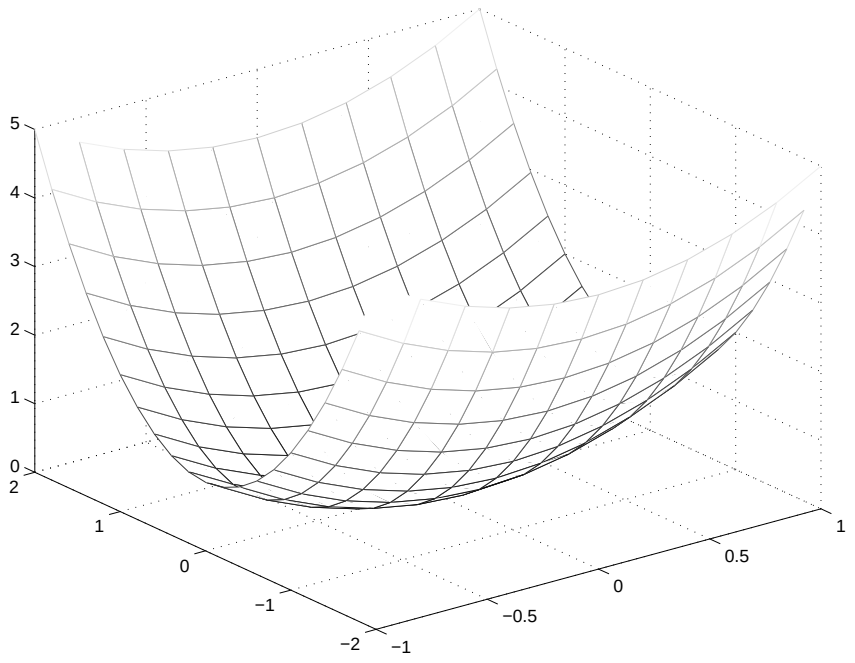
Il reste maintenant à calculer les $z = xy$ correspondants. C'est là que les calculs terme à terme sur les matrices montrent leur efficacité : on applique directement la formule aux tableaux `X` et `Y`, sans oublier de mettre un point devant les opérateurs `*`, `/` et `^`

```
Z = X .^2 +  Y.^2;
```

6.2.2 Tracé de la surface

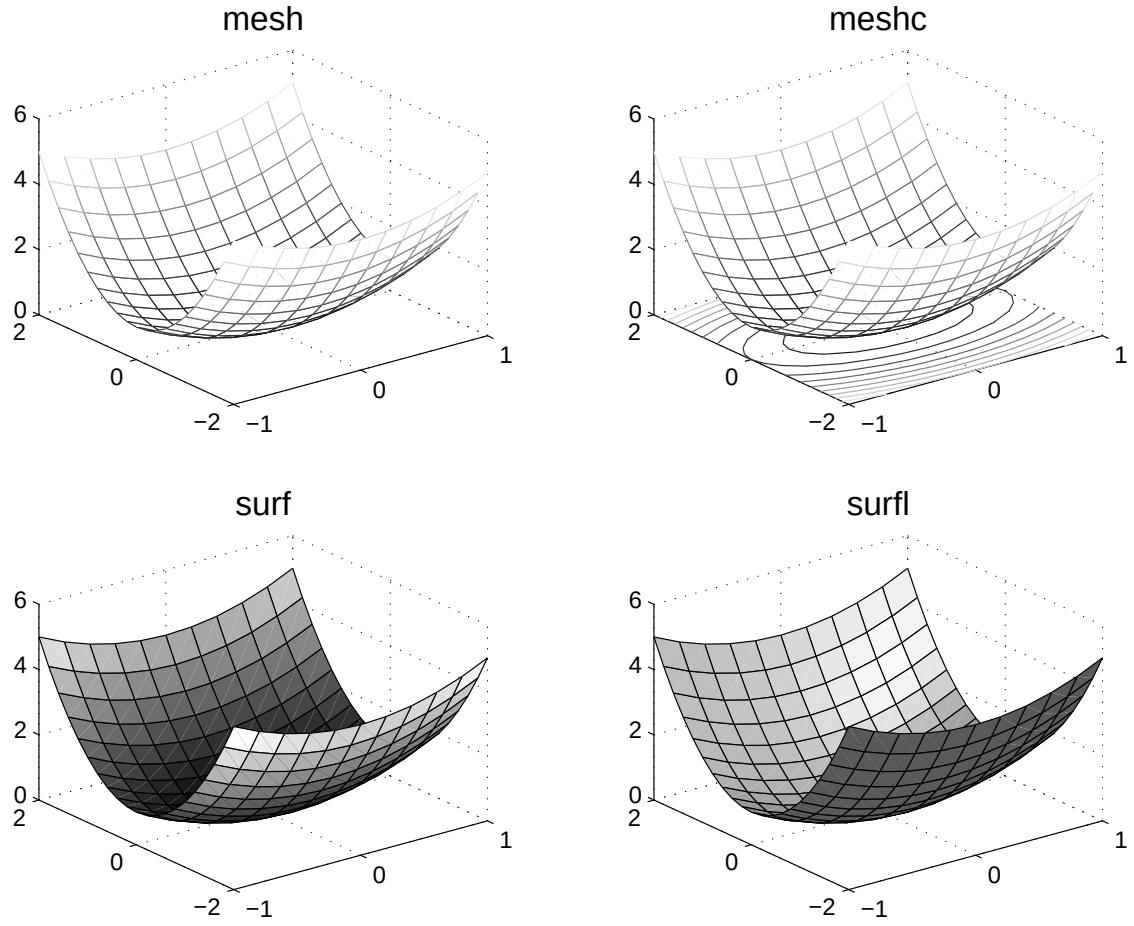
Ensuite on peut utiliser toutes les fonctions de tracé de surface, par exemple **mesh** :

>> mesh(X,Y,Z)



Les instructions les plus courantes sont
- **mesh** qui trace une série de lignes entre les points de la surface en mode «lignes cachées»
- **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)

Voilà le résultat pour les 4 premières [1] :



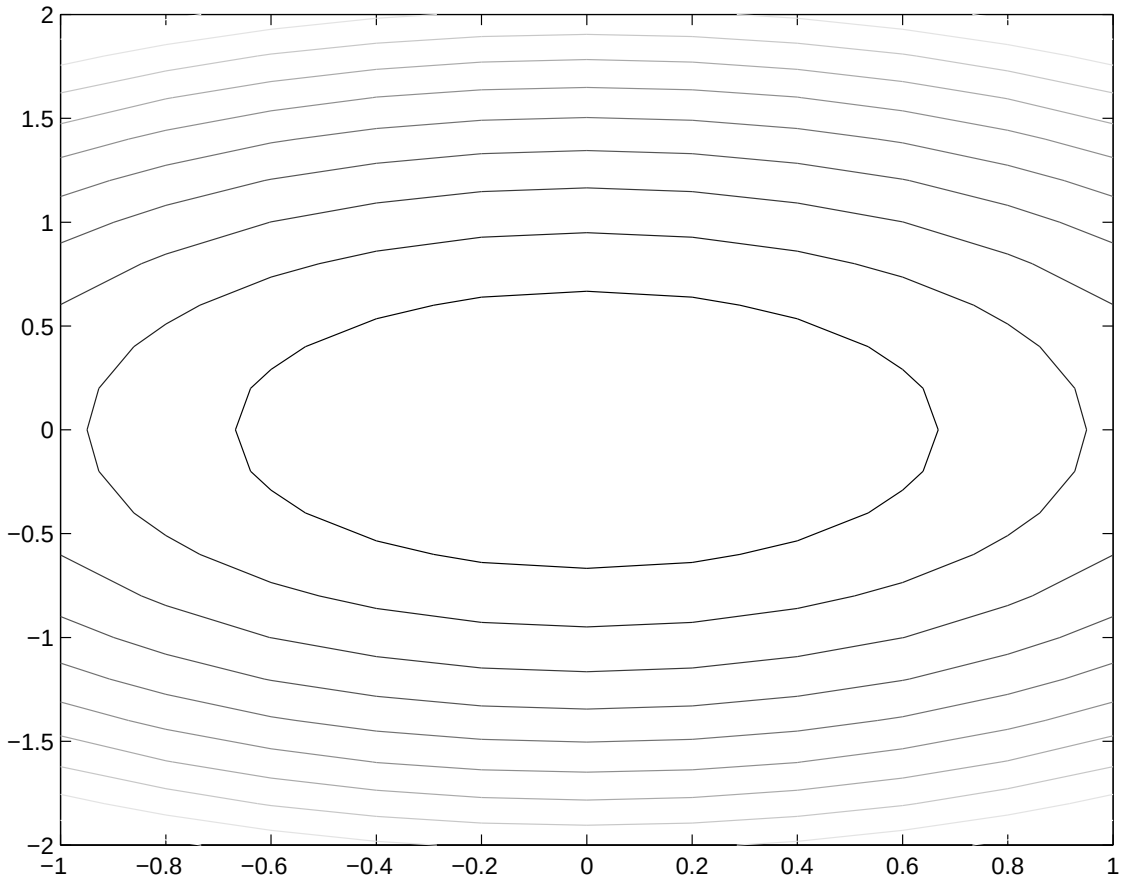
Notons enfin qu'il existe des fonctions de conversion entre les coordonnées cartésiennes, cylindriques et sphériques, permettant de tracer facilement des courbes définies dans l'un de ces systèmes de coordonnées. On regardera par exemple la documentation de **cart2pol**.

6.2.3 Courbes de contour

C'est 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)

[1] Ce graphique est effectué avec **subplot** !



6.2.4 Contrôle de l'angle de vue

Il existe la commande **view**, mais le plus simple est de  fenêtre graphique. 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 beacoup de temps se réafficher.

Chapitre 7

Echanges entre MATLAB et l'extérieur

7.1 Sauvegarde de données

Ce sont les 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.

La commande **diary** permet d'activer l'écho des commandes et résultats dans un fichier :

diary *nom_fichier*

Pour désactiver, taper **diary off**.

7.2 Importer des tableaux

MATLAB est souvent utilisé comme logiciel d'exploitation de résultats. Exemple classique : on a un programme FORTRAN 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, contenant chacune m lignes :

$$\begin{array}{llll} x_1 & y_1 & z_1 & \dots \\ x_2 & y_2 & z_2 & \dots \\ x_3 & y_3 & z_3 & \dots \\ \dots & & & \\ x_n & y_n & z_n & \dots \end{array}$$

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 s'appelle `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.

7.3 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. Le format PostScript est notamment disponible et fonctionne par exemple très bien ensuite avec la commande `\epsfig` de LaTeX(ce document en est la preuve).

Chapitre 8

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 documentation en ligne MATLAB. De même, les concepts théoriques sont supposés connus par le lecteur et il ne sera pas fait mention des méthodes utilisées.

8.1 Recherche des zéros d'une fonction

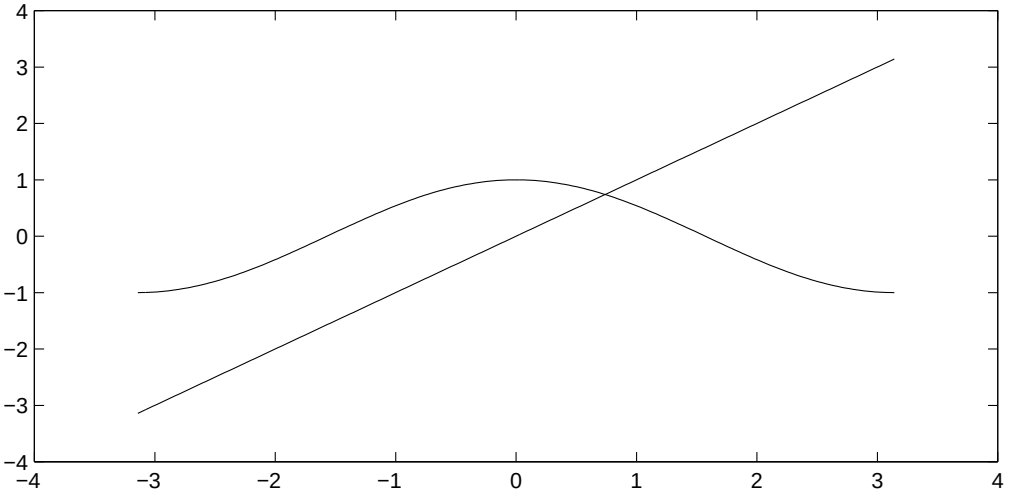
Problème : On cherche x_0 tel que $f(x_0) \simeq 0$.

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 de x_0. L'efficacité de l'algorithme est comme toujours dépendante de la valeur estimée choisie. La fonction f peut-être définie par une directive **inline** ou bien écrite dans une fichier.

Par exemple on cherche le zero 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 $x \to x$ et $x \to \cos x$ 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)
Zero found in the interval: [-0.9051, 0.9051].

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)
Zero found in the interval: [-0.9051, 0.9051].

ans =

    0.7391
```

ATTENTION ! On remarquera ici que le symbole **f** est mis entre quotes. 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 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 répètera à chaque fois la même instruction d'affectation de **m**.

La bonne solution est d'utiliser la directive **global** (voir section 5.2.3). 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.

8.2 Interpolation

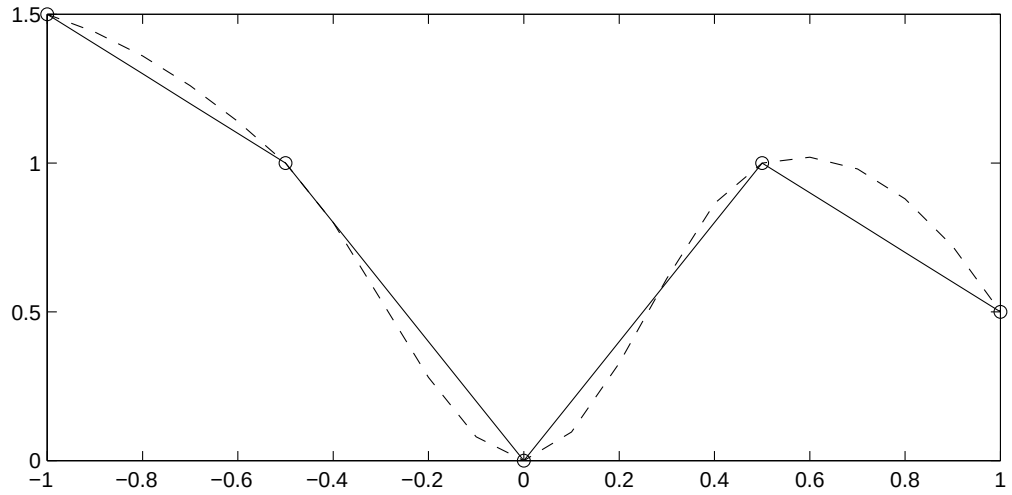
Le problème : connaissant des points tabulés (x_i, y_i), construire une fonction polynômiale par morceaux passant par ces points. En termes plus simples, c'est ce que fait la fonction **plot** quand elle «relie» les points que vous lui donnez : elle fait passer un polynôme de degré 1 entre deux points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

La fonction **interp1** résoud 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 cinq points (xi, yi) et on cherche une interpolation f(x) entre ces cinq points.

```
>> xi = [-1 -1/2 0 1/2 1];
>> yi = [1.5 1 0 1 1/2];
>> x = -1:0.1:1;
>> ylinear = interp1 (xi, yi, x);
>> ycubic = interp1 (xi, yi, x, 'cubic');
>> plot(xi,yi,'*', x,ylinear,'-', x,ycubic,'--')
```

La figure suivante illustre les interpolations linéaires (en traits pleins) et cubiques (en traits pointillés) obtenues par **interp1** sur ces 5 points (illustrés par des cercles). Notons que l'interpolation cubique ne fonctionne que pour des nombres de points impairs et fournit une interpolation de classe $\mathcal{C}^1$.



8.3 Approximation (estimation de paramètres ou «fitting»)

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.

8.3.1 Linéaire

Supposons que l'on dispose de n données expérimentales (x_i, y_i) et que l'on cherche à déduire une loi $y = f(x)$ dépendant *linéairement* de m coefficients :

$$y = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_m f_m(x)$$

Idéalement, les (x_i, y_i) vérifieraient cette relation, auquel cas on aurait le système de n équations linéaires à m inconnues $(a_1, \ldots a_m)$:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2) \\ \vdots & \vdots & & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{8.1}$$

soit sous forme matricielle $Y = F.A$ auquel cas on trouverait simplement les coefficients a_i par inversion :

$$A = F^{-1}.Y$$

En MATLAB, ce genre d'instruction peut se programmer simplement à l'aide de l'opérateur \ , et on écrirait alors

```
A = F  Y;
```

Cela dit le nombre d'équations n'est pas égal au nombre d'inconnues dans le système linéaire ci-dessus, il est (normalement) supérieur, et le système est surcontraint. Ca ne fait rien, l'opérateur \ le résoud alors «au mieux» c'est-à-dire qu'il trouve les a_i qui minimise la somme des carrés des résidus :

$$\min_{a_1 \ldots a_m} \sum_{i=1}^{n} \left[y_i - \sum_{j=1}^{m} a_j f_j(x_i) \right]^2$$

Voici un exemple : on cherche à fitter[1] des points expérimentaux (x_i, y_i) par une fonction de la forme :

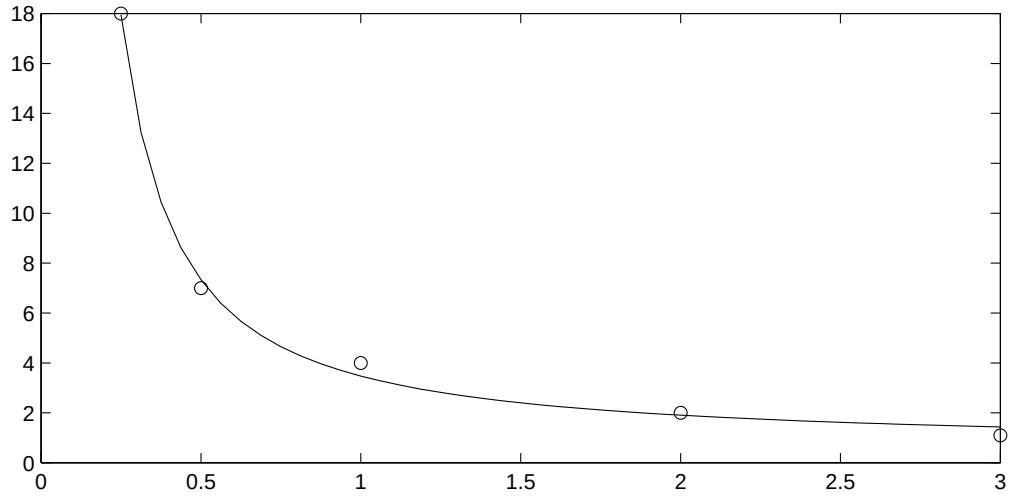
[1] Pardon pour l'anglicisme mais tout le monde l'utilise et il n'y a pas d'équivalent en français...

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

La séquence d'instructions est la suivante :

```
>> xi = [1/4 1/2 1 2 3]; xi = xi'; % on définit les abscisses et ordonnees
>> yi = [18 7 4 2 1.1]; yi = yi';  %                        en colonnes

>> F = [ones(size(xi)) 1./xi 1./(xi.^2)]; % on peut écrire aussi
                                          % xi./xi a la place de
                                          % ones(size(xi))


>> 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))
```

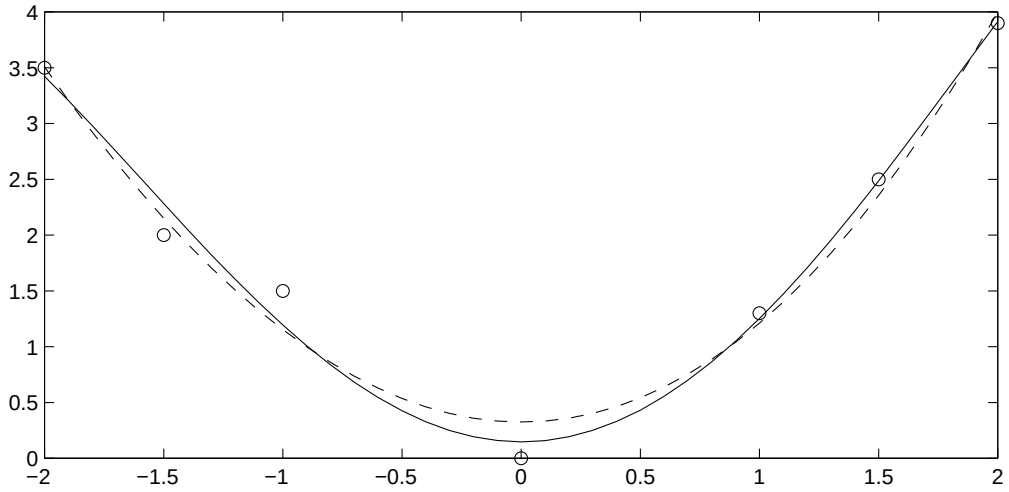
Ce programme trace la courbe fittée ainsi que les points expérimentaux :



Cette méthode marche dès que la fonction cherchée dépend linéairement des coefficients inconnus (le cas opposé est présenté dans la section suivante). Ça marche en particulier pour faire de l'approximation polynômiale (les f_j sont des mônômes x^j), 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 (voir la doc). **polyfit** attend en entrée simplement les x_i, les y_i 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), '--' )
```



8.3.2 Non-linéaire

On cherche maintenant à fitter des points expérimentaux (x_i, y_i) par une fonction $y = f(x)$ dépendant *non-linéairement* de m coefficients :

$$y = f(a_1, a_2, \ldots a_m; x) \tag{8.2}$$

Il s'agit ensuite de minimiser la distance entre cette fonction et les points expérimentaux, soit

$$\min_{a_1 \ldots a_m} F(a_1 \ldots a_m) \tag{8.3}$$

avec

$$F(a_1 \dots a_m) = \sum_{i=1}^{n} [y_i - f(a_1, a_2, \dots a_m; x_i)]^2 \tag{8.4}$$

La fonction F dépend non-linéairement des paramètres $a_1 \dots a_m$ et un algorithme d'optimisation doit être utilisé. Nous renvoyons le lecteur à un cours sur le sujet pour en acquérir les bases théoriques. Il faut simplement savoir que MATLAB possède des routines d'optimisation utiles pour ce genre de problème.

Si vous ne disposez pas de la toolbox «Optimisation», il vous faudra utiliser la routine de minimisation fournie dans la version MATLAB de base qui s'appelle **fminsearch**.

Pour cela il faut simplement programmer la fonction F à minimiser, en l'occurrence (8.4). Nous conseillons d'écrire la fonction de fitting f (8.2) dans une sous-fonction séparée.

Voici comment procéder dans l'exemple suivant (très classique) : des analyses spectrométriques fournissent des pics que l'on cherche à fitter (par exemple) par des gaussiennes. Par exemple dans le cas de deux pics :

$$y = B_1 \exp\left[-\left(\frac{x - x_1}{\sigma_1}\right)^2\right] + B_2 \exp\left[-\left(\frac{x - x_2}{\sigma_2}\right)^2\right]$$

Il y a 6 paramètres [2] B_1, B_2, x_1, x_2, σ_1 et σ_2 que nous regroupons dans un tableau **a** avec la correspondance suivante :

a_1	B_1
a_2	B_2
a_3	σ_1
a_4	σ_2
a_5	x_1
a_6	x_2

Ecrivons tout d'abord la fonction de fitting :

[2] Notons que dans ce genre de problème les centres des gaussiennes x_1 et x_2 sont souvent connus et correspondent à des longueurs d'ondes de raies de base, de telle sorte que le nombre de paramètres dans cet exemple serait réduit à 4. De façon générale il est toujours souhaitable de diminuer le nombre de paramètres $a_1 \dots a_m$.

```
function y = f (a, x)

y = a(1)*exp(- ( (x-a(5))/a(3) ).^2) + a(2)*exp(- ( (x-a(6))/a(4) ).^2)
```

Il est important que cette fonction puisse accepter un tableau en entrée, d'où l'usage des points devant les opérateurs puissance. Ensuite on écrit la fonction F représentant le résidu (8.4) à optimiser :

```
function out = F (a)

global xi yi

out = sum ( (yi - f(a,xi)).^2 );
```

L'usage de **global** pour passer les points expérimentaux **xi yi** est nécessaire car l'entête de la fonction F à optimiser est imposée par **fminsearch**.

Il reste à appeler l'optimiseur dans un programme principal :

```
% Pour passer les points expérimentaux à F
global xi yi

% Definition des points expérimentaux
% (en général chargés depuis un fichier)

xi = ...
yi = ...

% Estimation de la solution. Bien souvent le succès de l'opération
% dépend cette initialisation. Attention à ne pas rentrer des valeurs
% farfelues. Noter de plus que les parapètres doivent être entrés
% dans l'ordre défini plus haut.

a0 = [1 400 5 2 800 10] % correspond à B1, sigma1, x1, B2, sigma2, x2

% Appel de l'optimiseur. Attention aux '' autour de F

asol = fminsearch ( 'F', a0);

% (Facultatif mais en général utile)
% Superposition points experimentaux et fittés.

plot (xi, yi, '*', xi, f(asol, xi))
```

Voilà l'utilisation de base. Notez que l'algorithme utilisé par **fminsearch** est assez basique et est parfois insuffisant.

Si vous disposez de la toolbox «optimisation» (ce que nous conseillons si vous avez de nombreux problèmes de ce type), un grand nombre de routines d'optimisation vous est fourni. Toutes utilisent le même moteur d'optimisation qui propose de nombreux algorithmes possibles. Ce moteur est décliné sous forme de plusieurs fonctions, chacune étant dédiée à un problème bien particulier. Pour le problème d'estimation de paramètres proposé ci-dessus, la routine dédiée est **lsqcurvefit** (qui signifie «least squares curve fit»). Elle permet de ne programmer que la fonction f donnée par (8.2) et forme elle-même la somme des carrés.

Voici ce qu'il faut écrire pour résoudre le problème précédent :

```
function y = f (a, x)

y = a(1)*exp(- ( (x-a(5))/a(3) ).^2) + a(2)*exp(- ( (x-a(6))/a(4) ).^2)
```

C'est en fait la même que précédemment. Maintenant on peut appeler directement **lsqcurvefit** :

```
% Definition des points expérimentaux
% (en général chargés depuis un fichier)
xi = ...
yi = ...

% Estimation de la solution. Bien souvent le succès de l'opération
% dépend cette initialisation. Attention à ne pas rentrer des valeurs
% farfelues. Noter de plus que les parapètres doivent être entrés
% dans l'ordre défini plus haut.

a0 = [1 400 5 2 800 10] % correspond à B1, sigma1, x1, B2, sigma2, x2

% Appel de l'optimiseur. Attention aux '' autour de f
% On remarque que la fonction à passer est maintenant f et plus F
% et qu'on passe les points expérimentaux directement au solveur.

asol = lsqcurvefit ( 'f', a0, xi, yi);

% (Facultatif mais en général utile)
% Superposition points experimentaux et fittés.

plot (xi, yi, '*', xi, f(asol, xi))
```

Notons enfin que les caractéristiques du moteur d'optimisation (utilisation du jacobien ou non, algorithme, nombre max d'itérations, tolérance ...) peuvent être modifiées grâce à l'appel de la fonction **optimset**. Nous renvoyons le lecteur intéressé à la documentation.

8.4 Equations différentielles

Problème : on cherche à résoudre un système d'équations différentielles ordinaires (EDO, en anglais ODE) du premier ordre :

$$\begin{aligned}
\dot{y}_1 &= f_1(y_1, y_2, \ldots, y_n, t) \\
\dot{y}_2 &= f_2(y_1, y_2, \ldots, y_n, t) \\
&\ldots \\
\dot{y}_n &= f_n(y_1, y_2, \ldots, y_n, t)
\end{aligned}$$

avec les conditions initiales $y_1(0) = y_{10} \ldots y_n(0) = y_{n0}$

Rappelons au lecteur non averti que tout système ou équation différentielle d'ordre supérieur peut se ramener simplement à cette forme canonique, utilisée dans tous les solveurs d'EDO.

On voit donc que la définition d'un tel système repose sur la définition de n fonctions de $n+1$ variables. Ces fonctions devront être programmées dans une fonction MATLAB sous la forme canonique suivante :

```
function ypoint = f (y, t)

ypoint(1) = une expression de y(1), y(2)  ... y(n) et t
...
ypoint(n) = une expression de y(1), y(2)  ... y(n) et t

ypoint = ypoint';
```

On remarquera que les y_i et les $\dot{y}_i$ sont regroupés dans des vecteurs, ce qui fait que la forme de cette fonction est exploitable quel que soit le nombre d'équations du système différentiel.

La dernière ligne est (hélas) nécessaire car la fonction doit ressortir un vecteur colonne et non un vecteur ligne. C'est à mon sens un bug qui j'espère disparaîtra dans les versions ultérieures.

Ensuite, pour résoudre cette équation différentielle, il faut appeler un solveur et lui transmettre au minimum :

– le nom de la fonction.
– les bornes d'intégration ($t_{\min}$ et $t_{\max}$).
– les conditions initiales.

Le solveur fournit en sortie un vecteur colonne représentant les instants d'intégration t, et une matrice dont la première colonne représente les y_1 calculés à ces instants, la deuxième les y_2, et la nième les y_n.

L'appel du solveur prend donc en général la forme suivante :

```
[t, y] = ode45 ('f', [tmin tmax], [y10 y20 ... yn0] );

y1 = y(:,1);
y2 = y(:,2);
...
yn = y(:,n);

plot(t, y1, t, y2) % par exemple on trace y1(t) et y2(t)
plot(y1,y2)        % ou bien y2(y1) (plan de phase pour les oscillateurs)
```

Les lignes `y1 = ...` servent à extraire les différentes fonctions y_i dans des colonnes simples.

Nous avons utilisé ici **ode45** qui est un Runge-Kutta-Merson imbriqué d'ordre 4 et 5. C'est le plus courant et celui par lequel il faut commencer, mais il en existe d'autres, en particulier **ode15s** adapté aux systèmes raides (voir la doc).

Les spécialistes s'étonneront de ne pas avoir à spécifier d'erreur maximale admissible, relative ou absolue. Sachez que MATLAB prend une erreur relative max de 10^{-4} par défaut, et qu'il est toujours possible de modifier cette valeur, ainsi que bien d'autres paramètres grâce à la routine de gestion des options **odeset**.

Il est temps de passer à un exemple. On considère l'équation de Matthieu amortie :

$$\ddot{y} + b\dot{y} + a(1 + \epsilon \cos t)y = 0$$

où a, b et ϵ sont des paramètres. On prend comme conditions initiales $y(0) = 10^{-3}$ et $\dot{y}(0) = 0$. En posant $y_1 = y$ et $y_2 = \dot{y}$ on se ramène à la forme canonique :

$$\dot{y}_1 = y_2$$
$$\dot{y}_2 = -by_2 - a(1 + \epsilon \cos t)y_1$$

Ecrivons la fonction **matthieu** définissant cette équation dans un fichier **matthieu.m**. Comme pour le problème de recherche de zéro, on passera généralement les paramètres de l'équation dans une directive **global** :

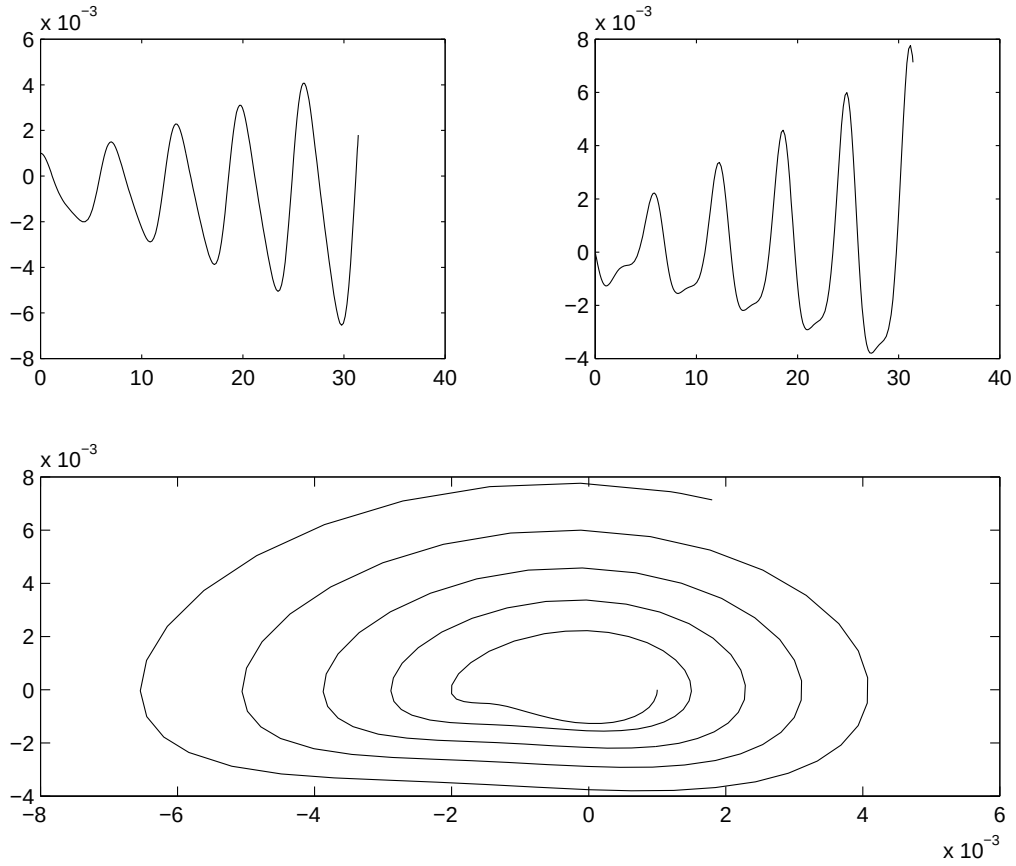
```
function ypoint = matthieu ( y, t)

global a b epsilon

ypoint(1) = y(2);
ypoint(2) = -b*y(1) -a*(1+epsilon*cos(t))*y(2);

ypoint = ypoint';
```

Pensez à mettre des ; à la fin de chaque ligne si vous ne voulez pas voir défiler des résultats sans intérêt.

La séquence d'instructions (à mettre dans un autre fichier **.m**) qui appelle le solveur sera par exemple :

```
global a b epsilon

% Parametres
a = 1;
b = 0.1;
epsilon = 1;

% Temps final
tfinal = 10*pi;

% Conditions initiales
y01 = 1e-3;
y02 = 0;

[t,y] = ode45('matthieu', [0 tfinal], [y01 y02]); % Resolution

y1 = y(:,1);  % Extraction de y1 et y2
y2 = y(:,2);

subplot(221)
plot(t,y1)    % y1 fonction du temps
              %      (represente y(t) pour l'EDO originale)
subplot(222)
plot(t,y2)    % y2 fonction du temps
              %      (represente dy(t)/dt pour l'EDO originale)
subplot(212)
plot(y1,y2)   % Plan de phase
```

Ce programme trace la figure suivante qui représente les grandeurs $y(t)$ et $\dot{y}(t)$ de l'équation originale en fonction du temps, plus le plan de phase. Au passage, on retrouve bien l'instabilité des solutions de l'équation de Matthieu pour les valeurs des paramètres choisis.


x 10^−3
6
4
2
0
−2
−4
−6
−8
0
10
20
30
40
x 10^−3
8
6
4
2
0
−2
−4
0
10
20
30
40
x 10^−3
8
6
4
2
0
−2
−4
−8
−6
−4
−2
0
2
4
6
x 10^−3

Conclusion

Les concepts exposés ici sont plus que suffisants pour résoudre la majorité des problèmes. Evidemment il faut ensuite apprendre à aller chercher dans la documentation la fonction dont on a besoin, et pratiquer régulièrement.

Le but de ce document était de convaincre le lecteur que MATLAB pouvait remplacer avantageusement les tableurs pour traiter des problèmes scientifiques. Toute personne ayant un jour programmé des macros dans un tableur verra, je l'espère, que la syntaxe MATLAB est bien plus simple, pourvu que l'on se donne la peine d'essayer.

Rappelons que le prix d'une licence MATLAB est similaire à celui d'un tableur classique, et que des versions bridées pour les étudiants existent (seule la taille des tableaux est limitée).

Pour le lecteur souhaitant aller plus loin, on notera que MATLAB peut s'interfacer avec des routines FORTRAN ou C externes, ou bien être appelé depuis un programme écrit dans l'un de ces langages.

Par ailleurs, tout le graphique MATLAB est construit sous forme de structures de données, assez faciles à utiliser. Il est ainsi aisé de construire ses propres menus, masques de saisie, boutons et autres fioritures pour développer des outils très conviviaux.

Enfin, pour montrer que l'auteur de ces lignes n'est pas subventionné par la société diffusant MATLAB, on notera qu'il existe des «clônes» domaine public.

– SCILAB
  `http ://www-rocq.inria.fr/scilab/scilab.html`
  développé par l'INRIA donc français. La compatibilité avec MATLAB est excellente notamment pour les strcutures de programmation. En revanche le graphique est tout-à-fait déroutant et pénible, ce qui gache un excellent produit par ailleurs. Des tiers ont cependant développés des packages de compatibilité graphique avec MATLAB.
– OCTAVE
  `http ://www.octave.org`
  distribué sous licence GNU. Le plus compatible avec la syntaxe MATLAB. Le graphique

semble satisfaisant. Installé à l'EMAC.
– RLAB
`http ://www.eskimo.com/~ians/rlab.html`
très inspiré par la programmation en C.