Matlab : applications en mécanique

LA207

Notes de cours

Université Pierre et Marie Curie

Jérôme Hoepffner

Janvier 2014

# Table des matières

# Chapitre 1

# Notes de cours

## 1.1 Introduction

Vous trouverez dans ces notes une base de connaissances et de pratiques pour utiliser Matlab. C'est, réuni en quelques pages, le minimum dont vous aurez besoin pour le cours "Matlab : applications en mécanique". Aujourd'hui, il est difficile de se passer de l'ordinateur pour faire de la science : pour analyser les données, les représenter, les manipuler. Pour ceci Matlab est un outil puissant.

Une grande partie de l'enseignement de l'informatique en licence de mécanique consiste à utiliser l'ordinateur pour calculer des solutions approchées aux équations différentielles de la mécanique. Ces équations décrivent par exemple la trajectoire d'un objet soumis à des forces, ou bien des choses bien plus complexes comme le champ de vitesse et de pression dans l'écoulement d'un fluide. Ces techniques sont basées sur la discrétisation des équations différentielles. Une étape technique importante pour cela consiste à résoudre des systèmes d'équations linéaires. Ces choses passionnantes vous les pratiquerez l'année prochaine en L3.

Dans notre cours, nous utilisons l'ordinateur pour faire de la science. Un cas typique d'étude : vous avez filmé la chute d'un objet et vous voulez savoir si ce que vous avez filmé respecte la loi de la chute des corps, et vérifier si le frottement avec l'air y joue un rôle important. A partir du film, vous pouvez obtenir (en mesurant la position de l'objet sur l'image) la position en fonction du temps. Cette courbe, si vous la tracez devrait donner une parabole. Vous pouvez à partir de cette trajectoire mesurée, calculer l'énergie cinétique et potentielle de gravité de votre objet, et vérifier que l'énergie totale est constante. Si le frottement joue un rôle important, alors l'énergie totale va décroître. Voilà : au cours de cette étude, vous avez pris des mesures, vous avez représenté ces mesures sous la forme de graphiques, à partir de ces mesures vous avez calculé d'autres quantités qui sont significatives pour la mécanique, et vous avez comparé ces mesures à des modèles théoriques : le principe fondamental de la dynamique avec ou sans frottement. Ces étapes sont les étapes typiques de la science. Ce sont les étapes

qui sont suivies par les ingénieurs et les chercheurs.

Pour cela vous avez manipulé des images, vous avez enregistré des données sous forme de tableaux. Vous avez effectué des opérations mathématiques sur ces tableaux : addition, multiplication, puissances. Vous avez calculé les valeurs théoriques de la position de l'objet en chute en utilisant une formule mathématique. Vous avez tracé un graphique, avec un titre et des labels. Savoir faire toutes ces choses vous sera très utile par la suite.

## 1.2 Matlab de base

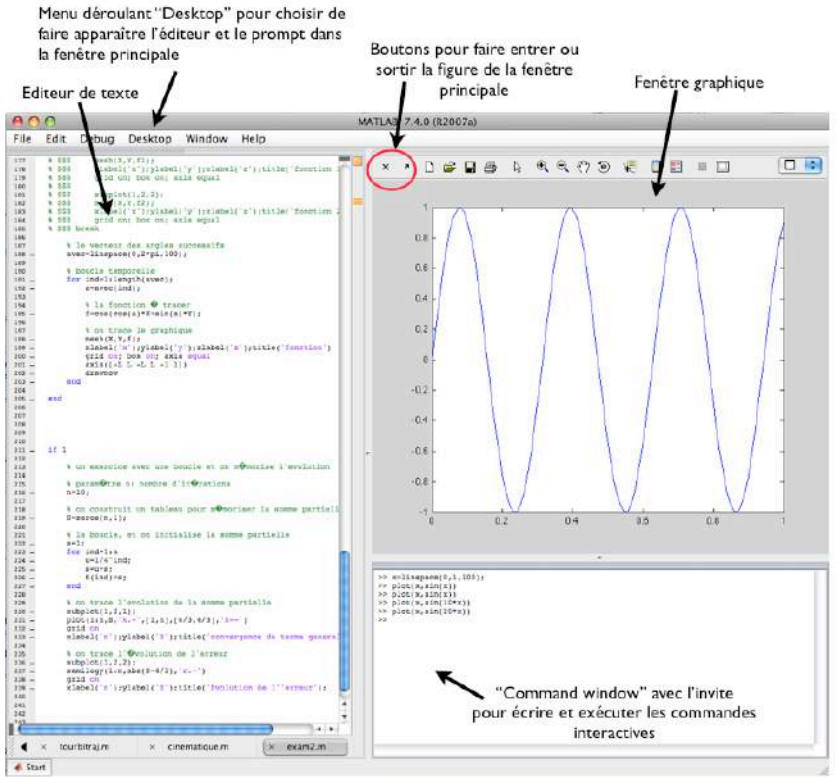
### 1.2.1 L'interface graphique de Matlab

Matlab propose une interface graphique, c'est la fenêtre qui apparaît lorsque vous lancez matlab. Vous disposez de plusieurs outils que vous pouvez disposer dans cette fenêtre graphique. Pour sélectionner ces outils, cliquer sur le menu déroulant "Desktop". Vous avez ainsi la liste de ces outils. Nous allons utiliser trois outils : l'éditeur de texte ("editor"), la fenêtre graphique ("figures"), et la fenêtre de commande ("commande window").

L'éditeur de texte nous permet d'écrire des commandes dans un fichier texte et de l'enregistrer sur le disque dur. Ces fichiers qui contiennent les commandes de nos programmes, nous les appelons des "scripts". C'est dans la fenêtre graphique que vont s'afficher les figures que nous allons tracer. Et dans la fenêtre de commande, nous pouvons écrire des lignes de commandes pour un contrôle interactif de matlab, et c'est dans cette fenêtre que Matlab nous écrit les messages d'erreurs.

Je vous conseille la configuration ci-dessous qui permet de voir à la fois l'éditeur pour votre script, l'invite ou vous verrez les messages d'erreur, en même temps que votre figure.






Pour supprimer un outil de la fenêtre graphique, cliquez sur la croix en haut de cet outil. Vous pouvez choisir de garder les outils à l'intérieur de la fenêtre principale ou bien vous pouvez les avoir dans une fenêtre extérieure. Pour cela, utiliser les petites flèches à côté de la croix. Faire sortir un outil de la fenêtre principale, cela s'appelle "Undock", et remettre un outil dans la fenêtre principale, cela s'appelle "Dock".

Pour réorganiser les outils dans la fenêtre principale, vous pouvez cliquer sur la barre de menu d'un outil, et le faire glisser, Matlab vous proposera différentes configurations en fonction de comment vous faites glisser la sous-fenêtre.

### 1.2.2 Syntaxe

Matlab offre une interface via laquelle on peut entrer des commandes, c'est "l'invite", ou en anglais, le "prompt" : la ligne qui commence avec >> dans la fenêtre de commande. On y écrit les lignes de commandes qui vont créer des tableaux, les manipuler, tracer des graphiques... Une fois que la ligne de commande est écrite, on tape sur "entrée" et l'ordinateur évalue cette commande, vérifie qu'elle est conforme, qu'elle veut dire quelque chose. Si il y a un problème : on appelle un tableau qui n'existe pas, on utilise un nom de fonction qui n'existe

pas, on demande quelque chose qui n'est pas possible, il y a un message d'erreur qui nous donne une indication sur le problème. Il faut lire ces messages et corriger l'erreur.

Si on crée un tableau, avec un nom, par exemple

```
>> A=[0.1,5,-2];
```

c'est un tableau ligne (une seule ligne et trois colonnes). Alors ce tableau est désormais disponible dans le "workspace", c'est à dire dans l'espace mémoire de matlab. On peut l'utiliser en l'appelant par son nom, par exemple ici on crée un tableau `B` dans lequel on met deux fois la valeur des éléments qu'il y a dans `A`

```
>> B=2*A;
```

Ici, je mets un point-virgule à la fin de la ligne pour dire à matlab de ne pas afficher à l'écran le résultat de l'opération. Si je ne le met pas j'obtiens :

```
>> B=2*A
0.2 10 -4
```

qui est bien le résultat escompté.

Je peux demander à matlab de me donner la liste des variables, ou tableaux qui sont dans le workspace avec la fonction `who`

```
>> who
A ...
B ...
```

Ici j'ai les deux tableaux que j'ai créés. Si je veux éliminer un tableau du workspace, j'utilise la fonction `clear` en appelant le tableau par son nom. Ca peut être utile, notamment si les tableaux sont très grands et qu'il faut libérer de l'espace mémoire.

```
>> clear A
```

Le deux grandes différences entre Matlab et des langages de programmation tels que le fortran (FORmula-TRANslation), et le `c`, c'est 1) l'interactivité, et 2) on n'a pas besoin de déclarer les variables avant de les utiliser ni de mettre d'en têtes.

Matlab est un langage sensible aux lettres majuscules et minuscules : `A` n'est donc pas la même variable que `a`.

### 1.2.3 Graphiques

Avec la ligne de commande, on peut créer des graphiques qui représentent les tableaux. Par exemple avec la fonction plot pour un graph en ligne (un vecteur en fonction d'un vecteur).

```
>> a=[2, 5, 6];
>> b=[ 3.2, 3.5, 5];
>> plot(a,b)
```



Ici je crée deux tableaux nommées `a` et `b`, et je trace les valeurs de `b` en ordonnée pour les valeurs de `a` en abcisse. Ce graph apparaît dans une nouvelle fenêtre. On peut utiliser des fonctions qui ajoutent des détails à ce graph :

```
>> xlabel('a'); ylabel('b'); xlim([0,8]); ylim([0,6])
```

Ici j'ai mis 'a' et 'b' en label des abcisses (axe horizontal, axe des "x") et des ordonnées (axe vertical, axe des "y"), et j'ai changé les limites des axes x et y à ma convenance. Notez que ici j'ai mis plusieurs commandes sur la même ligne d'invite en les séparant par des point-virgules.

### 1.2.4 Scripts

Si on veut garder une série de commandes, plutôt que de les écrire à chaque fois à l'invite, on peut les écrire dans un fichier texte. Ces fichier s'appellent des "scripts". Par exemple je crée un fichier nommé "test.m" avec l'éditeur de texte de matlab, et je l'enregistre dans le dossier "actif", c'est à dire le dossier dans lequel matlab cherche les scripts. L'extension `.m` signifie que ce fichier est un fichier de commandes Matlab.

Si à l'invite, j'écris le nom de mon fichier, sans l'extension `.m`, alors matlab va exécuter les unes après les autres les commandes que j'ai écrites dans ce fichier, comme si je les avait écrites une à une à l'invite. Voici un script simple. Je peut sauter des lignes sans soucis, elles seront ignorées.

```
% voici mon script

a=[2, 5, 6];
b=[ 3.2, 3.5, 5];

plot(a,b) % je trace le graphique
```

Les caractères qui suivent `%` ne sont pas interprétés, ce sont des commentaires. Ils sont utiles pour se souvenir de ce que font les blocs de commandes, ou bien pour qu'un utilisateur qui n'a pas codé lui même puisse plus facilement comprendre ce que fait le programme.

### 1.2.5 Répertoire courant

On execute les commandes qui sont dans un script en écrivant le nom du script à l'invite de Matlab. Pour cela, il faut que le script soit dans le répertoire courant, c'est à dire dans le repertoire actif. Les commandes associées au répertoire courant sont comme pour le terminal sous linux :

- `pwd` : "Print Working Directory" affiche à l'écran l'adresse du répertoire courant.
- `ls` : "List Directory" affiche la liste des fichiers qui sont présents dans le répertoire courant. Avec un peu de chance, votre script s'y trouve.
- `cd` : "Change Directory" pour se déplacer dans l'arborescence de répertoires.

Pour pouvoir exécuter votre script, il faut ou bien mettre le script dans le répertoire courant, ou bien changer le répertoire courant pour aller là ou se trouve votre script. Une autre manière peut-être plus simple consiste à exécuter le script directement à partir de l'éditeur Matlab, avec le bouton d'exécution du menu de l'éditeur. Si le script en question n'est pas dans le répertoire courant, Matlab vous propose de changer le répertoire courant vers le répertoire dans lequel se trouve le script.

### 1.2.6 Boucles et tests

Pour réaliser des actions répétitives, on peut utiliser la boucle `for`. J'écris par exemple les commandes suivantes dans un fichier :

```
a=7
for i=1:10
  a*i
end
```

Je crée un tableau à une ligne et une colonne, nommé `a`, et puis pour l'indice `i` valant successivement de 1 à 10, je vais afficher à l'écran la valeur de a*i. Plutôt que de créer le vecteur lors de l'appel de `for`, je peux créer ce vecteur à l'avance, ici sous le nom `vec`, c'est équivalent et parfois pratique :

```
a=7
vec=1:10
for i=vec
  a*i
end
```

Je peux utiliser la structure `if` pour n'exécuter des commandes que si une condition est satisfaite :

```
a=2; b=3;
if a>b
  disp('a est plus grand que b')
else
  disp('a est plus petit que b')
end
```

Il faut bien noter ici que `a>b` est une valeur binaire : vrai ou faux. Dans matlab, vrai, c'est `1` et faux c'est `0`. Ici, la fonction `disp` affiche à l'écran la chaîne de caractère qui lui est donnée en argument. On signale une chaîne de caractère avec les guillemets `' '`. Comme pour la boucle `for` ci-dessus, le test sur lequel est fait le `if` peut être dans une variable :

```
a=2; b=3;
test=a>b
if test
  disp('a est plus grand que b')
```



```
else
  disp('a est plus petit que b')
end
```

Ici la variable `test` est un scalaire (un tableau à une ligne et une colonne) dont la valeur est 1 si `a` est plus grand que `b` et 0 si `a` est plus petit que `b` ou égal à `b`. Notez que l'expression `if test` signifie "si `test` est vrai" et est un raccourci pour l'expression plus explicite `if test==1`. L'opérateur `==` est le test d'égalité, comme `>` est le test "plus grand que" et `<` est le test "plus petit que". Il ne faut pas confondre le test d'égalité `==` avec l'opérateur `=` qui assigne une valeur à une variable. Le test "n'est pas égal" s'écrit `~=`, puisque `~` est l'opérateur de négation : si `a` est vrai, alors `~a` est faux.

### 1.2.7 Fonctions simples de matlab

Il y a beaucoup de fonctions dans matlab, auxquelles on donne des "arguments d'entrée" et qui nous rendent des "arguments de sortie". Ce qui se passe à l'intérieur de la fonction ne nous intéresse pas et peut être très complexe, ce qui nous intéresse c'est ce que la fonction rend. Parmi ces fonctions, il y a les fonctions mathématiques de base :

```
>> a=sin(2.3);
```

c'est la fonction sinus. Ici, la variable `a` reçoit la valeur du sinus de 2.3 radiants. De la même manière, nous avons les fonctions

```
cos: cosinus
tan: tangente
exp: exponentielle
sqrt: racine carrée
log: logarithme
abs: valeur absolue
sinh: sinus hyperbolique
cosh: cosinus hyperbolique
tanh: tangente hyperbolique
erf: fonction erreur
...
```

et ainsi de suite. Pour le moment, nous avons donné des arguments d'entrée scalaires, mais on verra plus tard que si on donne un tableau en arguments d'entrée, ces fonctions donnent en sortie des tableaux de la même taille en appliquant la fonction mathématique pour chaque élément du tableau. On verra que cela ouvre de grandes perspectives...

Il y a d'autres genres de fonctions, qui peuvent être très utiles, comme par exemple

```
a=num2str(2.3)
```

qui transforme l'argument d'entrée en chaîne de caractère et la met dans la variable a. Une fois ceci fait, a contient la chaîne `'2.3'`. On imagine aisément ce que fait la fonction `str2num`.

### 1.2.8 Créer des fonctions

On peut créer nos propres fonctions. Pour cela, on crée un fichier dont le nom est le nom de la fonction, en mettant l'extension `.m` comme pour les scripts. Par exemple, le fichier `testfonction.m`. Ce fichier texte est comme un script, mais avec une en-tête particulière :

```
function [s1,s2]=testfonction(a,b,c)
s1=a+b;
s2=s1+c;
```

Cette fonction a trois arguments d'entrée, `a`, `b`, et `c`, et rend deux arguments de sortie `s1` et `s2`. Le type et le nombre des arguments sont arbitraires en entrée et en sortie, ils peuvent être des tableaux ou n'importe quoi d'autre. Ci-dessous un script qui utilise cette fonction :

```
toto=1
pilou=3
[p,r]=testfonction(toto,pilou,10);
disp(p)
disp(r)
```

où on a bien nos trois arguments d'entrée et nos deux arguments de sortie, puis on affiche `p` et `r` à l'écran.

### 1.2.9 L'aide matlab

Si vous avez oublié les détails d'une fonction ou d'un opérateur, utilisez la fonction `help`. Par exemple :

```
>> help for
 FOR    Repeat statements a specific number of times.
    The general form of a FOR statement is:

       FOR variable = expr, statement, ..., statement END

    The columns of the expression are stored one at a time in
    the variable and then the following statements, up to the
    END, are executed. The expression is often of the form X:Y,
    in which case its columns are simply scalars. Some examples
    (assume N has already been assigned a value).

         for R = 1:N
             for C = 1:N
                 A(R,C) = 1/(R+C-1);
             end
         end

    Step S with increments of -0.1
         for S = 1.0: -0.1: 0.0, do_some_task(S), end

    Set E to the unit N-vectors
         for E = eye(N), do_some_task(E), end

    Long loops are more memory efficient when the colon expression appears
    in the FOR statement since the index vector is never created.

    The BREAK statement can be used to terminate the loop prematurely.
```



```
    See also if, while, switch, break, continue, end, colon.

    Reference page in Help browser
       doc for
```

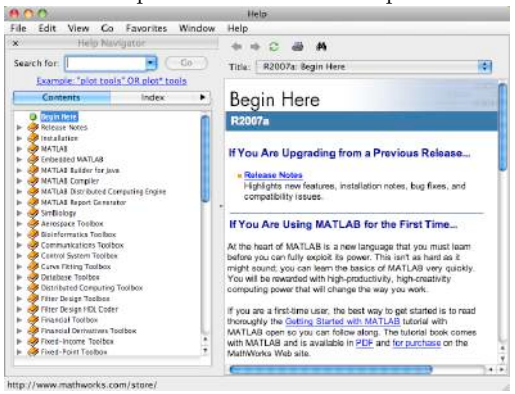
Un aspect très intéressant de cette aide, c'est qu'elle propose dans "see also" en bas de texte une liste de fonction et commandes proches de celle que vous avez demandé, ici : `if, while, switch, break, continue, end, colon`. En procédant d'aide en aide, vous pouvez rapidement apprendre beaucoup de fonctions et fonctionnalités, ce qui vous facilitera grandement la tâche.

Si vous vous demandez la manière dont fonctionne un caractère spécial, par exemple à quoi servent les deux points `:` ("colon" en anglais), tapez

```
>> help colon
help colon
 :  Colon.
    J:K  is the same as [J, J+1, ..., K].
    J:K  is empty if J > K.
    J:D:K  is the same as [J, J+D, ..., J+m*D] where m = fix((K-J)/D).
    J:D:K  is empty if D == 0, if D > 0 and J > K, or if D < 0 and J < K.

    COLON(J,K) is the same as J:K and COLON(J,D,K) is the same as J:D:K.

    The colon notation can be used to pick out selected rows, columns
    and elements of vectors, matrices, and arrays.  A(:) is all the
    elements of A, regarded as a single column. On the left side of an
    assignment statement, A(:) fills A, preserving its shape from before.
    A(:,J) is the J-th column of A.  A(J:K) is [A(J),A(J+1),...,A(K)].
    A(:,J:K) is [A(:,J),A(:,J+1),...,A(:,K)] and so on.

    The colon notation can be used with a cell array to produce a comma-
    separated list.  C{:} is the same as C{1},C{2},...,C{end}.  The comma
    separated list syntax is valid inside () for function calls, [] for
    concatenation and function return arguments, and inside {} to produce
    a cell array.  Expressions such as S(:).name produce the comma separated
    list S(1).name,S(2).name,...,S(end).name for the structure S.

    For the use of the colon in the FOR statement, See FOR.
    For the use of the colon in a comma separated list, See VARARGIN.

    Overloaded functions or methods (ones with the same name in other directories)
       help sym/colon.m

    Reference page in Help browser
       doc colon
```

Pour avoir la liste générale des noms associés aux caractères spéciaux, tapez par exemple `help :`.

Si plutôt que d'aide, vous avez besoin de documentation, utilisez la commande `doc` qui ouvre une fenêtre séparée.




Il y a beaucoup de choses dans Matlab. C'est en maîtrisant son `aide` que vous allez véritablement pouvoir l'utiliser comme un langage de haut niveau.

### 1.2.10 Caractères spéciaux

Ce sont les caractères qui jouent un rôle important pour la syntaxe, en voici une petite liste :

- `[ ]` Les crochets pour concaténer des tableaux
- `;` Le point-virgule ("semicolon" en anglais), à mettre à la fin d'une commande dont on ne veut pas le résultat affiché à l'écran. Sert aussi pour les concaténations de tableau : passage à la ligne suivante.
- `.` Le point c'est pour écrire les chiffres réel : `9.2`. Les anglophones mettent des points plutôt que des virgules.
- `:` Les deux points ("colon" en anglais), pour définir un vecteur à progression géométrique : `3:10` c'est la même chose que le tableau ligne `[3,4,5,6,7,8,9,10]`, de même, `0:0.1:1` c'est la même chose que `[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]`
- `,` La virgule, à la fin d'une commande s'il y a plusieurs commandes sur une ligne, si l'on veut l'affichage du résultat de la commande à l'écran. Sert aussi pour la concaténation des tableaux : séparation de case sans passage à la ligne.
- `==` Opérateur logique d'égalité
- `=` A lire "reçoit" et non pas "égal" : pour donner une valeur à une variable.
- `' '` Les guillemets pour définir une chaîne de caractères.
- `*` Multiplication de matrices.
- `.*` multiplication de tableaux : élément par élément.
- `./` Division de tableaux : élément par élément.

## 1.3 Tableaux

Dans cette section, on voit qu'on peut faire beaucoup de choses avec des tableaux, opérations qui vont bien nous servir par la suite lorsqu'ils contiendront des données intéressantes, pour l'analyse et les graphiques.

### 1.3.1 Construire des tableaux par concaténation

Le tableau le plus simple a une ligne et une colonne, c'est un scalaire

```
a=-3.7;
b=2;
```

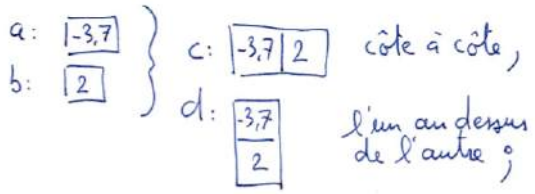
J'ai maintenant deux variables, `a` et `b` qui ont la même taille et contiennent chacun un nombre réel. Je peut concaténer ces deux tableaux pour faire un tableau plus grand

```
c=[a,b];
```

les crochets `[ ]` sont les symboles de concaténation. Cette opération met dans la variable `c` un tableau obtenu en mettant `a` et `b` "côte à côte", ceci étant spécifié par la virgule. Si je veux mettre `a` et `b` "l'un en dessous de l'autre", j'utilise les crochets avec un point-virgule



```
d=[a;b];
```

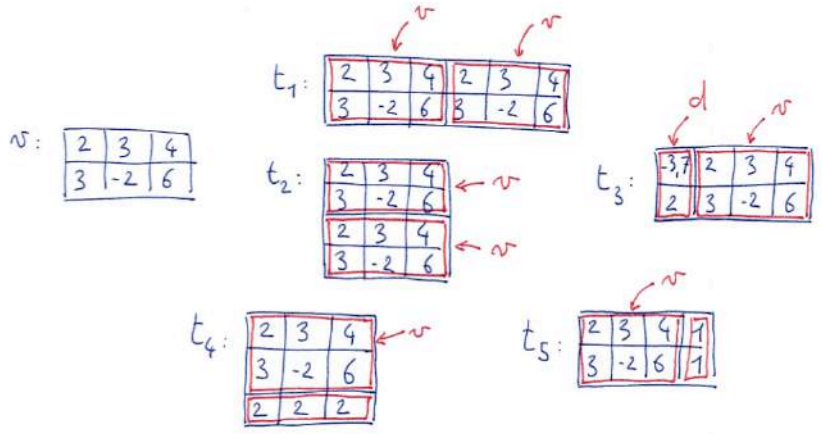



Je peux construire directement un tableau en concaténant des nombres

```
v=[2, 3, 4; 3, -2, 6];
```

j'ai ainsi dans la variable `v` un tableau à deux lignes et trois colonnes. Je peux faire beaucoup de manipulations de la même sorte, il suffit de penser à des blocs que l'on mettrait les uns contre les autres. La seule contrainte est que le bloc résultant doit nécessairement être un rectangle.

```
t1=[v,v]
t2=[v;v]
t3=[d,v]
t4=[v; 2, 2, 2]
t5=[v,[1;1]]
```




J'ai fait successivement :

1. Mis dans `t1` un tableau construit en mettant deux `v` "côte à côte".
2. Mis dans `t2` un tableau construit en mettant deux `v` "l'un au dessus de l'autre".
3. Mis côte à côte `d` et `v`, c'est possible parce qu'ils ont le même nombre de lignes.

4. mis dans `t4 v`, avec "en dessous" une troisième ligne avec des 2 comme éléments.
5. Mis dans `t5`, `v`, avec à droite une cinquième colonne avec des 1 comme éléments.

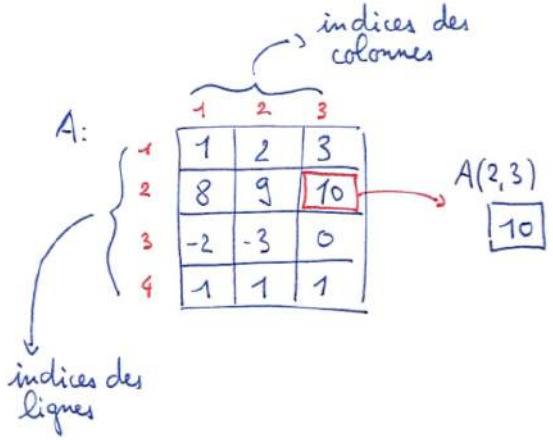
### 1.3.2 Accéder aux sous-tableaux

Dans la section précédente, nous avons vu comment créer des tableaux en concaténant des tableaux plus petits. Ici nous allons créer des tableaux plus petits en sélectionnant des sous-tableaux, ou bien changer directement les valeurs dans les sous-tableaux.

Supposons que nous avons le tableau

```
A=[1, 2, 3; 8, 9, 10; -2, -3, 0; 1, 1, 1];
```

C'est un tableau à 4 lignes et 3 colonnes. Le sous-tableau le plus simple, c'est un élément scalaire. On y accède par son indice de ligne et de colonne, par exemple `A(2,3)` est un tableau à une ligne et une colonne qui contient la valeur 10. Ici le premier indice est l'indice des lignes, et le second indice est l'indice des colonnes :




Je peux faire

```
b=A(2,3)+A(1,1)
```

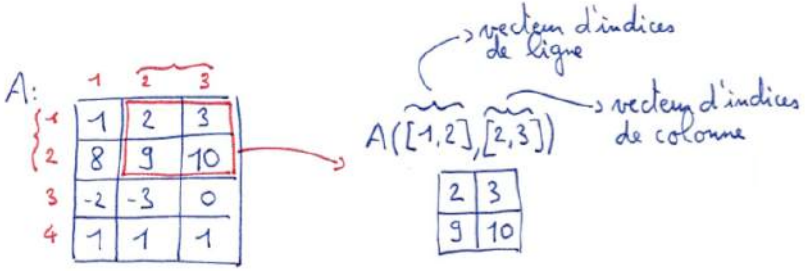
ici je mets dans une nouvelle variable b la somme des éléments (2,3) et (1,1) de `A`.

On peut de la même manière accéder aux sous-tableaux de `A`, mais en mettant maintenant des vecteurs d'indice :

```
c=A([1,2],[2,3])
```



maintenant, `c` est un tableau à deux lignes et deux colonnes :




Je peux mettre les vecteurs d'indice dans des variables, plutôt que de les écrire explicitement si ça m'arrange :

```
vl=[1,2];
vc=[2,3];
c=A(vl,vc);
```

est une séquence équivalente à la séquence précédente, ce sera parfois très pratique, par exemple lorsqu'on veut extraire le même sous-tableau de plusieurs matrices.

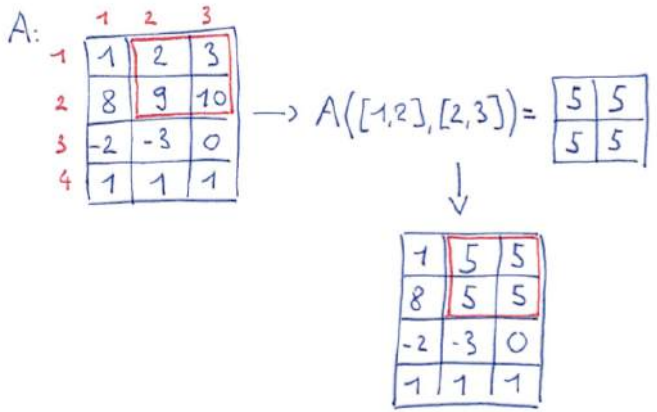
Une chose très intéressante, c'est que l'on peut changer la valeurs des sous-tableaux. Tout d'abord on change un seul élément :

```
A(2,3)=2;
```

Ici on ne change que l'élément de la deuxième ligne, troisième colonne. On peut aussi changer directement une plus grande sous-matrice :

```
A([1,2],[2,3])=[5, 5; 5, 5]
```

Ici, on remplace le bloc spécifié par les vecteurs d'indice par un bloc de 5. Pour que ça marche, il faut que le bloc qui reçoit (à gauche du `=`), ait la même taille que le bloc qu'on donne (à droite du `=`) :



### 1.3.3 Opérations avec des tableaux

Toutes les manipulations arithmétiques que nous avons l'habitude de pratiquer sur des scalaires peuvent facilement être étendues pour agir sur les tableaux, simplement en appliquant ces opérations *élément par élément*. Il faudra seulement faire attention avec les multiplications, puisqu'il existe une règle de manipulation très utile, la multiplication de *matrices* qui n'est pas une opération élément par élément.

Supposons que nous avons deux tableaux

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

alors

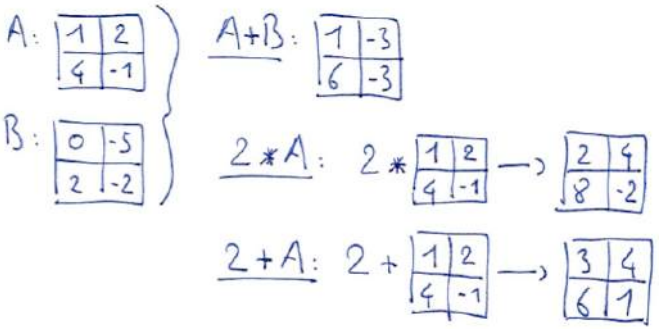
```
C=A+B
```

consiste à mettre dans l'élément `C(i,j)` la somme des éléments `A(i,j)` et `B(i,j)`. Par exemple, `C(2,2)` est égal à -3.

Les deux instructions suivantes

```
D=2*A
E=2+A
```

consistent à mettre dans `D` les éléments de `A` multipliés par deux, et de mettre dans `E` les éléments de `A` auxquels on ajoute 2.




La multiplication des tableau est aussi une multiplication élément par élément, et se note `.*` et non `*`. L'instruction

```
C=A.*B
```

met dans `A(i,j)*B(i,j)` dans `C(i,j)`. Elle est équivalente à la suite d'instructions suivante :

```
C=[0, 0; 0, 0];
for i=[1,2]
  for j=[1,2]
    C(i,j)=A(i,j)*B(i,j)
  end
end
```



A ne pas confondre avec la multiplication de matrices, notée `*` :

```
C=A*B
```

qui consiste à considérer `A` et `B` comme des matrices et non comme des tableaux. Cette instruction est équivalente à :

```
C=[0, 0; 0, 0];

for i=1:2
  for j=1:2
    for k=1:2
      C(i,j)=C(i,j)+A(i,k)*B(k,j);
    end
  end
end
```

On verra que matlab sera très utile pour toutes les opérations qui prennent en compte des matrices, avec l'aide des propriétés de l'algèbre linéaire, pour résoudre des systèmes d'équations, calculer des vecteurs propres et des valeurs propres... D'ailleurs, matlab signifie "MATrix LABoratory".

On peut aussi utiliser les fonctions mathématiques sur les tableaux, en appliquant la fonction élément par élément, par exemple :

```
C=sin(A)
D=exp(B)
E=cos(A)+tanh(B)
```

Ceci fait gagner des lignes de codes (donc du temps et des soucis), puisqu'il suffit d'écrire une seule instruction pour appliquer la fonction à tous les éléments. Par exemple la troisième instruction ci-dessus est équivalente à

```
E=[0, 0; 0, 0];

for i=[1,2]
  for j=[1,2]
    E(i,j)=cos(A(i,j))+tanh(B(i,j))
  end
end
```

Sans cette capacité de matlab de traiter les tableaux élément par élément, on a vu dans les exemples précédents, qu'il faut utiliser des boucles `for` imbriquées qui parcourent tous les indices. On voit bien que ce type de codage alourdi considérablement le code.

### 1.3.4 Fonctions spéciales pour les tableaux

Après avoir vu des fonctions qui s'appliquent élément par élément, et qui agissent donc sur des tableaux de la même manière que sur des scalaire, nous voyons des fonctions qui servent à créer et manipuler des tableaux.

#### pour créer des tableaux

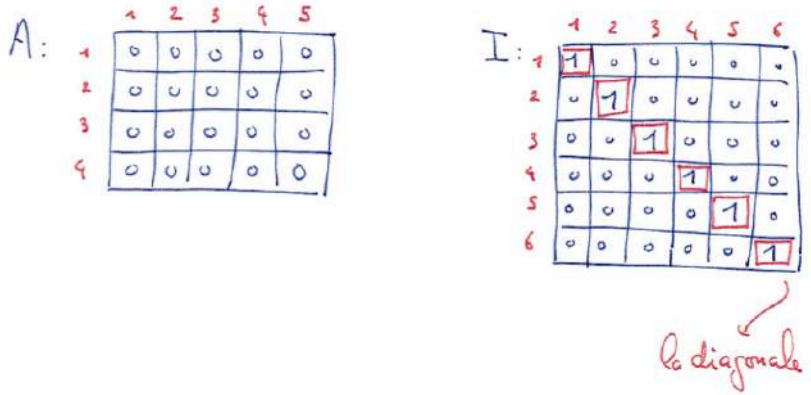
Pou créer un tableau rempli de zéros on utilise la fonction `zeros`

```
A=zeros(4,5)
```

ici, `A` devient un tableau à quatre lignes et cinq colonnes, remplis de zeros. On peut aisément deviner à quoi sert la fonction `ones` qui fonctionne de la même manière. Il y a la fonction `eye`

```
I=eye(6)
```

qui construit une matrice unitaire, c'est à dire remplie de zeros, à part les éléments diagonaux qui sont des 1. La matrice unitaire est souvent notée `I`. Le nom "eye" provient de l'anglais : "eye-dentity".




Une fonction qui sera utile pour faire varier les paramètres, la fonction `linspace`, qui crée des vecteurs-ligne (un tableau à une ligne et n colonnes)

```
v=linspace(0,1,65)
```

ici v est un vecteur de 65 éléments équidistants entre 0 et 1. Cette commande est équivalente à

```
v=0:1/64:1
```

qui est encore équivalent à la séquence explicite

```
v=zeros(1,65);
for i=1:65
  v(i)=(i-1)/64
end
```



#### Pour extraire de l'information des tableaux

Pour un tableau `A`, la fonction `max` donne en sortie un tableau ligne qui contient l'élément le plus grand de chaque colonne de `A`

```
>> A=[2, 3, 4; 1, 4, 4]
>> v=max(A)
v= 2 4 4
```

pour avoir l'élément le plus grand de `A`, sans le détail colonne par colonne, il suffit de faire

```
m=max(max(A))
```

on peut aussi obtenir l'indice auquel l'élément le plus grand se trouve dans le tableau, pour cela, voir dans l'aide matlab : `help max`. La fonction `min` fonctionne de manière similaire. On retrouve souvent dans matlab cette propriété de fonctions agissant colonne par colonne.

On peut calculer la somme ou le produit des éléments d'un tableau

```
v=sum(A)
r=prod(A)
```

ici aussi, la dimension de l'argument de sortie est égale à la dimension de l'argument d'entrée (ici `A`) moins 1, la fonction agissant colonne par colonne ; donc pour la somme de tous les éléments du tableau, faire `sum(sum(A))`.

## 1.4 Vectorisation

La vectorisation—en fait ici on devrait dire la *tableau-isation*—est une pratique de programmation qui consiste à éviter les manipulations élément par élément. Dans cette section, on verra des exemples et quelques principes directeurs. Vous verrez que nous avons déjà utilisé beaucoup de formulations vectorisées, qui sont très naturelles en matlab.

Ici une série d'exemples. Création d'un vecteur de zeros. élément par élément :

```
x=[];
for i=1:20
  x=[x, 0];
end
```

vectorisé :

```
x=zeros(1,20);
```

en utilisant une fonction prédéfinie `zeros`. Construction d'un vecteur d'éléments répartis linéairement entre 0 et $2\pi$, élément par élément

```
for i=1:20
  x(i)=2*pi*(i-1)*1/19;
end
```

(il faut bien faire attention à ne pas se tromper...) vectorisé :

```
x=linspace(0,2*pi,20);
```

en utilisant la fonction prédéfinie `linspace`. Calcul du sinus de ces valeurs, élément par élément :

```
for i=1:20
  f(i)=sin(x(i));
end
```

vectorisé :

```
f=sin(x)
```

graph de ce vecteur du sinus de $x$, élément par élément :

```
for i=1:19
  line([x(i),x(i+1)],[f(i),f(i+1)]);
  hold on
end
hold off
```

ou nous avons tracé un à un tous les segments reliant les points (abscisse,ordonnée) consécutifs. Vectorisé :

```
plot(x,f)
```

en utilisant la fonction prédéfinie `plot`. Calcul de la valeur maximale dans le vecteur `f`, en indiquant à quel indice cette valeur ce trouve dans `f` ; élément par élément :

```
maxval=-inf;
indloc=0;
for i=1:20
  if f(i)>maxval;
    maxval=f(i);
    indloc=i;
  end
end
```

Ici, `inf` c'est la valeur infinie qui est plus grande que toutes les autres. Vectorisé :

```
[maxval,indloc]=max(f);
```

On pourrait continuer cette liste pendant longtemps, par exemple, pensez comment coder élément par élément ce que fait la fonction `sort`.

Jusqu'ici, pour vectoriser il a fallu connaître beaucoup de fonctions de matlab. Matlab est un langage dit "de haut niveau", non pas parce qu'il faut être très fort pour pouvoir l'utiliser, mais parce qu'il permet d'éviter les manipulations élémentaires, dites de "bas-niveau", ces manipulations élémentaires étant



déjà codées de manière sophistiquée et efficace dans une très large librairie de fonctions. Maintenant quelques exemples plus subtils utilisant les manipulations de tableaux.

Compter le nombre d'éléments égaux à $\pi$ dans un vecteur `v` donné. Elément par élément :

```
n=0;
for i=1:length(v)
  if v(i)==pi;
    n=n+1;
  end
end
```

dans cet exemple, `n` est une variable que l'on utiliser pour compter. Vectorisé

```
n=sum(v==pi);
```

ici, `v==pi` est un vecteur de la même taille que `v`, avec des zeros dans les cases ou `v` n'est pas égal à $\pi$ et des 1 dans les cases ou `v` est égal à $\pi$. La somme avec `sum` des éléments de ce vecteur est le nombre d'éléments de `v` égaux à $\pi$. Maintenant un peu plus subtil, on veut connaître les indices des éléments de `v` qui sont égaux à $\pi$. Élément par élément :

```
indlist=[]
for i=1:length(v)
  if v(i)==pi;
    indlist=[indlist, i];
  end
end
```

vectorisé :

```
k=1:length(v);
indlist=k(v==pi);
```

dans cet exemple, on a commencé par construire un vecteur auxiliaire—une aide—`k` comme le vecteur de tous les indices : $(1, 2, 3, ...)$, et on n'a sélectionné dans ce vecteur que les cases telles que la case correspondante dans `v` contient $\pi$. En découpant tout cela en petites étapes pour bien voir ce qui se passe, au prompt de matlab :

```
>> v=[0,2,4,pi,0,pi]
v =
         0    2.0000    4.0000    3.1416         0    3.1416
>> v==pi
ans =
     0     0     0     1     0     1
>> v(v==pi)
ans =
```

```
    3.1416    3.1416
>> k=1:length(v)
k =
     1     2     3     4     5     6
>> k(v==pi)
ans =
     4     6
```

J'ai d'abord construit mon vecteur `v`, en mettant $\pi$ en quatrième et sixième position. Je regarde à quoi ressemble le vecteur `v==pi`, c'est un vecteur binaire avec des zéros partout sauf en position quatre et six... je teste `v(v==pi)`, c'est à dire, j'évalue quels éléments de `v` sont égaux à $\pi$, le résultat est rassurant... Je construit ensuite mon vecteur auxiliaire `k`, et je regarde quels sont les éléments de `k` qui correspondent aux cases ou `v` est égal à $\pi$, et je trouve bien le résultat escompté.

En fait pour ce problème là, on aurait pu utiliser la fonction `find` qui donne directement les indices à partir du tableau de test `v==p`

```
>> find(v==pi)
ans =
     4     6
```

La vectorisation est avantageuse pour plusieurs raisons :

- Eviter les boucles `for`, cela économise des lignes de codes, économise des indices à faire varier. C'est une économie de temps de codage.
- En matlab, c'est beaucoup plus rapide en temps de calcul : la plupart des fonctions de matlab et des opérations de tableaux (addition, multiplication...) sont compilées et optimisées pour une efficacité maximale ; lorsque l'on fait boucles et petites opérations, ces instructions sont interprétées une-à-une, ce qui est lent et doit donc être évité autant que faire se peut.

## 1.5 Graphiques

Les fonctions principales sont les fonctions `plot` pour les lignes, et `mesh` pour les surfaces. Nous commencerons par voir comment ces fonctions se comportent, et nous verrons ensuite comment tracer des isolignes avec `contour`, des surfaces en couleurs avec `surf`, ou encore des champs de vitesses avec `quiver`. Dans ce cours, nous insistons sur le fait que *toute donnée peut être visualisable, et doit être visualisée.*

### 1.5.1 Lignes

Nous allons visualiser la fonction

$$f(x) = sin(x)e^{\frac{-x^2}{10}}$$



elle est composée d'une fonction sinus, qui ondule, et d'un facteur en cloche Gaussienne en $e^{-x^2}$ qui joue ici le rôle d'enveloppe", elle force l'amplitude du sinus à décroître vers zeros lorsque $|x|$ devient grand.

On commence par définir le vecteur des $x$ et calculer la valeur de $f$ pour tous les points en $x$

```
n=200
x=linspace(-10,10,n);
f=sin(x).*exp(-x.^2/10);
```

avec $n$ le nombre de points. Lorsque je tape la commande

```
plot(f)
```

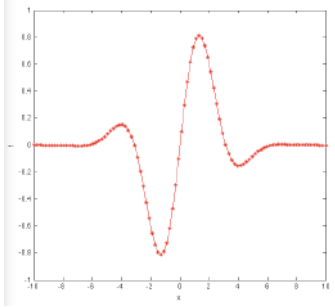
je n'ai pas précisé quels sont les points d'abscisse (les $x$), je n'ai donné que les ordonnées. Dans ce cas matlab suppose que le vecteur des abscisses est $(1, 2, 3, 4...)$, jusqu'à $n$. Pour tracer correctement la fonction, il faut entrer

```
plot(x,f)
```

Je peux choisir la couleur et le style de ligne

```
plot(x,f,'r*--')
```

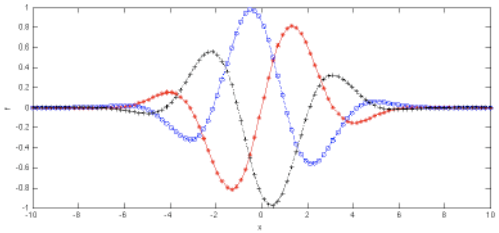
ici, je demande a ce que le graph soit en rouge (`r`), avec des astérisques aux points $x(i)$ (`*`), et une ligne en pointillés (`--`).



Quelques couleurs : "b" pour bleu, "k" pour noir, "m" pour magenta, "c" pour cyan... et pour les lignes, "-" pour ligne continue, "–" pour pointillés, "-." pour une ligne pointillée mixte... Pour plus d'informations, appelez à l'aide : tapez `help plot`.

Je peux aussi superposer plusieurs courbes, par exemple en mettant plusieurs couples abscisses/ordonnées à la suite

```
f1=sin(x).*exp(-x.^2/10);
f2=sin(x+2*pi/3).*exp(-x.^2/10);
f3=sin(x+4*pi/3).*exp(-x.^2/10);
plot(x,f1,'r*-',x,f2,'bo--',x,f3,'k+-.')
```



Ici, j'ai mis trois fonctions obtenues en déphasant de $2\pi/3$ et $4\pi/3$ les oscillations du sinus, on commence à voir apparaître l'effet de l'enveloppe en exponentielle.

On peut également utiliser les commandes `hold on` et `hold off` avec la séquence suivante

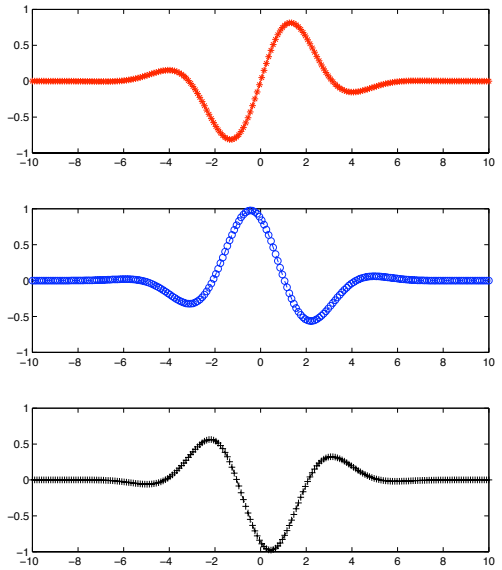
```
plot(x,f1,'r*-');
hold on
plot(x,f2,'bo--');
plot(x,f3,'k+-.');
hold off
```

qui est une séquence de commandes équivalente à la commande précédente. Ce sera particulièrement utile lorsque nous ferons des graphes dans des boucles `for`. Le "hold on" consiste à ce que la commande graphique suivante n'efface pas les graphiques précédents.

Si l'on veut voir plusieurs courbes à la fois, plutôt que de les superposer, on peut les mettre dans plusieurs sous-fenêtres. Pour cela on utilise la fonction `subplot`

```
subplot(3,1,1); plot(x,f1,'r*-');
subplot(3,1,2); plot(x,f2,'bo--');
subplot(3,1,3); plot(x,f3,'k+-.');
```





`subplot(nl,nc,n)` découpe la fenêtre graphique en $nl$ lignes et $nc$ colonnes, et trace les commandes graphiques suivantes dans la sous-fenêtre $n$, en comptant de gauche à droite et de haut en bas. Dans l'exemple précédent, il y a trois colonnes et une seule ligne.

Si l'on veut visualiser nos données sous la forme d'une animation, il est commode d'utiliser une boucle `for`

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-x.^2/10),'r*-')
  ylim([-1,1])
  drawnow
end
```

On a utilisé la commande `drawnow`, comme son nom l'indique afin de forcer matlab à tracer *pendant* la boucle et non *à la fin*. Par défaut, le traitement des graphiques est *asynchrone*, c'est à dire que tous les calculs sont fait, et ensuite matlab s'occupe des graphiques. Pour une animation ce n'est bien sûr pas le bon choix.

Dans cet exemple, on déphase continûment le sinus en ajoutant $t$ à $x$. Cela donne l'effet d'une onde qui se déplace vers la gauche. On voit maintenant clairement l'effet de l'enveloppe qui ici est fixe. On peut aussi faire que cette enveloppe se déplace vers la droite en rajoutant une dépendance temporelle dans l'argument de l'exponentielle :

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-(x-t).^2/10),'r*-')
  ylim([-1,1])
  drawnow
end
```

On peut en fait changer tout ce que l'on veut pendant l'animation, par example dans l'exemple suivant, je fais évoluer la couleur du blanc au noir en utilisant la propriété `'color'` : le code de couleur par défaut c'est le "rgb", c'est à dire "red-green-blue" : `[1,0,0]` c'est rouge pur, `[0,1,0]` c'est vert pur, et `[0,0,1]` c'est bleu pur. Accessoirement, `[1,1,1]` c'est noir, et `[0,0,0]` c'est blanc. Je fais aussi évoluer l'épaisseur de ligne, avec la propriété `'linewidth'`, ici de 0 à 20 :

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-(x).^2/10),'color',1-[1,1,1]*t/20, ...
              'linewidth',20*t/20+1)
  ylim([-1,1])
  drawnow
end
```

J'ai utilisé ici une fonctionnalité usuelle des fonctions matlab, qui consiste à mettre après les arguments habituels, des couples "propriété-valeur", ou le nom de la propriété est une chaîne de caractère, ici `'color'` pour la couleur de la ligne, et `'linewidth'` pour l'épaisseur de la ligne.

### 1.5.2 Surfaces

Maintenant, nous allons tracer des surfaces. Considérons la fonction $sin(x)e^{-y^2}$ qui évolue comme un sinus selon $x$ et comme une cloche Gaussienne selon $y$. Nous avons vu à la section précédente que pour tracer une fonction, il faut d'abord la "construire", c'est à dire créer un vecteur dans lequel on va mettre les valeurs de la fonction aux différents points de l'espace que nous allons utiliser pour le graphique.

Pour calculer la valeur de fonctions mathématiques en deux dimensions, on crée les points de discrétisation selon $x$ et $y$

```
n=20;
x=linspace(-5,5,n);
y=linspace(-4,4,n);
```

Et maintenant, on pourrait calculer pour chaque couple $x(i), y(j)$ la valeur de la fonction en faisant deux boucles `for`

```
f=zeros(n,n);
for i=1:n
  for j=1:n
    f(i,j)=sin(x(i))*exp(-y(j)^2);
```



```
  end
end
```

En fait, on peut utiliser une astuce pratique pour vectoriser cette opération, c'est à dire réaliser notre but sans boucles, en utilisant la fonction `meshgrid`

```
[X,Y]=meshgrid(x,y);
f=sin(X).*exp(-Y.^2);
```

Pour comprendre ce que fait la fonction `meshgrid`, méditer l'exemple suivant :

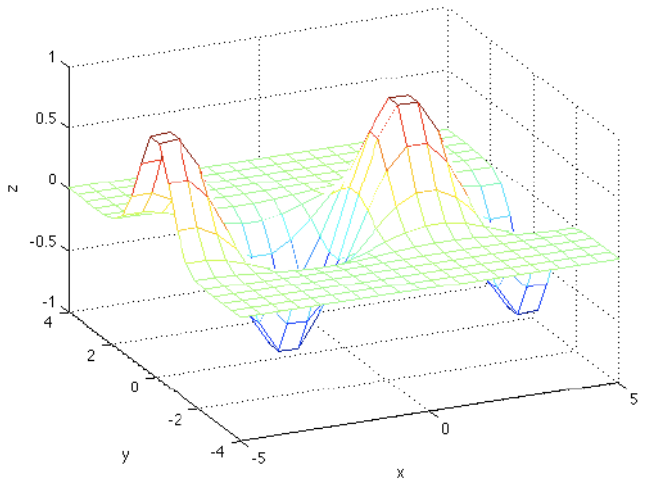
```
>> [X,Y]=meshgrid([1,2,3],[4,5,6])
X =
     1     2     3
     1     2     3
     1     2     3
Y =
     4     4     4
     5     5     5
     6     6     6
```

Ainsi, `sin(X)` est un tableau de la même taille que `X`, avec pour valeurs les sinus des valeurs de `X`. De manière similaire, `Y.^2` est un tableau de la même taille que `Y` avec pour valeurs le carré des valeurs de `Y`, et encore de manière similaire, `exp(-Y.^2)` est un tableau qui a comme éléments l'exponentielle du carré des éléments de `Y`. De cette manière, `sin(X).*exp(-Y.^2)` correspond bien à ce que nous voulons créer. L'utilisation de `meshgrid` est typique de la pratique en vectorisation : plutôt que de faire des boucles, on se débrouille pour construire des tableaux pratiques. Ca vaut le coup de prendre le temps qu'il faut pour bien comprendre le principe de `meshgrid`.

Une fois cette astuce décrite et notre fonction construite, nous pouvons tracer le graph $f$

```
mesh(X,Y,f);
xlabel('x');  ylabel('y'); zlabel('z');
title('sin(x)exp(-y^2)');
```

C'est la fonction `mesh` que nous avons utilisée ; en anglais, "mesh" signifie "filet" ou "grillage". Si on veux changer les limites des axes $x$, $y$, $z$, on peut utiliser les commandes `xlim, ylim, zlim` comme nous l'avons fait plus tôt.




Nous allons maintenant réaliser une petite animation, comme précédemment, en ajoutant à notre fonction une dépendance en temps

```
for t=linspace(0,2*pi,30)
  f=sin(X+t).*exp(-Y.^2);
  mesh(X,Y,f)
  drawnow
end
```

On peut changer tout ce que l'on veut lors de cette boucle d'animation, par example on peut faire orbiter la caméra

```
for t=linspace(0,2*pi,30)
  mesh(X,Y,sin(X+t).*exp(-Y.^2));
  camorbit(380*t/(2*pi),0);
  drawnow
end
```

Ici la fonction `camorbit` fait tourner la caméra d'un angle donné, en degrés, par rapport à la position par défaut. Ici nous faisons faire à la caméra un tour complet (de 0 à 380 degrés) lorsque $t$ va de 0 à $2\pi$.

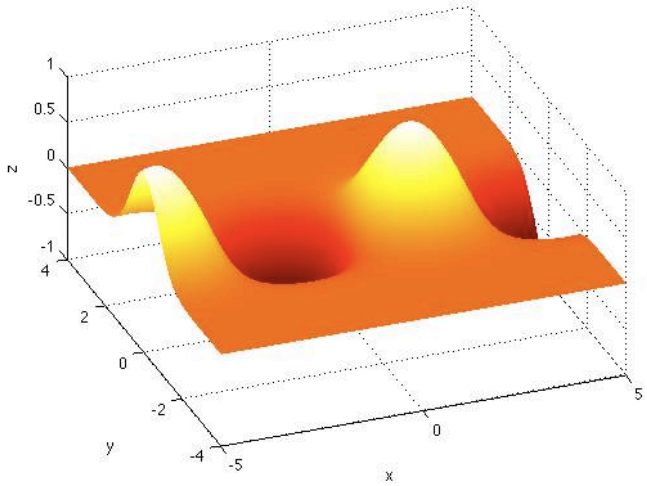
Si l'on veut une belle surface plutôt qu'un grillage, on peut utiliser la fonction `surf`

```
surf(X,Y,sin(X+t).*exp(-Y.^2));
shading interp;
colormap(jet(400))
```

Ici, j'ai utilisé la fonction `shading interp`, qui interpole les couleurs entre les points de la surface, et j'ai changé la "carte des couleurs", avec la fonction



`colormap`, en utilisant la palette "jet" et en utilisant 400 teintes, plutôt que les 64 par défaut, pour avoir une belle continuité de couleurs (j'ai mis 100 points en $x$ et en $y$).

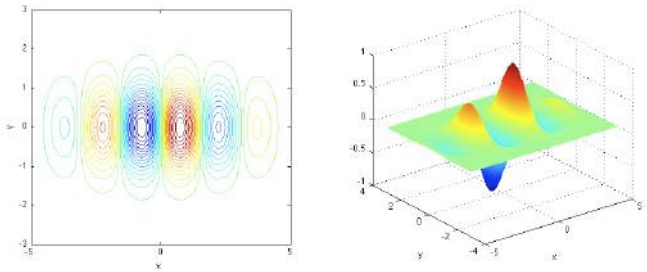



### 1.5.3 Isovaleurs

Il est parfois utile de tracer des isovaleurs. Par example les iso-valeurs de la fonction de courant en mécanique des fluides sont les lignes de courant : les lignes en tous points tangentes au vecteur vitesse des particules fluides. Reprenons notre fonction

```
n=200;
x=linspace(-5,5,n);
y=linspace(-3,3,n);
[X,Y]=meshgrid(x,y);
f=sin(2*X).*exp(-Y.^2).*exp(-X.^2/8);

subplot(1,2,1)
contour(X,Y,f,30)
xlabel('x'); ylabel('y')

subplot(1,2,2)
surf(X,Y,f); shading interp
xlabel('x'); ylabel('y')
```

Ici, dans la sous-fenêtre de gauche nous avons 30 iso-contours de la fonction $f$ linéairement répartis (par défaut) entre la plus grande valeur et la plus petite valeur de $f$, et à droite le graph en `surf` de la même fonction.

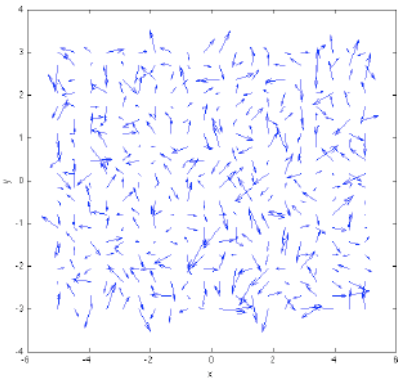


### 1.5.4 Champs de vecteurs

La fonction `quiver` a pour arguments le tableau des points en $x$, le tableau des points en $y$, et les coordonnées en $x$ et en $y$ des vecteurs à y mettre. Prenons pour faire simple un champ de vecteurs aléatoire

```
n=20;
x=linspace(-5,5,n);
y=linspace(-3,3,n);
[X,Y]=meshgrid(x,y);
u=randn(n,n);
v=randn(n,n);
quiver(X,Y,u,v);
```

Ici, la fonction `randn` crée en sortie un tableau de taille $(n, n)$, dont les éléments sont choisis de manière aléatoire de telle sorte que leur valeur moyenne est nulle et que leur variance est égale à 1.



On peut faire une petite animation

```
for t=linspace(0,10,30)
  quiver(X,Y,u+t,v);
  drawnow
end
```



ici a chaque pas de temps, on ajoute $t$ à la composante horizontale du champs de vecteurs et on trace : l'ordre apparaît dans le chaos... La fonction `quiver` normalise par défaut la longueur des vecteurs de sorte à ce que les vecteurs ne se chevauchent pas. Pour éviter cette renormalisation et voir la vraie taille de vecteurs, écrire `quiver(X,Y,u+t,v,0)`.

## 1.6 Compétences techniques

Dans cette section, je présente une liste des activités de base pour l'utilisation que nous allons faire de matlab dans notre UE. Ce sont les savoir-faire de base qui sont le coeur de cet enseignement. Ce sont des blocs pratiques qu'il faut que vous sachiez faire. L'examen nous sert à vérifier que vous avez bien acquis ces compétences.

### 1.6.1 Calcul et convergence d'une série

Il s'agit ici de calculer la constante d'Euler, c'est à dire la valeur du "e" de l'exponentielle. Cette valeur est définie ici par la limite d'une série. Formule :

$$e^x = 1 + x + x/2 + x/3! + \cdots + x^k/k! + \ldots$$

Ou le signe " !" représente l'opération de "factoriel". Matlab met à notre disposition la fonction "factorial" qui prend en argument d'entrée un scalaire et rend en argument de sortie son factoriel.

On va calculer cette série avec une boucle `for`. Voici pour commencer un script très simple :

```
% calcul de e par une serie
e=1; % initialisation de la somme

for ind=1:10
    e=e+1/factorial(ind);
    disp(e)
end
```

Ici on a juste accumulé dans la variable "e" les termes successifs de la série, et on affiche à l'écran la nouvelle valeur à chaque itération de la boucle "for".

Pour étudier la convergence de cette série, on peut comparer la valeur de notre somme à la valeur la plus précise dont nous disposons, c'est à dire `exp(1)`. Dans le script ci-dessous, j'affiche à l'écran la valeur de l'erreur entre la somme partielle de ma série à chaque itération et la valeur précise donnée par matlab de l'exponentielle de 1 :

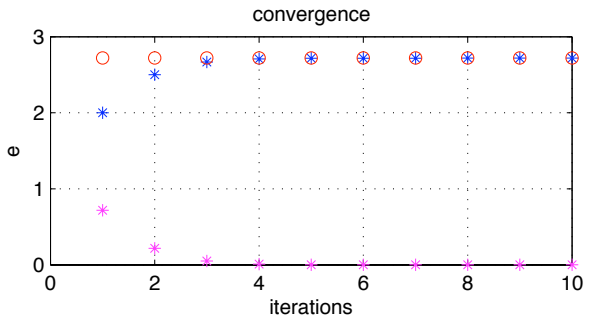
```
% calcul de e par une serie
e=1; % initialisation de la somme
eexact=exp(1); % la valeur donnée par Matlab
```

```
for ind=1:10
    e=e+1/factorial(ind);
    disp(e-exp(1)) % on affiche l'erreur à l'ecran
end
```

Ce n'est pas très pratique de lire des chiffres à l'écran pour évaluer la convergence de la série, cela est bien plus pratique visuellement en traçant des graphiques. C'est ce que fait d'une manière très élémentaire le script suivant :

```
% calcul de e par une serie
e=1; % initialisation de e
eexact=exp(1); % la valeur donnée par Matlab

for ind=1:10
    e=e+1/factorial(ind);

    % affichage
    plot(ind,e,'b*',ind,eexact,'ro',ind,abs(e-eexact),'m*'); hold on
    xlabel('iterations');
    ylabel('e');
    title('convergence');

end
grid on
```

Et voici la figure :




Ici, j'ai tracé un graphique à chaque itération. Comme j'ai utilisé la fonction `hold on`, le graphique nouveau n'efface pas le graphique ancien. Le graphique garde ainsi la mémoire de l'évolution des itérations. J'ai tracé en abscisse la valeur de l'indice, et en ordonnée d'une part la valeur actuelle de la somme partielle "e" en bleu avec une astérisque, la valeur précise en rouge avec un cercle, et d'autre part la valeur de l'erreur entre la somme partielle et la valeur précise. Bien sûr, il faut toujours mettre des labels et un titre sur son graphique.

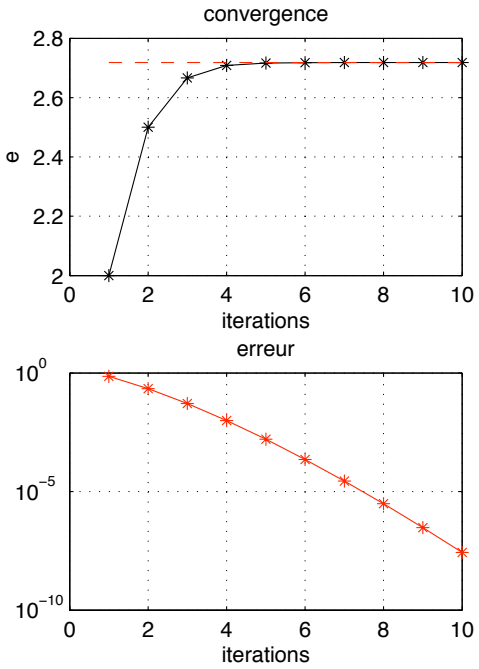
Ce script est pas mal, parce que je peux voir comment la série converge (ou bien si j'ai fait une erreur de programmation, je peux voir que en fait elle ne converge pas). Par contre, une fois le calcul terminé, j'ai perdu toutes les



valeurs qui sont témoin de la convergence. Je peux avoir besoin de les mémoriser pour en faire le traitement après le calcul. De plus, avec cette manière de tracer l'évolution, je ne peux pas lier les points successifs du graphique avec un segment, ce qui est dommage parce que lier les points successifs ça facilite la lecture. Voici un script qui fait d'abord tous les calculs, qui mémorise ce dont on a besoin, puis qui trace les graphiques à la fin :

```
% calcul de e par une serie
e=1; % initialisation de e
n=10; % nombre d'itérations
emem=zeros(1,n); % pour mémoriser l'évolution de e

for ind=1:n
    e=e+1/factorial(ind);

    % memorise la valeur de e a cette iteration
    emem(ind)=e;
end

% graph de l'évolution de la somme partielle
subplot(2,1,1)
plot(1:n,emem,'k-*'); % evolution de la somme partielle
hold on
plot(1:n,emem*0+exp(1),'r--'); % valeur de référence
xlabel('iterations');
ylabel('e');
title('convergence');
grid on

% graph de l'évolution de l'erreur
subplot(2,1,2);
semilogy(1:n,abs(emem-exp(1)),'r-*');
xlabel('iterations');
ylabel('erreur');
title('erreur');
grid on
```

Et la figure associée :




J'ai juste créé un tableau rempli de zéros qui a la bonne taille (un tableau ligne qui a autant de cases que d'itérations dans ma boucle), et à chacune de mes itérations, j'ai mis la valeur de la somme partielle dans la bonne case, la case numéro "ind". Ensuite j'ai tracé l'évolution de la somme partielle et l'évolution de l'erreur dans deux sous-graphiques en utilisant la fonction subplot. Pour tracer l'évolution de l'erreur, j'ai utilisé un graphique semi-logarithmique, c'est à dire pour lequel l'axe des y est gradué de manière logarithmique. Cela me permet de continuer à bien voir l'évolution de l'erreur même lorsqu'elle est très petite. On voit ici que l'erreur de ma somme partielle est aussi petite que $10^-7$, c'est à dire 0.0000001 après seulement dix itérations. C'est une convergence rapide.

Notez que cet exemple, j'ai utilisé une astuce pour tracer la valeur de référence : `plot(1:n,emem*0+exp(1),'r--');`. Je veux créer rapidement (en une seule commande) un tableau ligne de taille n qui contient dans chaque case la valeur de `eexact`. Pour cela je multiplie `emem` par zéro. C'est ainsi un tableau rempli de zéros qui a la bonne taille. Et ensuite j'ajoute à ce tableau la valeur scalaire de `eexact`, j'ai ainsi ajouté à chacune des cases la valeur que je voulais.



### 1.6.2 Prises de mesures sur une image

C'est quelque chose que nous allons souvent faire dans les TPs : une expérience physique a été capturée par une photographie, et nous allons mesurer des quantités pour les analyser ensuite. Nous pouvons afficher une image dans la fenêtre graphique de Matlab. Cette image est en fait un tableau d'un grand nombre de pixels : chaque point de l'image est repéré par les coordonnées du pixel correspondant. C'est ainsi qu'un appareil photo peut être un instrument de mesure extrêmement précis : d'autant plus précis que son capteur dispose d'un grand nombre de pixels.

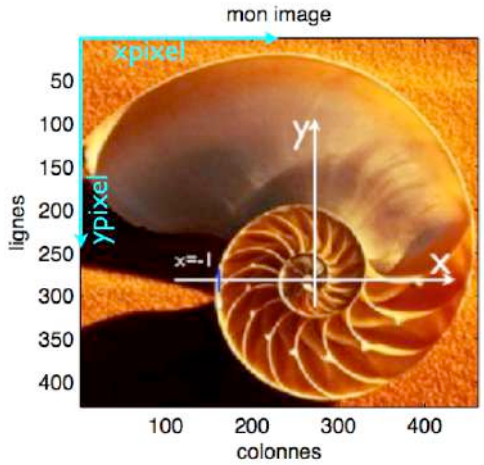
Dans cette section, nous allons étudier la forme d'un coquillage. On montrera par la suite que ce coquillage ressemble beaucoup à une spirale logarithmique.

L'image est stockée sur le disque dur dans le fichier "nautile.png". Je met ce fichier dans le répertoire courant de Matlab de sorte à pouvoir y accéder depuis la ligne de commande. Le contenu de l'image est transféré dans le tableau "a" avec la fonction "imread" :

```
clear all; clf

% on lit l'image et on l'affiche
a=imread('nautile.png');
image(a); axis equal tight
xlabel('colonnes')
ylabel('lignes')
title('mon image')
```

Le nom du fichier est donné en argument d'entrée sous la forme d'une chaîne de caractères, et l'argument de sortie est "a". Je pense bien à mettre un point virgule à la fin de cette commande parce que il y a un très grand nombre de pixels dans l'image, donc le tableau "a" est très grand : cela prendrait beaucoup de temps à afficher toutes les valeurs à l'écran.

J'ai ensuite affiché cette image dans une fenêtre graphique à l'aide de la fonction `image` qui prend en argument d'entrée la tableau "a" qui contient les données de l'image. Bien sûr j'ai ajouté des labels et un titre à ma figure. Voici la figure que l'on obtient :




On note que les axes sont numérotés en pixels : il y a à peu près 500 pixels dans la direction horizontale et 430 pixels dans la direction verticale. L'origine de la numérotation est en haut à gauche de l'image, et on note de plus que l'axe vertical est orienté de haut en bas. En termes de référentiel, cela signifie que le référentiel de l'image a son centre en haut à gauche de l'image, que l'échelle de longueur est le pixel et que l'axe des y est à l'envers par rapport au graphique cartésien habituel en mathématiques. Ce référentiel est représenté sur l'image en bleu clair et les axes sont nommés "xpixel" et "ypixel".

Nous allons maintenant mesurer la forme de ce coquillage en prenant une vingtaine de points de mesures le long de la ligne spirale. Pour cela nous pouvons par exemple utiliser l'outil d'étiquetage de la fenêtre Matlab :

En ayant sélectionné cet outils dans la barre de menu de la fenêtre graphique, il suffit ensuite de cliquer sur l'image pour avoir les coordonnées du pixel sur lequel on a cliqué, ainsi que le triplet RGB de la couleur de ce pixel. Le code RGB décrit la couleur en spécifiant la proportion de rouge (R pour "red") de vert (G pour "green") et bleu (B pour "blue").

Pour mesurer la position d'une vingtaine de points sur la coquille, on pourrait cliquer successivement sur 20 points et noter les coordonnées, mais nous avons pour cela un outil plus pratique, la fonction `ginput`.

La fonction ginput est une fonction interactive dont le nom est un raccourci pour "graphical input", (entrée graphique). J'insiste sur le fait que c'est une fonction interactive : on ne la met pas dans un script, mais on la tape à l'invite de la fenêtre de commande. En effet, lors du développement de notre projet, on est amené à exécuter de nombreuses fois le script en cours de codage : pour l'améliorer, pour éliminer les erreurs, pour tester de nouvelles idées. Par contre, il suffit de faire la prise de mesure une bonne fois pour toutes, et de sauver les données quelque part. A chaque fois que l'on appelle la fonction "ginput", Matlab attend que l'on clique pour la prise de mesures.



Par exemple pour notre coquillage :

```
[xpixel,ypixel]=ginput
```

Ici il n'y a pas d'argument d'entrée à ginput, et j'ai mis deux arguments de sortie : "xpixel" est le tableau des coordonnées x des points sur lesquels je vais cliquer, et "ypixel" est le tableau des coordonnées y :

```
xpixel =

  266.9891
  281.6022
  283.6898
  249.2445
  233.5876
  261.7701
  313.9599
  330.6606
  303.5219
  244.0255
  181.3978
  161.5657
  179.3102
  239.8504
  391.2007
  453.8285
  392.2445
  242.9818
  112.5073
    8.1277


ypixel =

  291.1971
  292.2409
  265.1022
  254.6642
  290.1533
  321.4672
  313.1168
  264.0584
  213.9562
  195.1679
  234.8321
  308.9416
  368.4380
  414.3650
  375.7445
  252.5766
   88.7007
   18.7664
   45.9051
  118.9708
```

Une fois que j'ai mesuré les coordonnées de ces points, je vais les sauver sur le disque dur avec la fonction "save" :

```
save('mesdata.dat','xpixel','ypixel');
```

J'ai sauvé les deux tableaux "xpixel" et "ypixel" dans un nouveau fichier nommé "mesdata.dat" dans le répertoire courant. Notez que le nom des tableaux à enregistrer est donné en argument d'entrée de la fonction "save" en tant que chaînes de caractères. Vérifiez bien que ce fichier à été créé en regardant la liste des fichiers de votre répertoire courant.

Ensuite, lorsque je vais avoir besoin de ces données, je vais les charger dans le workspace de Matlab avec la fonction "load" :

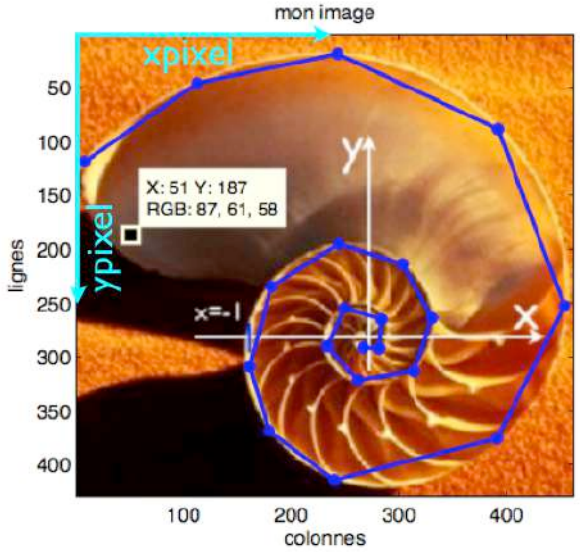
```
load('mesdata.dat');
```

et j'ai ainsi disponibles les tableaux "xpixel" et "ypixel" des coordonnées des points mesurés.

Pour bien vérifier que nous avons bien pris les points de mesure, nous allons tracer par dessus l'image les points de coordonnées "xpixel" et "ypixel" :

```
image(a);
hold on
plot(xpixel,ypixel,'r*-');
```

Voilà la figure que nous obtenons (on y voit aussi la mesure faite avec l'outil d'étiquetage) :




### 1.6.3 Changement de référentiel

Les coordonnées que nous avons mesurées sont exprimées dans le référentiel des pixels de l'image, qui n'est pas pratique pour la comparaison avec des formules mathématiques. Nous devons donc effectuer un changement de référentiel. Sur l'image du Nautile, nous avons représenté les axes d'un référentiel physique et une référence de longueur (la position du -1 sur l'axe des abscisses). C'est dans ce référentiel là que nous voulons manipuler nos données. Voici les opérations de ce changement de référentiel :

```
% le centre du référentiel
xpixel0=271;
ypixel0=281;
x=xpixel-xpixel0; y=ypixel-ypixel0;

% inversion de l'axe des y
y=-y;

% taille de pixel et remise à l'échelle
```



```
taillepix=1/(272-161);

x=x*taillepix;
y=y*taillepix;
```

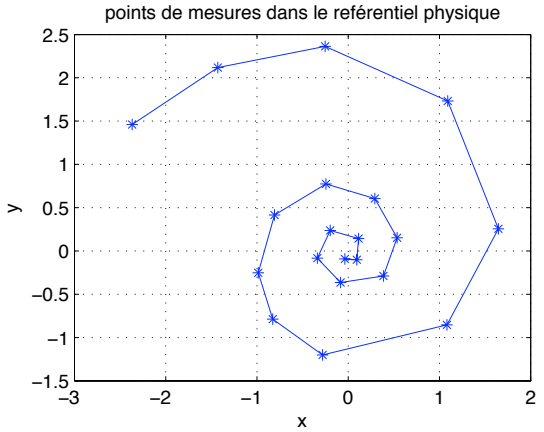
La première chose que nous avons faite est de mesurer sur la figure les coordonnées du centre du référentiel physique avec l'outil d'étiquetage (pour cela pas besoin de la fonction "ginput" puisqu'il n'y a qu'un seul point à mesurer). Les coordonnées du centre du référentiel physique sont stockées dans les variables "xpixel0" et "ypixel0". Ensuite nous avons soustrait ces coordonnées aux coordonnées des points mesurés. Cette soustraction c'est la translation de la courbe vers le centre du référentiel physique.

Ensuite nous avons inversé l'axe des "y". En effet l'axe vertical des pixels est gradué de haut en bas et non pas de bas en haut.

Ensuite nous avons mesuré la taille d'un pixel en unités de longueur physique. Pour cela nous disposons d'un étalon de longueur : la position "-1" indiquée sur l'image. Je mesure la taille en pixel de la distance entre ce point "-1" et le centre du référentiel, et cela me donne la taille physique d'un pixel. Cette taille est stockée ici dans ma variable "taillepix". La dernière opération à réaliser pour le changement de référentiel est maintenant de multiplier les coordonnées de mes points mesurés par ce "taillepix". Je peux maintenant tracer mes points de mesures dans le référentiel physique.

```
% on trace la spirale
plot(x,y,'b*-');
grid on
xlabel('x'); ylabel('y');
title('points de mesures dans le référentiel physique')
```

Et voici la figure obtenue :




Cette figure montre bien que nous avons effectué correctement le changement de référentiel : le point $(0, 0)$ est bien au centre de la spirale, preuve que nous

avons bien translaté les coordonnées, la courbe n'est pas retournée, preuve que nous avons bien inversé l'axe vertical, et l'échelle est bonne puisque la seconde spire de la spirale est de taille à peu près 1.

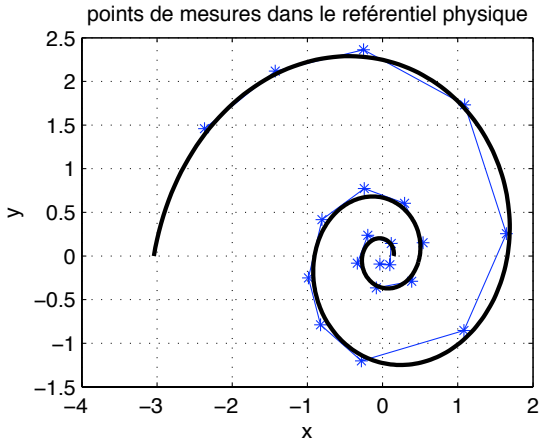
Maintenant je peux tracer par dessus mes points de mesure une spirale logarithmique pour comparer les formes. Pour ma spirale théorique, j'ai besoin d'identifier les deux paramètres "a" et "b" de la formule :

$$r = ab^{\theta}, \quad x = r\cos(\theta), \quad y = r\sin(\theta)$$

ou $\theta$ et $r$ sont les coordonnées polaires de ma courbe. Dans la section 1.6.5 plus bas, je montre comment estimer la valeur d'un paramètre d'une formule mathématique d'après une image.

```
% on trace la spirale
plot(x,y,'bo','linewidth',2)
xlabel('x'); ylabel('y');
title('mesures')

% on trace une spirale théorique
th=linspace(0,2*pi*2.5,500);
a=0.1471;
b=(1.66/a)^(1/(4*pi));
r=a*(b.^th);
xx=r.*cos(th); yy=r.*sin(th);
hold on;
plot(xx,yy,'k-','linewidth',2);
```

Et voici la figure :

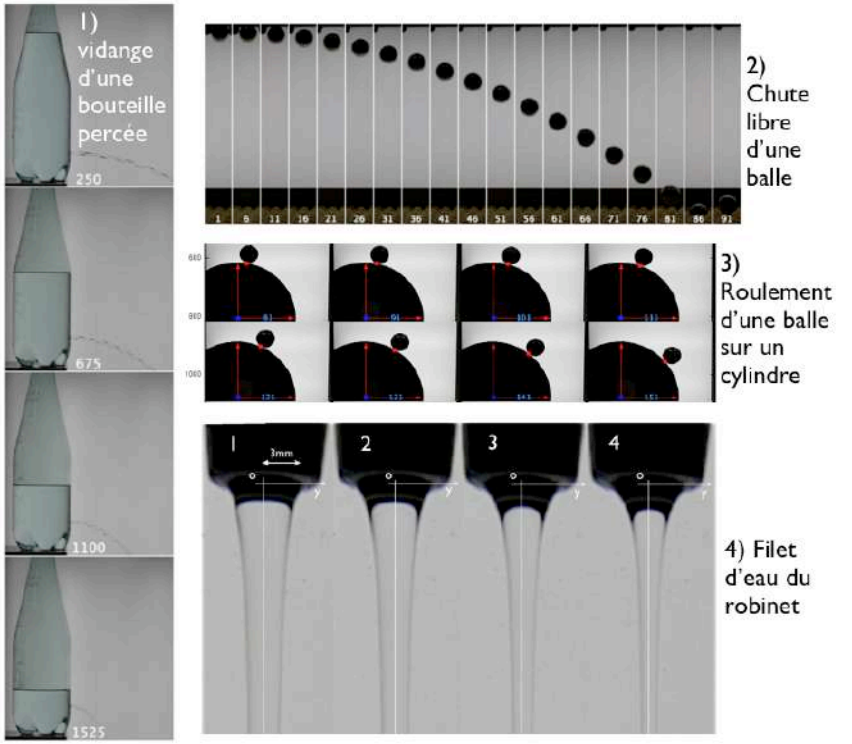



On observe un bon accord entre la formule théorique et la forme du coquillage. C'est une indication que le coquillage croît de façons récursive : le mollusque qui l'habite agrandit sa coquille lorsqu'il a grandit lui-même, et il agrandit sa coquille en proportion de sa propre taille.



### 1.6.4 Mesures sur une image multiple

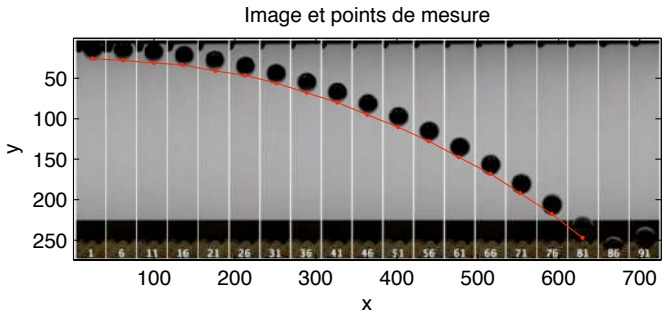
Dans l'exemple que nous venons de traiter il y a juste une image statique et nous mesurons une forme sur cette image. En général en physique, nous nous intéressons à comment une grandeur physique dépend d'un paramètre. Par exemple pour une balle en chute libre, on s'intéresse à la hauteur de la balle en fonction du temps (le paramètre qui varie c'est le temps). Dans ce cas là on capture le phénomène grâce à un film. On transforme ce film en une image multiple en mettant les unes à côté des autres les images du film. La figure ci-dessous représente quatre exemples d'images multiples :




Les trois premiers exemples représentent les temps successifs du film du phénomène comment le niveau d'eau varie lorsqu'un bouteille se vidange par un trou (problème de Torricelli, mécanique des fluides, niveau Licence 2), la chute libre d'un corps (mécanique du point, niveau Licence 1), le roulement d'une balle sur un cylindre (dynamique du solide rigide, niveau Licence 3). La quatrième image concerne la forme du filet d'eau qui coule d'un robinet (mécanique des fluides, niveau Licence 2). Les images successives représentent non pas comment ce filet d'eau change dans le temps, mais comment il change lorsqu'on diminue le débit progressivement en fermant petit à petit le robinet.

Nous allons traiter ici le cas de la chute libre d'une balle. La première chose

à faire est de prendre les points de mesure sur l'image, comme décrit plus haut. l'image obtenue est la suivante ;




et mon script :

```
a=imread('chutelibre.png');
image(a);
axis equal tight

d=[  23.8318   25.7242
   61.8139   27.8946
   99.7960   31.1502
  134.5224   33.3206
  175.7601   40.9170
  212.6570   46.3430
  251.7242   56.1099
  288.6211   68.0471
  326.6031   79.9843
  363.5000   95.1771
  401.4821  110.3700
  439.4641  127.7332
  476.3610  147.2668
  515.4283  167.8857
  552.3251  191.7601
  591.3924  217.8049
  629.3744  247.1054];

x=d(:,1);
y=d(:,2);

hold on;
plot(x,y,'r.-')

xlabel('x'); ylabel('y'); title('Image et points de mesure')
```

Notez que ici, au lieu de sauver les coordonnées des points mesurés dans un fichier sur le disque dur, j'ai copié dans mon script la sortie écran de la fonction "ginput", c'est plus rapide et plus pratique s'il y a peu de points de mesure. Sur



l'image, plutôt que de mesurer la position du centre de la balle, j'ai mesuré la position du bas de la balle parce que c'est plus précis (c'est difficile de voir sur l'image ou se situe le centre).

Je vais maintenant effectuer le changement de référentiel. Pour les "y" la procédure est la même que décrit plus haut : je prend comme origine du référentiel la première position de la balle, comme cela ma courbe va commencer à $y = 0$. Par contre, il y a une différence pour le traitement de l'axe des $x$ : la coordonnée $x$ des points de mesure ne me sert ici à rien, puisque chacun de mes points de mesure correspond à un temps différent : dans mon graphique final je trace la hauteur de chute en fonction du temps.

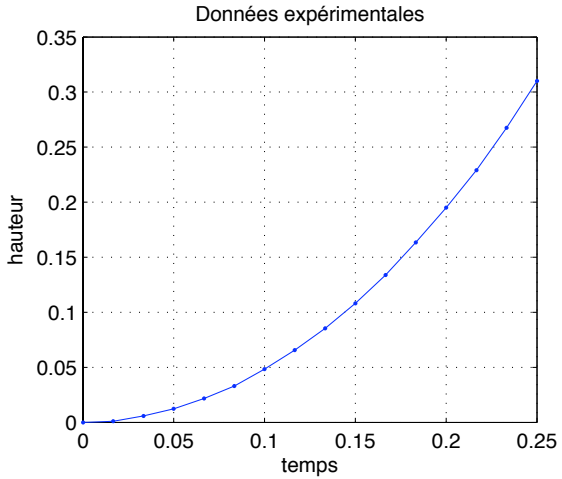
Il faut donc construire le vecteur des temps successifs du film. Le numéro affiché sur chacune des images correspond au numéro de l'image dans le film : 1,6,11,16..., 91 c'est à dire que nous avons gardé les images de 1 à 91 en sautant à chaque fois 4 images. Ce film est enregistré à 300 images par seconde, le pas de temps est donc `dt=1/300` c'est l'intervalle de temps entre deux images du film. Le vecteur des coordonnées temporelles est donc :

```
dt=1/300; % le pas de temps
tvec=(1:5:91)-1)*dt
```

Dans "tvec", j'ai soustrait 1 de sorte à ce que les points de mesure commencent au temps $t = 0$.

Pour étalon de longueur, nous savons que la hauteur de la fenêtre lumineuse est de 35,5 cm, nous pouvons en déduire la taille d'un pixel.

On obtiens le graphique suivant :



### 1.6.5 Comparer une courbe expérimentale à une formule mathématique

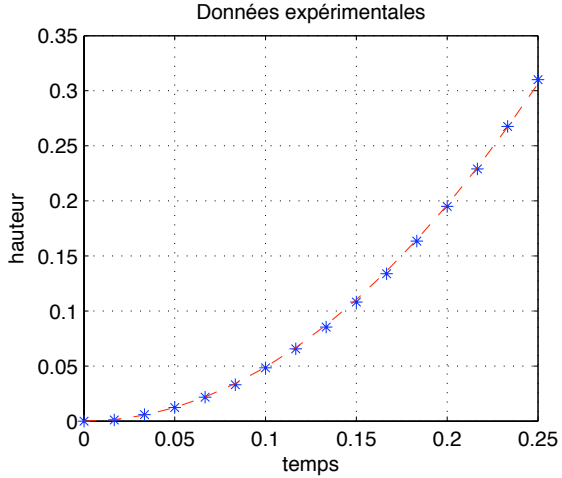
Les expériences nous donnent des points de mesure que nous pouvons tracer sur un graphique. les modèles théoriques nous donnent des formules mathématiques qui dépendent de paramètres physiques. Pour la chute libre d'un corps, le seul paramètre physique est l'accélération de la gravité $g$. La formule théorique est :

$$h = gt^2/2$$

Au script précédent, nous rajoutons les commandes :

```
hold on
g=9.81;
plot(t,g*t.^2/2,'r--');
```

ce qui nous donne le graphique :



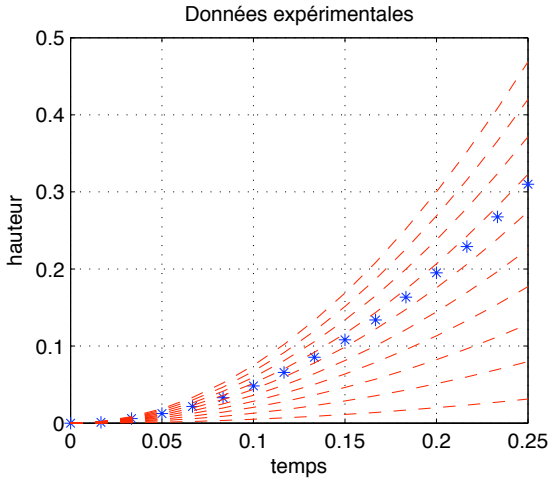

On voit que la correspondance est très bonne entre l'expérience et la théorie.

Souvent, on ne connaît pas la valeur numérique des paramètres physiques dans une formule mathématique. La comparaison entre la formule théorique et les mesures expérimentales sont alors un moyen pour déterminer la valeur de ce paramètre physique. Supposons par exemple que nous ne connaissons pas la valeur de $g$ et que nous utilisons notre expérience pour la mesurer. Je peux tester la formule pour plusieurs valeurs de $g$ en faisant une boucle :

```
hold on
gvec=linspace(1,15,10);
for g=gvec
plot(t,g*t.^2/2,'r--');
end
```

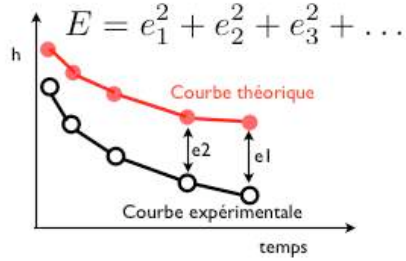


et la figure est alors :




C'est là une manière de représenter comment la formule théorique change lorsque l'on fait varier la valeur d'un paramètre physique.

Il est aussi possible de quantifier précisément l'erreur qui distingue des données expérimentales et une formule théorique pour une valeur des paramètres physiques. Voici une figure qui montre cette erreur :




Pour chaque point de mesure numéro $i$, on calcule le carré $e_i^2$ de la distance entre le point expérimental et la formule théorique. On fait la somme de toutes ces erreurs dans $E$ qui devient ainsi une mesure globale de l'erreur. Si $E$ est nul, alors la courbe théorique est exactement sur la courbe expérimentale. Par contre, plus $E$ sera grand, plus la distance entre la formule théorique et la courbe expérimentale sera grande (c'est à dire que la valeur de $g$ associée sera d'autant plus fausse).

Voici un script qui calcule cette erreur entre la mesure de la chute libre et la formule théorique pour plusieurs valeurs de $g$ que nous supposons inconnue :

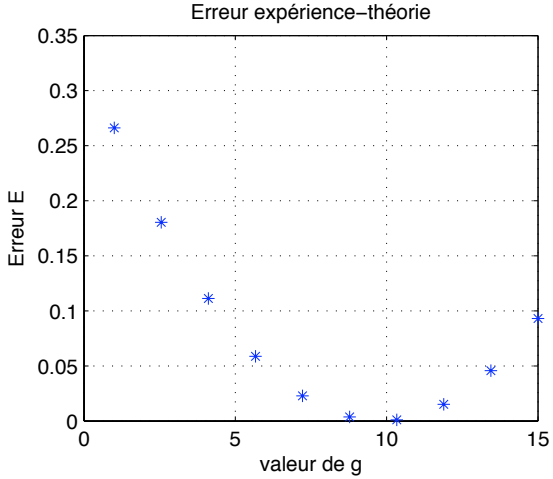
```
gvec=linspace(1,15,10);
for g=gvec
    xtheo=transpose(g*t.^2/2);
    E=sum((x-xtheo).^2);
```

```
    plot(g,E,'b*');
    hold on
end
xlabel('valeur de g'); ylabel('Erreur E');
title('Erreur expérience-théorie')
grid on
```

Dans ce script, j'ai fait une boucle pour faire évoluer la valeur numérique de $g$, et à chaque itération de la boucle, je mesure et je trace l'erreur sur un graphique. Pour faire la somme des $e_i$ à chacune itération de la boucle, j'ai utilisé la fonction `sum`, qui donne en argument de sortie la somme des éléments dans les cases du tableau donné en argument d'entrée.

Attention : "x" est un tableau colonne alors que "t" est un tableau ligne. Si je veux pouvoir les soustraire, il faut que je transpose l'un des deux avec l'opérateur de transposition `transpose`.

Et voici la figure obtenue :




On voit bien, que l'erreur devient très petite pour $g$ proche de 10. Et que pour des valeurs inférieures ainsi que pour des valeurs supérieures, l'erreur devient très grande.

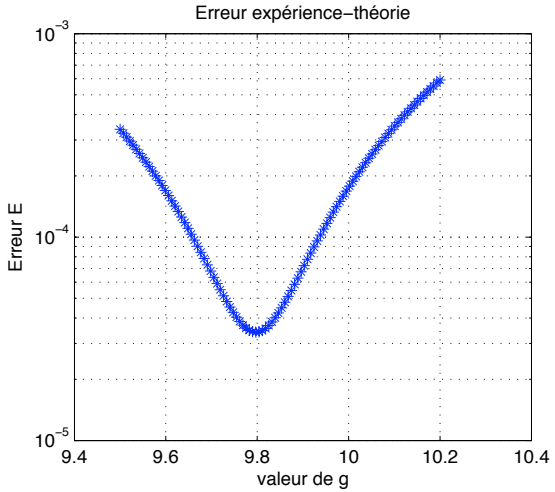
Pour mesurer avec plus de précision la valeur de $g$ qui est la plus compatible avec nos données expérimentales, je fait une boucle avec beaucoup plus de valeurs de $g$ et je trace l'erreur en coordonnées semi-logarithmiques :

```
gvec=linspace(9.5,10.2,100);
for g=gvec
    xtheo=g*t.^2/2;
    E=sum((transpose(x)-xtheo).^2);
    semilogy(g,E,'b*');
    hold on
end
```



```
xlabel('valeur de g'); ylabel('Erreur E');
title('Erreur expérience-théorie')
grid on
```

Et voici la figure :




Ici, on voit que l'erreur n'atteint jamais zéro, effectivement il y a une petite erreur de mesure (position exacte de la balle, identification du temps initial...) et des effets physiques qui ne sont pas pris en compte dans la modélisation (friction de l'air, ...). Le minimum de l'erreur est pour $g$ proche de 9.8, légèrement inférieur à la valeur acceptée de 9.81.

## 1.7 Votre compte-rendu

Ce que vous avez fait pendant les travaux pratiques est décrit dans votre compte-rendu. Pour cette UE, il ne suffit pas de programmer et de tracer les graphiques et de les voir à l'écran. Une part importante de votre travail consiste à présenter vos codes et vos résultats dans un document. Ce document est celui que vous rendrez pour être notés, que ce soit pour les TP notés ou pour l'examen final.

Le format est celui que vous utiliseriez comme support à une présentation orale de ce que vous avez fait.

Le compte-rendu comporte :

- Une page de titre avec le nom du cours, le nom et numéro du TP, les noms des étudiants qui ont participé ainsi que leurs numéros d'étudiant.
- Pour chaque question de l'énoncé de TP, une page avec les graphiques que vous avez tracés, les scripts matlab que vous avez codé. Et des commentaires qui décrivent les choses intéressantes que l'on voit sur les graphiques : la physique qui se cache derrière les données. Utiliser les informations que

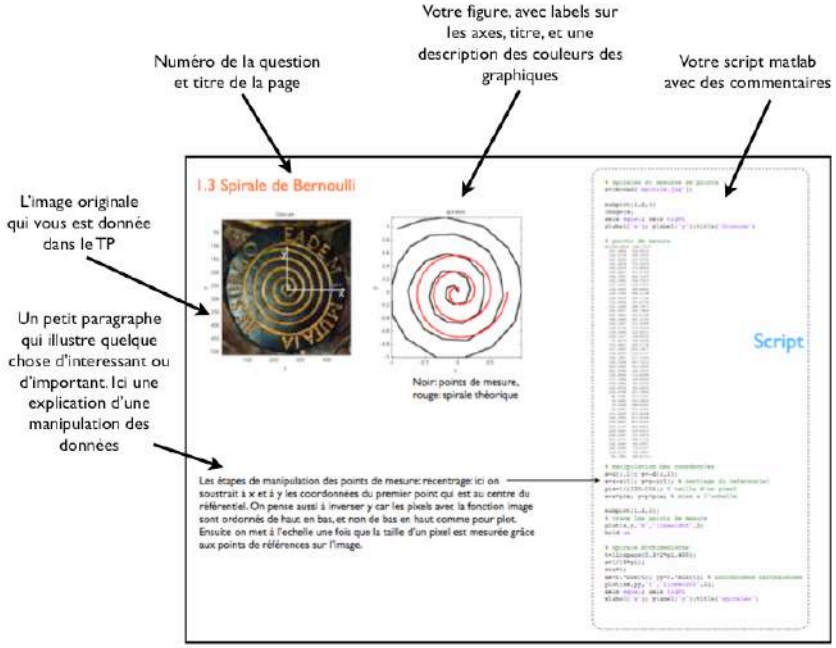
nous vous donnons dans les énoncés de TP, et n'hésitez pas à nous faire profiter des idées et connaissances que vous avez développées en dehors de notre cours.

– Pour ce que vous n'avez pas réussi à faire : décrire les erreurs que vous avez obtenu, décrire ce que vous avez tenté pour corriger ces erreurs. Il est important d'avoir une attitude positive par rapport à ce que vous n'avez pas su faire, et que cette attitude transparaisse dans le compte-rendu que vous faites de votre travail. Avoir tenté des choses, c'est un pas vers la solution, et ça compte pour donner de la valeur à votre travail.

Vous trouverez beaucoup d'exemples de compte-rendu de TP dans les corrigés des TP de l'année précédente sur le site internet du cours.

Notez bien que c'est le compte-rendu de votre travail qui est évalué. Vous devez le produire en pensant bien à mettre votre travail en valeur. Lorsque nous évaluons votre travail, on vérifie que vous avez bien compris ce qui vous est demandé, et que vous avez réussi à utiliser les outils techniques qui font l'objet de ce cours.

Voici un exemple d'une page de compte-rendu :




Important : les scripts doivent comporter des commentaires qui expliquent ce que fait chaque bloc de commande ; sinon les codes sont illisibles : on ne peut pas vous donnez de conseils de programmation, et il nous est difficile de savoir si vous n'avez rien pu faire ou bien si simplement c'est une petite erreur qui a fait que vous n'avez pas obtenu de graphique. Les graphiques doivent avoir un titre et des labels pour les axes, et une légende pour les courbes tracées.

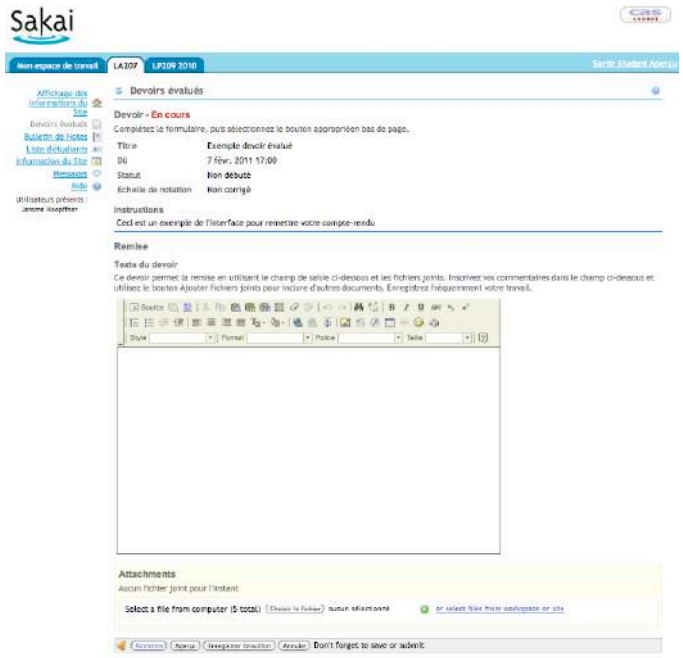


Pour rédiger votre compte-rendu, utilisez le logiciel "powerpoint" sous windows ou "openoffice" sous linux. Une fois le compte-rendu prêt, faites la transformation de votre document de travail en un document de présentation en le sauvant sous le format pdf. Le format pdf est un bon format pour envoyer des documents, c'est un format portable, c'est à dire qui peut être lu sous tous les types de systèmes d'exploitation et qui réduit la taille des fichiers par rapport à powerpoint ou openoffice.

## 1.8 Remettre votre compte-rendu

Nous utilisons le site interactif de l'UPMC pour remettre les comptes-rendu et vous communiquer vos notes de TP ainsi que les commentaires sur votre travail. C'est "australe" qui est accessible depuis votre compte "mon Upmc", ou bien directement (et plus rapidement) à l'adresse : http ://australe.upmc.fr. Vous devriez avoir un onglet "LA207" sur votre interface. Si vous ne l'avez pas c'est que vous n'avez pas été inscrit pour ce cours, et il faut vous manifester au secrétariat du L2 de mécanique.


Pour rendre un devoir, dans l'onglet LA207, cliquez sur "Devoirs évalués". En temps voulu, il y aura un lien portant le nom du TP en question. Sur la page du devoir évalué, télétransmettez le ficher de votre compte-rendu que vous avez produit pendant le TP, au format pdf. Voici l'interface sur laquelle vous pouvez télétransmettre votre compte-rendu. Cliquez sur "choisir le fichier", et vous avez aussi une zone de texte pour éventuellement vous exprimer.




Ensuite cliquez sur "remettre". Attention, votre travail n'est pas remis tant que vous n'avez pas cliqué sur "remettre" et vu la fenêtre de confirmation. De plus un email de confirmation vous est envoyé.


Avant de quitter la salle de TP, passer voir votre encadrant qui vérifiera que vous avez rendu le bon ficher au bon endroit. Pour les TPs notés, c'est lors de cette confirmation que vous pourrez signer la feuille de présence.

## 1.9 Difficultés habituelles

Dans cette section, je rajoute quelques descriptions qui correspondent à ce qui a été difficile pour vos collègues des années précédentes : évitez les embûches qui les ont gênés !



### 1.9.1 Le graphique actif

Lorsque vous avez plusieurs sous-graphiques, les commandes de tracé de graphique sont exécutées dans le sous-graphique "actif". Il s'agit du sous-graphique dans lequel à été tracé la dernière figure, ou bien le sous-graphique sur lequel on a cliqué en dernier, ou bien encore le sous-graphique rendu actif par la commande `subplot`.

### 1.9.2 Stopper un calcul

Il arrive que l'exécution d'une commande ou d'un script prenne beaucoup de temps parce qu'on s'est trompé. Pour arrêter l'exécution et pouvoir entrer d'autres commandes, cliquez dans la fenêtre du prompt, et tapez "controle-c", c'est à dire enfoncez simultanément la touche CTRL et la touche c.

### 1.9.3 Sauver les figures

Pour sauver sur le disque vos figures, il y a deux possibilité : les enregistrer au format "Matlab figure" `.fig` qui vous permet de ré-ouvrir la figure dans matlab comme si vous venez juste de la créer pour éventuellement la modifier, ou bien vous pouvez les enregistrer dans un format d'image, par exemple `.jpg`.

Pour sauver une figure : dans le menu déroulant "Fichier" ("File" en anglais) sélectionner "save as" et choisissez le format jpg et le répertoire dans lequel vous voulez enregistrer l'image produite.

N'essayez pas de mettre une figure sauvée au format `.fig` dans votre compte-rendu. Pour le compte rendu ; il faut les sauver sous un format d'image.

Une solution d'extrême urgence pour mettre une figure dans votre compte-rendu consiste à faire une capture d'écran (touche sur votre clavier), en général cela sauve comme une image tout ce que vous voyez sur votre écran. Vous pouvez ensuite insérer cette image dans votre compte-rendu et la rogner pour que n'apparaisse que la figure. La qualité de l'image obtenue ainsi sera moindre qu'en sauvant par le menu de Matlab.

# Index