Matlab: applications en mécanique
LA207
Notes de cours
Université Pierre et Marie Curie

Jérôme Hoepffner

Février 2009

Contents

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 efficace.

2 Matlab de base

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 `>>`. 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 veuille 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 met 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`.

2.1 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.

2.2 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 mon éditeur de texte préféré, 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'écrit 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.

2.3 Boucles et tests

Pour réaliser des actions répétitives, on peut utiliser la boucle `for`. J'écrit 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`. Noter que l'expression `if test` signifie "si `test` est vrai" et est un raccourcis pour l'expression plus précise `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.

2.4 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`.

2.5 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.

2.6 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
```

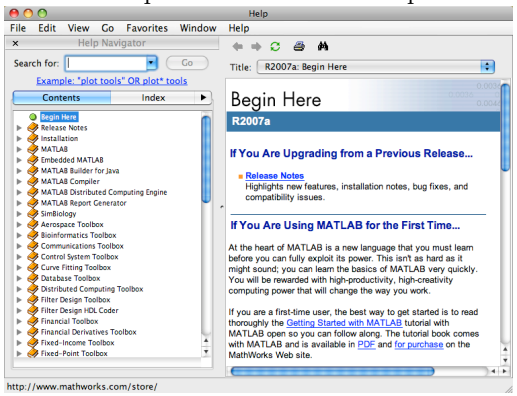
Un aspect très intéressant de cette aide, c'est qu'elle propose dans "see also" en bas de texte une liste de fonction et commandes proches de celle que vous avez demandé, ici: `if, while, switch, break, continue, end, colon`. En procédant d'aide en aide, vous pouvez rapidement apprendre beaucoup de fonctions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

2.7 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 tableau: pas de 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.

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.

3.1 Construire des tableaux par concaténation

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

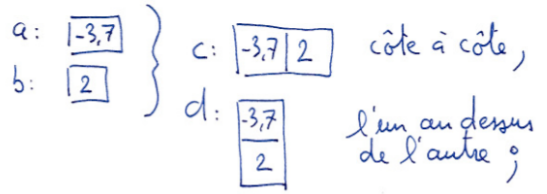
```
a=-3.7;
b=2;
```

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];
```

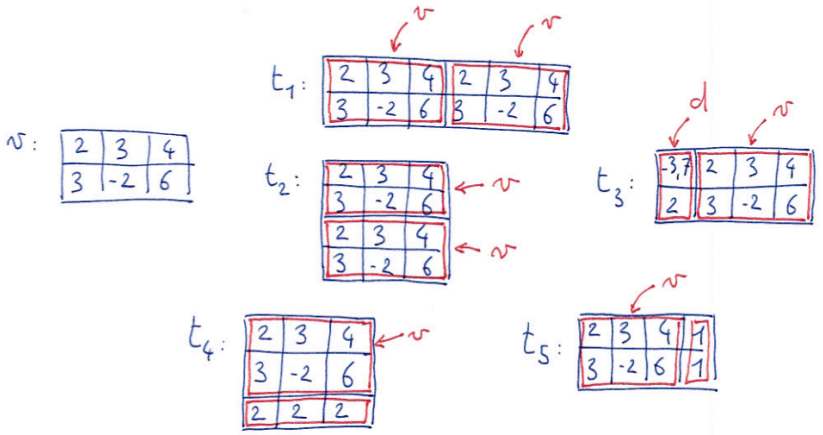


Je peux construire directement un tableau en concaténant des nombres

```
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.

3.2 Accéder aux sous-tableaux

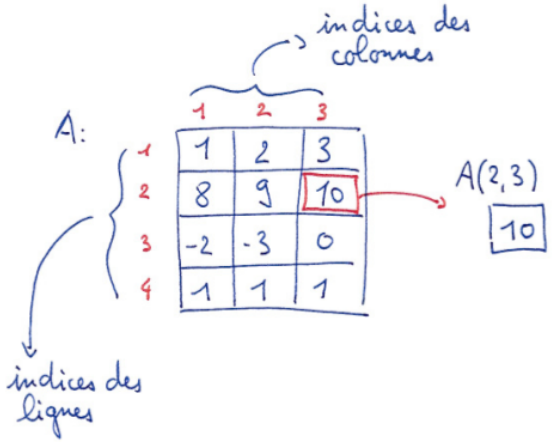
Dans la section précédente, nous avons vu comment créer des tableaux en concaténant des tableaux plus petits. Ici nous allons créer des tableaux plus petits en sélectionnant des sous-tableaux, ou bien changer directement les valeurs dans les sous-tableaux.

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

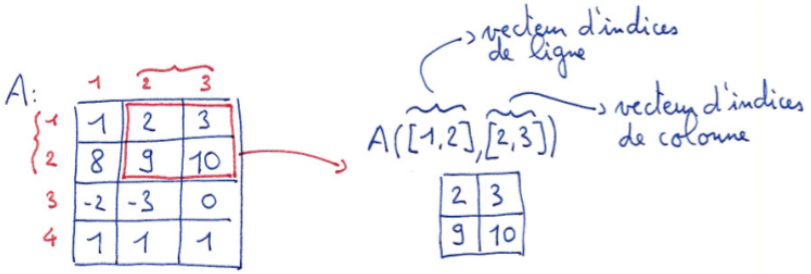
```
b=A(2,3)+A(1,1)
```

ici je met 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.

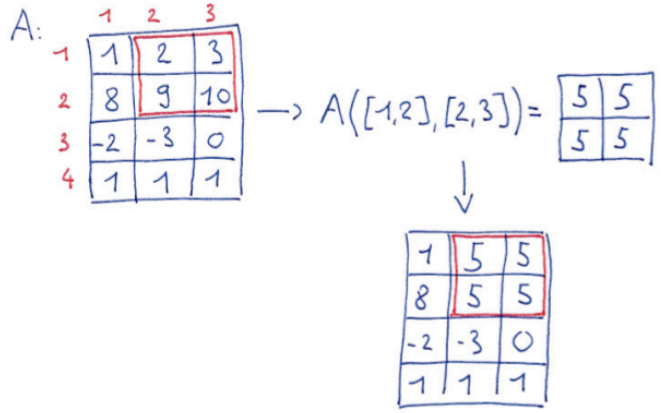
Une chose très intéressante, c'est que l'on peut changer la valeurs des sous-tableaux. Tout d'abord on change un seul élément:

```
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 =):



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

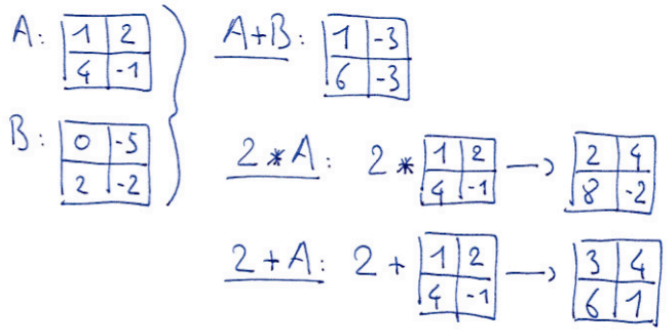
```
C=A+B
```

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.

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.

3.4.1 pour créer des tableaux

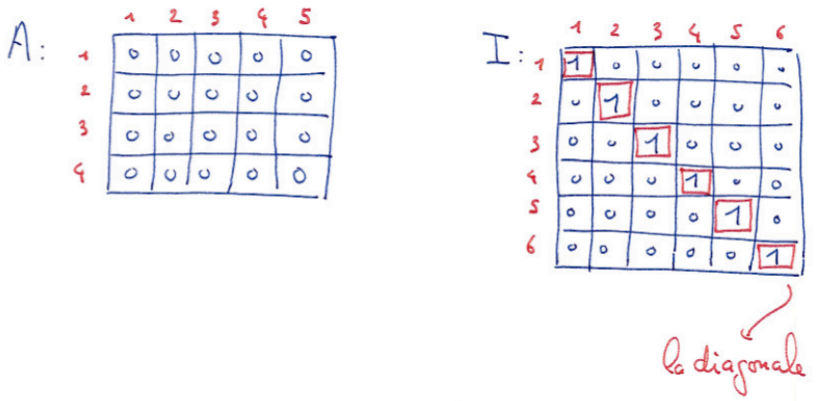
Pou créer un tableau rempli de zéros on utilise la fonction `zeros`

```
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 65. 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
```

3.4.2 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 auxquels 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))`.

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 2π, 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 π 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 π et des 1 dans les cases ou `v` est égal à π. La somme avec `sum` des éléments de ce vecteur est le nombre d'éléments de `v` égaux à π. Maintenant un peu plus subtil, on veut connaître les indices des éléments de `v` qui sont égaux à π. 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 π. 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 π 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 π, 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 π, 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.

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.*

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)
```

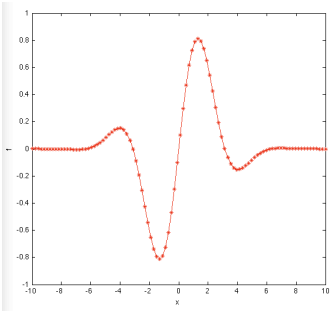
je n'ai pas précisé quels sont les points d'abscisse (les x), je n'ai donné que les ordonnées. Dans ce cas matlab suppose que le vecteur des abscisses est $(1, 2, 3, 4...)$, jusqu'à n. Pour tracer correctement la fonction, il faut entrer

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

Je peux choisir la couleur et le style de ligne

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

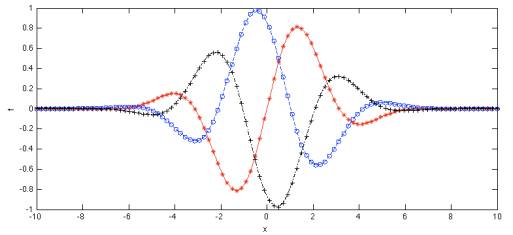
ici, je demande a ce que le graph soit en rouge (`r`), avec des astérisques aux points $x(i)$ (`*`), et une ligne en pointillés (`--`).



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.

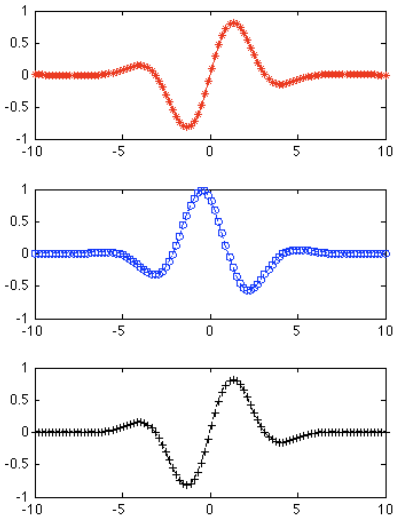
On peut également utiliser les commandes `hold on` et `hold off` avec la séquence suivante

```
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,f1,'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.

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
```



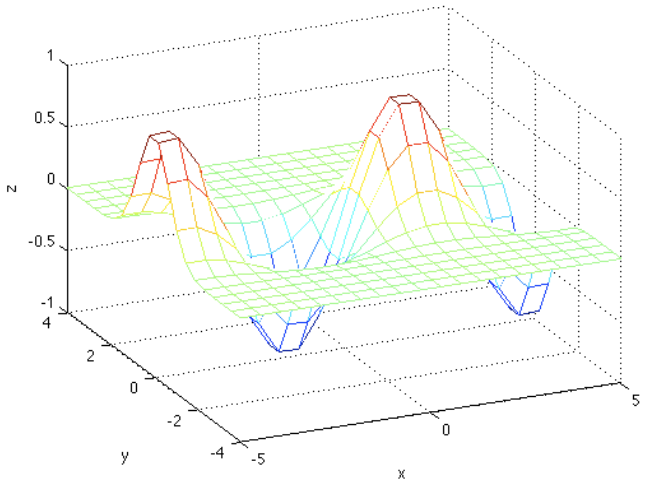
```
Y =
     4     4     4
     5     5     5
     6     6     6
```

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 `mesgrid` 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

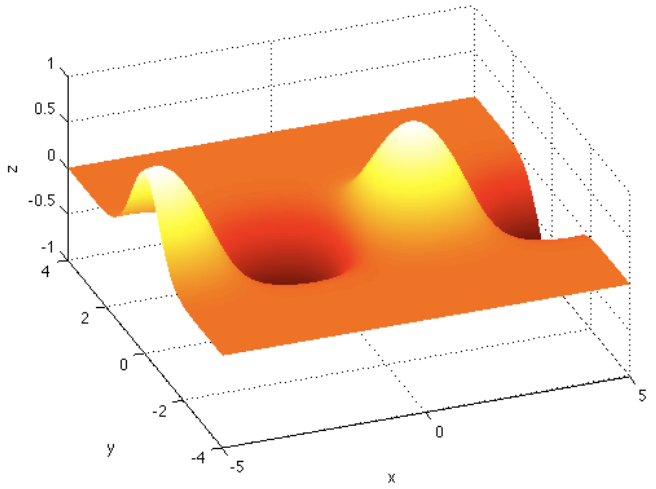
```
for t=linspace(0,2*pi,30)
  mesh(X,Y,sin(X+t).*exp(-Y.^2));
  camorbit(380*t/(2*pi),0);
  drawnow
end
```

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 2π.

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).



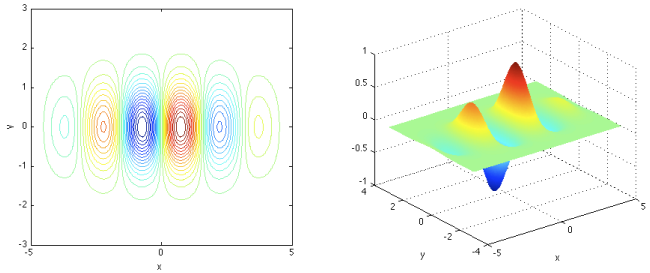
5.3 Isovaleurs

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.

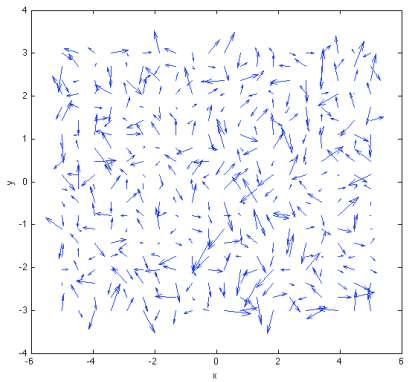


5.4 Champs de vecteurs

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)`.

Index