2013

FILS-Master2

Traitement du Signal Numérique
Applications en MATLAB

Introduction à Matlab

MATLAB est un langage de calcul numérique constitué d'un noyau de routines graphiques et d'algorithmes de calculs pré-programmés auxquels viennent se greffer des modules optionnels spécialisés.

Les quelques renseignements et explications que l'on trouvera ci-dessous, n'ont pas la prétention de se substituer à ceux que l'on ne manquera pas de trouver (chercher?) dans le manuel (ou le help) fournit avec le logiciel, mais ils répondent aux questions et problèmes que l'on est amené à se poser lors d'un premier contact avec MATLAB.

Principe général d'utilisation

Le principe général de programmation sous MATLAB est d'exécuter des commandes les unes à la suite des autres un peu à la manière des anciens interpréteurs BASIC. Le langage de programmation proprement dit de MATLAB ressemble assez au C avec toutefois des différences notables. Il existe deux méthodes pour faire exécuter des commandes au logiciel : les taper directement dans la fenêtre de commande.

Exemple :

a=1 :10 ;
b=5 ;
a*b
ans =
5 10 15 20 25 30 35 40 45 50

placer les mêmes commandes dans un fichier texte (ASCII) à l'aide d'un éditeur quelconque (par exemple jetedit) puis donner à ce fichier un nom quelconque suivi de l'extension **.m**. Par exemple **essai.m**. Revenir alors au prompt MATLAB et entrer au prompt le nom du fichier (sans son extension) suivi de enter :

essai
ans =
5 10 15 20 25 30 35 40 45 50

MATLAB est un langage structuré il est donc également possible de définir des fonctions et des procédures pouvant être utilisées au cours d'un calcul. Une fonction est un ensemble de commandes placées dans un fichier ASCII dont l'entête porte le mot clef réservé **function** suivi de la déclaration de la fonction.

Les principales commandes système de gestion du disque DOS (cd,dir , mkdir) ou UNIX (pwd, cd, ls , mkdir) fonctionnent dans l'environnement MATLAB.

Les variables

Il existe sous MATLAB les types de variables habituels c'est à dire entier, réel, complexe, chaîne de caractères, tableaux d'entiers, de réels, de complexes et de chaînes de caractères. Les types structurés font leur apparition à partir de la version 5 mais les pointeurs sont cependant toujours absents. Pour l'ensemble des variables utilisées l'initialisation tient lieu de déclaration. D'une manière similaire à la quasi totalité des langages de programmation il existe une distinction entre variables locales et variables globales. Par défaut toute variable est locale, la commande **global** permet de d'étendre la portée de la variable.

Les variables complexes

Elles s'obtiennent à l'aide de l'emploi du nombre imaginaire **i** ou **j**. Attention en dehors du contexte de l'emploi dans un nombre complexe **i** ou **j** peuvent servir d'identificateurs à des variables quelconques. Les nombres suivants sont des nombres complexes :

```
a=1+2i ;
a=1+2*i ;
a=32j ;
a=32*j ;
a=32*sqrt(-1) ;
```

Les opérateurs **real()** et **imag()** permettent d'obtenir la partie réelle et la partie imaginaire d'un nombre complexe :

```
a=2+6i ;
real(a)
ans =
        2
imag(a)
ans =
        6
```

a'est le complexe conjugué de **a** (on peut également utiliser la fonction **conj()**):

```
a=6+2j ;
a'
ans =
        6-2j
```

conj(ans) (**ans** est une variable contenant le résultat de la dernière commande)

```
ans =
        6+2j
```

Les variables tableaux

Initialisation

Sous MATLAB un tableau peut se présenter sous la forme d'une colonne ou d'une ligne. La distinction est importante puisqu'elle définit la portée des opérateurs arithmétiques s'appliquant à ces variables. Il existe plusieurs moyens pour créer et initialiser une variable de type tableau.

La création « naturelle » : elle est très utile pour les tableaux de petite dimension.

```
a = [ 1 2 3 4 5 6]
ans =
```

1 2 3 4 5 6

La commande précédente crée une ligne de 6 valeurs de 1 à 6. Pour obtenir une colonne on utilise soit l'opérateur transposé **‘** :

a = [1 2 3 4 5 6] ‘

ans =

1
2
3
4
5
6

soit le séparateur de lignes **;** :

a=[1 ;2 ;3 ;4 ;5 ;6]

ans =

1
2
3
4
5
6

Si le tableau à créer se présente sous la forme d'une suite de raison (entière ou réelle) connue, on peut également utiliser la commande suivante :

a=1 :1 :6 ;

L'exécution de cette commande crée également une ligne de 6 valeurs de 1 à 6 par pas de 1. La syntaxe générale de cette commande est donc la suivante : NomVariable=ValeurMin :Raison :ValeurMax ; .Pour obtenir une colonne :

a=[1 :1 :6]' ;

Attention : l'opérateur transposé ‘ sert également à calculer le conjugué d'un nombre complexe. La commande **a=1 :1 :6' ;** génère donc toujours une ligne de 6 valeurs de 1 à 6' c'est à dire de 1 à 6 (6'=6).

Il existe deux fonctions possédant un fonctionnement similaire : les fonctions **linspace** et **logspace**. Elles créent des variables tableaux dont, respectivement, les éléments, ou les logarithmes (en base 10) des éléments, sont linéairement espacés. On ne fournit plus la raison de la suite mais le nombre total d'éléments souhaité.

linspace(1,6,6)
ans =
1 23 4 5 6

logspace(1,5,5)
ans =
10 100 1000 10000 100000

- Une autre manière de procéder est d'utiliser une boucle **for** :

```
for i=1 :1:6
        a(i)=cos(pi*i/3) ;
end
```

L'index **i** parcourt l'ensemble de valeurs définies par la commande **1 :1 :6** soit : {1, 2, 3, 4, 5, 6}.

- Deux commandes permettent également de générer des tableaux de 0 ou de 1, ce sont les commandes suivantes :

a=zeros(1,10) ;

b=ones(10,1) ;

Ces deux commandes créent respectivement une ligne de 10 zéros et une colonne de 10 uns. La syntaxe d'utilisation est donc :

zeros(NombreLigne,NombreColonne) ;
ones(NombreLigne,NombreColonne) ;

Indexation d'une variable tableau

Il est souvent très utile de pouvoir accéder à un élément ou à un ensemble d'éléments d'un tableau. Attention l'indexation du tableau commence à l'indice 1:

a=0 :2 :8 ; (création d'une ligne de 5 éléments)
a(3) (référence au troisième élément)
ans =
4
a(2 :5) (référence aux 2, 3, 4 et 5ième éléments)
ans =
2 4 6 8

Il peut être utile également d'opérer une indexation « différée » :

a=1 :100 ;
i=[3 5 7 11 13] ;
a(i)
ans =
3 5 7 11 13

Les opérateurs arithmétiques

Ce sont les opérateurs +, -, *, /, ^ et ‘ qui désignent respectivement : l'addition, la soustraction, la multiplication, la division, la puissance et la transposition. Ils s'appliquent aux grandeurs entières, réelles et complexes scalaires sans difficulté majeure. Leur utilisation pour des matrices ou des tableaux nécessite toutefois quelques explications.

Le ;

Le lecteur averti aura remarqué sa présence ou son absence à la fin de chaque commande. Cet opérateur possède en fait deux fonctions :

Sa présence à la fin d'une commande permet ou non d'autoriser l'affichage (ce que l'on appelle l'écho d'une commande) à l'écran du résultat de l'exécution de la commande :

a=10*10 ; (pas d'affichage du résultat)
a=10*10 (affichage du résultat)
ans=
100

Sa présence est donc importante sous peine de voir un calcul ralenti par l'affichage inutile d'un ensemble de calculs.

Dans l'initialisation d'une matrice il sert de séparateur de lignes.

a=[1 2 3 ;4 5 6 ;7 8 9]
ans =
1 2 3
4 5 6
7 8 9
a=[1 2 3 4 5 6 7 8 9]
ans =
1 2 3 4 5 6 7 8 9

+ *(et -)*

La commande **a+b** (ou **a-b**) additionne (ou soustrait) le contenu de **b** à **a**. **a** et **b** doivent être des objets de même dimension : même nombre de lignes et de colonnes sauf si l'une des deux variables est un scalaire :

a=[1 3 5 7 11 13] ; (**a** est une ligne de 6 éléments)
b=[1 ; 2 ; 3 ; 4 ; 5 ; 6] ; (**b** est une colonne de 6 éléments)
a+b'
ans =
2 5 8 11 16 19 (**a+b'** est une ligne de 6 éléments)

a+5.8
ans =
6.8000 8.8000 10.8000 12.8000 16.8000 18.8000

a+2*b
Eeeeeerrrrrrrreeeeeeuuuuuuuurrrrrrr ! ! ! ! ! ! ! !

* *et* .*

La commande **a*b** est le produit matriciel des matrices **a** et **b**. Le nombre de colonnes de **a** doit être égal au nombre de lignes de **b**, sauf si **a** ou **b** est un scalaire :

a=[1 3 5 7 11 13] ; (**a** est une ligne de 6 éléments)
b=[1 ; 2 ; 3 ; 4 ; 5 ; 6] ; (**b** est une colonne de 6 éléments)
a*b
ans =
183

Il peut être intéressant d'avoir également un produit élément par élément des matrices **a** et **b**. Cette opération est réalisée à l'aide de l'opérateur **.*** . Les matrices **a** et **b** doivent être de même dimension :

a=[1 2 3 ; 4 5 6] ;

b=[7 8 9 ; 10 11 12] ;
a.*b
ans =
7 16 27
40 55 72

/ *et* ./

Si **a** et **b** sont deux matrices carrées **a/b** est égal à **a*inv(b)** où **inv(b)** représente la matrice inverse de la matrice **b**. Si **a** est une matrice (m,n) et **b** un vecteur (une colonne), **a/b** est solution du système **bx=a** où **x** est inconnu (l'algorithme de calcul est la méthode d'élimination de Gauss). **a./b** est la matrice résultant de la division élément par élément de **a** par **b**.

^ *et* .^

Si **a** est une matrice et **b** un scalaire **a^b** est la matrice **a** à la puissance **b**. Si **b** est entier **a^b** est obtenue par multiplications successives. Si **b** est négatif **a** est d'abord inversée. **a.^b** est la matrice résultant de l'opération consistant à prendre chaque élément de **a** à la puissance **b** (**b** doit être scalaire).

Les instructions conditionnelles

Schématiquement on retrouve les mêmes instructions que dans les langages évolués habituels:

- instruction **for**.

  for i=1:1:10 (c'est à dire: for index_de_boucle=valeur_debut:pas:valeur_fin)
  a(i)=cos(2*pi*i/3);
  end
  Noter qu'il est plus simple (et élégant!) de faire:
  i=1:1:10;
  a=cos(2*pi*i/3);

- instruction **if**:

  % a est un scalaire
  if ((a<5)|(a>6)) (c'est à dire si a<5 ou a>6)
  a=0;
  elseif ((a==5)|(a==6) (a=5 ou a=6)
  a=1;
  else
  a=-1;
  end

- instruction **while**

  i=1;
  while (i<11)
  a(i)=cos(2*pi*i/3);
  i=i+1;
  end

- instruction **switch** (MATLAB 5). Idem qu'en C, mais attention, pour chaque branchement la présence du 'break du langage C' est implicite:

  switch a
  case 1, disp('égal 1')
  case 2, disp('égal 2');
  otherwise, disp('j''en sais rien!');
  end

  Remarque: après l'instruction **disp** on peut ou non mettre un point virgule la présence du break est implicite (j'insiste!): si **a=1**, l'exécution de ce bloc de commandes donnera **égal 1**.

Les fonctions

Elles permettent de structurer un programme en plusieurs taches élémentaires. Une fonction (ou une procédure) ne doit pas (en principe !) excéder une vingtaine de lignes de commandes. Les fonctions sont déclarées dans des fichiers texte avec l'extension **.m** (par exemple **essai.m** ou **laplacien.m**). Chaque déclaration de fonction fait donc l'objet d'un fichier. La première ligne du fichier doit contenir la déclaration de la fonction :
function [var_out1 var_out2]=toto(var_in1,var_in2,var_in3)

Le nom du fichier doit être le même que celui de la fonction (ce n'est pas une obligation mais c'est conseillé). Dans l'exemple ci-dessus le fichier doit donc s'appeler **toto.m**.

Lorsque l'on programme une fonction MATLAB il est souvent utile de prévoir de pouvoir lui passer en paramètres des variables de type tableaux. Les instructions de calculs utilisant ces variables doivent donc être prévues pour manipuler des variables tableaux. L'exemple ci-dessous montre comment réaliser facilement (et sans l'utilisation d'une boucle **for**)le tracé de

la loi de perte de charge de Blasius: $\lambda \approx 0.316.\widetilde{Re}^{-0.25}$. On déclare d'abord la fonction dans une fichier texte appelé **blasius.m**.

function lambda=blasius(reynolds);
%calcul de la perte de charge de Blasius
lambda=0.316*reynolds.^-0.25; (noter la présence du **.^** au lieu d'un ^)
Sous MATLAB on crée le tableau des valeurs du Reynolds et l'on appelle la fonction **plot**.

```
re=logspace(4^E3,1^E7,100);
plot(re,blasius(re))
grid
xlabel 'Re'
ylabel 'Coef. de perte de charge'
title 'Loi de Blasius'
```

L'opération consistant à adapter les calculs réalisés par une fonction pour des variables tableaux est pompeusement appelée par MATLAB 'vectorisation de la fonction'. L'avantage procuré par cette façon d'écrire les fonctions utilisateurs est que l'on évite l'emploi de boucles et surtout que l'on peut utiliser les fonctions pré-programmées de MATLAB: dans l'exemple ci-dessus on aurait pu utiliser la fonction **fplot** à la place des commandes de création du tableau **re** et de création du graphique:

```
fplot('blasius',1^E3,1^E7)
grid
xlabel 'Re'
ylabel 'Coef. de perte de charge'
title 'Loi de Blasius'
```

Une fonction rend normalement la main au programme appelant lorsque la fin de son bloc principal d'instructions est atteinte. On peut forcer le retour à l'aide de la commande **return**.

A partir de la version 5 de MATLAB, il est possible de déclarer des sous fonctions d'une fonction. C'est un essai de structuration complète du langage, essai malheureusement non transformé car les sous fonctions demeurent toujours inconnues (elles sont donc locales) des parties du programme situées à l'extérieur du fichier où elles sont déclarées (voir à ce sujet l'aide en ligne sur les fonctions: **help function**).

Fonctions de gestion de fichiers

cd ../repertoire_machin/truc/chouette
comme sous unix!

unix 'cp toto.dat ../chemin/toto.dat'
exécute (sous unix seulement!) la commande définie entre les **' '**.

load toto.dat
super pratique: si toto.dat est un fichier ascii où des données sont rangées en format de n lignes par m colonnes, la fonction load récupère ces données et les stocke dans une variable matrice (n,m) appelée toto.

save toto.dat a –ascii
l'inverse: range la variable MATLAB **a** dans le fichier **toto.dat**.

help toto
si, quand vous avez programmé le fichier **toto.m,** vous avez placé en 1ère ou 2ième ligne des commentaires à l'aide du symbole **%**, ceux-ci sont affichés.

lookfor machin

recherche dans l'aide de tous les fichiers connus possédant l'extension **.m** la chaîne de caractères **machin**. Pratique mais assez long à l'usage!

keyboard

n'a rien à voir avec de la gestion de fichiers mais mérite d'être connu: un peu équivalente au breakpoint du C, cette instruction placée dans un programme interrompt son exécution et redonne la main à l'utilisateur pour par exemple consulter le contenu de variables. L'exécution est reprise avec la commande **return**.

clear all ou **clear a b c**

clear all efface toutes les variables du programme. **clear a b c** efface seulement les variables **a**, **b** et **c**.

Fonctions graphiques

plot(x,y,'r')

si **x** et **y** sont deux vecteurs de même longueur, trace le vecteur **y** en fonction de **x** en utilisant la ligne rouge (voir aussi la commande **subplot,** bien pratique pour créer plusieurs graphiques dans une même fenêtre).

plot(x1,y1,'r',x2,y2,'b')

même chose mais trace simultanément **y2**=f(**x2**) en bleu sur le même graphique.

plot3(x,y,z)

si **x**, **y** et **z** sont trois vecteurs de même longueur, trace le graphe **z**=f(**x**,**y**).

loglog(x,y,'c')

si **x** et **y** sont deux vecteurs de même longueur, trace en échelles logarithmiques le vecteur **y** en fonction de **x** en couleur cyan. Voir également **semilogx** et **semilogy**.

hold on

Si l'on utilise deux fonctions **plot** l'une après l'autre seule la dernière exécutée est affichée (elle efface la précédente), sauf si le mode **hold** est activé (**hold off** pour désactivé).

grid on rajoute une grille au graphique.

xlabel 'u(m/s)'

légende de l'axe des abscisses. **ylabel** pour les ordonnées, **title** pour le titre.

gtext 'effet truc'

en cliquant avec la souris (bouton gauche) le texte est placé sur le graphique à la position du curseur (à peu près!).

text(x,y,'effet machin')

même résultat en donnant les coordonnées du texte (justification à gauche du point par défaut).

fplot('toto',[0 10])

trace le graphe de la fonction **y=toto(x)** pour x∈[0,10]. La fonction **toto** doit être "vectorisée".

figure crée une nouvelle fenêtre graphique.

clf efface le contenu d'une fenêtre graphique.

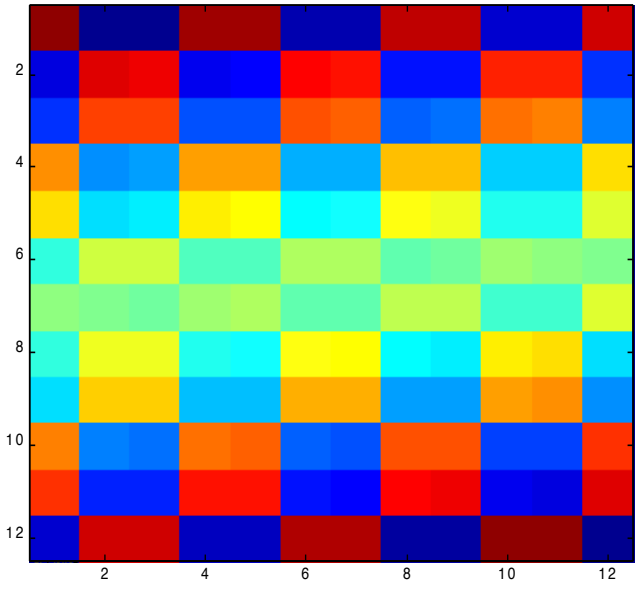
Exemple d'affichage d'une matrice sous la forme d'une image :

```
x=magic(12)
```

```
x =
   144     2     3   141   140     6     7   137   136    10    11   133
    13   131   130    16    17   127   126    20    21   123   122    24
    25   119   118    28    29   115   114    32    33   111   110    36
   108    38    39   105   104    42    43   101   100    46    47    97
    96    50    51    93    92    54    55    89    88    58    59    85
    61    83    82    64    65    79    78    68    69    75    74    72
    73    71    70    76    77    67    66    80    81    63    62    84
    60    86    87    57    56    90    91    53    52    94    95    49
    48    98    99    45    44   102   103    41    40   106   107    37
   109    35    34   112   113    31    30   116   117    27    26   120
   121    23    22   124   125    19    18   128   129    15    14   132
    12   134   135     9     8   138   139     5     4   142   143     1
```

La commande **imagesc** crée une image par mise à l'échelle des valeurs d'une matrice générée par magic

```
imagesc(x)
```



Calcul numérique

fzero('toto',[0 10])

calcule le passage à zéro de la fonction **toto** dans l'intervalle [0 10]. Dans cet intervalle la fonction doit changer de signe.

polyfit(x,y,n)

renvoie les coefficients du polynôme P de degré **n** tel que: **y**(i)=P(**x**).

[t,y]=ode45('df_sur_dt',[to, tf],yo)

résolution numérique par la méthode de Range-Kutta du système différentiel **y**=f(**t**) décrit dans la fonction **df_sur_dt** entre les instants **to** et **tf** avec pour conditions initiales **yo**.

i=find(machin<2)

si **machin** est un vecteur trouve les indices du vecteur vérifiant l'expression **<2**. A noter que **machin(i)** est un vecteur de taille différente de **machin** et qui ne contient que les éléments vérifiant l'inégalité.

Fonctions mathématiques

Fonction racine, exponentielle et logarithme :
Sqrt
Exp
Log
Log2
Log10
Pow2

Fonctions trigonométriques directes :
Sin
Cos
Tan
Cot

Fonctions trigonométriques inverses :
Asin
Acos
Atan
Atan2
Acot

Fonctions hyperboliques :
Sinh
Cosh
Tanh
Coth

Fonctions hyperboliques inverses :
Asinh
Acosh
Atanh
Acoth

Signaux et Systèmes Discrets

Définition
Un signal discret est défini par des valeurs mesurées au moment de temps discret.

Dirac

$$\delta(n)=\begin{cases} 1 & n=0 \\ 0 & n\neq 0 \end{cases}$$

Exercice

1. Représenter graphiquement les signaux :
   a) x[n]=0.8 δ[n-333] pour 300<= n <=350
   b) x[n]= δ[n]-0.5δ[n-1]+0.3δ[n-2]-2δ[n+1] pour -10<= n <=10

Indication :
Utiliser les fonction zeros, stem ….

Échelon

$$u(n)=\begin{cases} 1 & n\geq 0 \\ 0 & \text{en rest} \end{cases}$$

Exercice

2. Représenter graphiquement les signaux :
   a) x1[n]=u[n]
   b) x2[n]=u[n-2]
   c) x3[n]=0.7(u[n+3]-u[n-3] -5<=n<=10

Convolution linéaire de deux séquences :

$$x[n]=x_1[n]*x_2[n]=\sum_{k=-\infty}^{\infty}x_1[k]*x_2[n-k]$$

Syntaxe :
conv(x1,x2)

Exercice :
Calculer le produit de convolution linéaire de :
x1[n]=u[n]-u[n-5] 0<=n<=10
x2[n]=$(0.9)^n$ 0<=n<=20

Solution :

```
X1=[ones(1,5),zeros(1,6)] ;
N=0 :20 ;
X2=0.9.^n ;
X=conv(x1,x2) ;
subplot(2,2,1), stem(0 :10,x1), title('x1'),grid
subplot(2,2,2), stem(n,x2), title('x2'),grid
subplot(2,1,2), stem(0 :length(x)-1,x), title('x1'),grid
```

Transformée de Fourier Discrète

Transformée de Fourier à temps discrets est :

$$X(e^{j\omega})=\sum_n x[n]e^{-j\omega n}$$

où $\omega=2\pi f/F_s$

C'est une fonction à fréquence continue. Il est d'usage de la représenter sur l'intervalle (-1/2,1/2) ou (0.1), du fait de sa périodicité.

Donc, la transformée de Fourier discrète est :

$$X(k)=\sum_n x[n]e^{-j\frac{2k\pi}{N}n}$$

Syntaxe :

Y=fft(x) ;

Exemple : Décalage en temps

Soit {x[n]} un signal nul à l'extérieur de l'intervalle {-n0,…,n1} où n0 et n1 sont deux entiers positifs et soit y(n)=x(n-n0) obtenu par décalage en temps de n0 échantillons.

1. Déterminer la TFtd de {x[n]} en fonction de celle de {y[n]}
2. Ecrire un programme qui vérifie le résultat précédent pour n0=5. Pour cela prendre x[n] égal entre –5 et 5 et y(n)=x(n-5). Pour évaluer numériquement la TFtd sur 256 points de fréquence uniformément répartis sur l'intervalle (0.1) on utilise la fonction fft.

Solution

```
Clear ; clg
Lfft=256 ;
f=(0 :Lfft-1)/Lfft ;
n0=5 ;n1=5 ;
yt= ones(n1+n0+1,1) ;
yf=fft(yt,LFFT) ;
xf=yf.*exp(+2*j*pi*5*f') ;
plot(real(xf)) ;
plot(imag(xf)) ;
```

Exercice :

Calculer les transformées de Fourier discrètes des séquences :

1. x1[n]=u[n]-u[n-10]
2. x2[n]=sin(nπ/5)
3. x3[n]=cos(nπ/5)

```
n=0 :20 ;
zeros=[ones(1,10),zeros(1,11)] ;
X=fft(x1) ;
plot(X) ;
```

ou

```
w=-pi :2*pi/512 :pi-2*pi/512 ;
plot(w,fftshift(abs(x1))),grid
plot(w,fftshift(angle(x1))),grid
```

Système à temps discrets

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

$$X(z) = \sum_{n} x[n] z^{-n}$$

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}$$

La réponse à une impulsion d'un système linéaire invariant

Syntaxe
[h,t]=impz(b,a)
[h,t]=impz(b,a,n)

Exercices :

1. y[n]-0.9y[n-1]=0.3x[n]+0.6x[n-1]+0.6x[n-2]

2. $H(z) = \frac{1 + 0.5 z^{-1}}{1 - 1.8\cos(\pi/16) z^{-1} + 0.8 z^{-2}}$

Solution :

1.
```
b=[0.3,0.6,0.6] ;
a=[1,-0.9]

[h,t]=impz(b,a) ;
impz(b,a),grid
stem(t,h),grid
```

2.
```
b=[1,0.5]
a=[1,-1.8*cos(pi/16)),0.81] ;
h=impz(b,a) ;
impz(b,a),grid
```

Réponse d'un système linéaire à un signal d'entrée

Syntaxe :
Y= filter(b,a,x) ;

Exercice :
Déterminer la sortie d'un système défini par :

a) y[n]-0.9y[n-1]=0.3x[n]+0.6x[n-1]+0.6x[n-2]

b) $$H(z)=\frac{1+0.5z^{-1}}{1-1.8\cos(\pi/16)z^{-1}+0.8z^{-2}}$$

à la séquence d'entrée x1[n]=u[n]-u[n-10], 0<=n<=40

```
b=[0.3,0.6,0.6]
a=[1,-0.9]

x=[ones(1,10),eros(1,31)] ;
y=filter(b,a,x) ;

n=0 :40 ;
subplot(3.1.1),stm(n,x),grid,title(x[n]')
subplot(3.1.2),impz(b,a), grid, title,('h[n]')
subplot(3.1.3), stem(n,y),grid,title('y[n]')

b=[1,0.5]
a=[1.-1.8*cos(pi/16),0.8] ;
y=filter(b,a,x) ;
subplot(3.1,1),stem(n,x),grid,title('x[n]')
subplot(3.1,2),stem(b,a),grid,title('h[n]')
subplot(3.1,3),stem(n,y),grid,title('y[n]')
```

Réponse en fréquence SDLIT

[H,W]=freqz(b,a,n)
[H,F]=freqz(b,a,n,Fs)

Exercice
Déterminer la réponse en fréquence du système défini par :

1.y[n]-0.9y[n-1]=0.3x[n]+0.6x[n-1]+0.3x[n-2]

2. $H(z)=\frac{0.634-0.634z^{-2}}{1-0.268z^{-2}}$

b=[0.3,0.6,0.3]
a=[1,0.9]
[H,W]=freqz(b,a) ;.

figure(2) ; freqz(b,a) ;

Les pôles et les zéros d'un système linéaire IT

La fonction z plane permet d'afficher les zéros et les pôles dans le cas d'un système LIT si on connaît leur valeurs ou les coefficients a_k et b_k de l'équation aux différences finies.

Syntaxe
zplane(z,p) ;

Exemple :
1. La fonction de transfert d'un SDLIT a un zéro de valeur r=1/2 exp(jπ/3). Sachant que la fonction a des zéros en r*,1/r et 1/r* et deux pôles en q=1/3 exp(j2π/3) et q*, représenter le diagramme des pôles et des zéros et trouver les valeurs de ak et bk de la fonction de transfert aux différences finies.

r=1/2 * exp(j*pi/3) ;
q=1/3*exp(j*2*pi/3) ;
z=[r ;conj(r) ,1/r,1/conj(r)] ;
p=[q ;conj(q)] ;
zplane(z,p) ;
b=poly(z)
a=poly(z)
Vérification
roots(b)
roots(a)
zplane(b,a)

2.Représenter les zéros et les pôles pour les systèmes à temps discret définis par :

a) $H(z)=\frac{1-0.5z^{-1}}{1-1.8\cos(\pi/16)z^{-1}+0.81z^{-2}}$

b) y[n]+0.1y[n-1]+0.1y[n-3]+0.1y[n-4]=0.3x[n]+0.6x[n-1]+0.6x[n-2]

Solution

```
b=[1,0.5]
a=[1, -1.8*cos(pi/16),0.81]
zplane(b,a) ;
```

```
b=[0.3, 0.6 , 0.6]
a=[1,0.1,0,0,0.1,0.1]
zplane(b,a) ;
```

Transformée en Z

Les fonctions de transfert sont décrites par :

$$Y(z) = H(z)X(z) = \frac{B(z)}{A(z)}X(z)$$

où

$$B(z) = b(1) + b(2)z^{-1} + \cdots + b(M)z^{-(M-1)}$$

$$A(z) = a(1) + a(2)z^{-1} + \cdots + a(N)z^{-(N-1)}$$

$X(z)$ et $Y(z)$ sont les transformée en Z de l'entrée et de la sortie $x(n)$ et respectif $y(n)$:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n}$$

$$Y(z) = \sum_{n=-\infty}^{\infty} y(n)z^{-n}$$

En MATLAB le filtre $H(z)$ est représenté par des vecteurs formés par les coefficients polynomiaux de $B(z)$ et $A(z)$.

Un filtre numérique est un système discret utilisé dans le but de modifier l'amplitude et/ou la phase d'un signal. Les systèmes utilisés sont linéaires et invariants en temps. La réponse dans le domaine temps d'un SDLIT est donnée par le produit de convolution entre le signal d'entrée x[n] et la réponse impulsionelle notée h[n]

$$y[n]=x[n]*h[n]=\sum h[k]x[n-k]$$

Par l'utilisation de la transormée Z nous avons :

Y(z)=H(z)X(z)

Où H(z) est la fonction de transfert du SDLIT. Pour avoir des filtres réalisables physiquement il faut imposer la causalité et la stabilité du système.

Le système décrit par des équations aux différences finies

$$\sum_{k=0}^{N} a_k y[n-k]=\sum_{k=0}^{M} b_k x[n-k]$$

nous permet de définir deux types de filtres :
Filtres à réponse impulsionnelle finie RFI
Filtres à réponse impulsionnelle infinie RII

Le **filtrage** des signaux est un très vaste domaine. On trouvera une série de filtres prêts à l'emploi dans divers toolbox.

Citons les fonctions:
`conv`
- convolution
`filter`
- 1D Digital filter
`filter2`
- 2D Digital filter
`latcfilt`
- Lattice filter
`fir1, fir2`
...-Window based FIR filter design (Finite Impulse Response)
medfilt1
- 1D median filtering
besself
- Bessel analog filter

Exemple 1: Utilisation simple de `filter`
:
filtrage d'un signal bruyant avec une fonction carrée constante:

```
% TEST_FILTER.M test de la fonction filter
% Filtrage d'un signal bruyant avec une fonction
% carrée de largeur 2*m + 1 .
%
clear
figure(1)
clf
subplot(2,1,1)
N = 1001;
bruit = 0.1
t = [0 : N-1]'/(N-1);
Smooth = exp( -100*(t-1/5).^2) + exp( -500*(t-2/5).^2) + ...
exp(-2500*(t-3/5).^2) + exp(-12500*(t-4/5).^2) ;
Noisy = Smooth + bruit* rand(size(t)); % avec bruit
plot(t, Noisy)
hold on
title('Filtre fenêtre constante')
m = 5; % Demi largeur de la fenêtre de filtrage
b = ones(2*m+1, 1) ./(2*m+1) ;
Filtered = filter(b, 1, Noisy);
% compensation du retard introduit par le filtre:
Filtered(1:m-1) = Noisy(1:m-1);
Filtered(m:N-m) = Filtered(m+m:N);
Filtered(N-m+1:N) = Noisy(N-m+1:N);
plot(t, Filtered+0.2,'-r')
subplot(2,1,2)
plot(t, Smooth-Filtered)
title('Smooth-Filtered')
```

Filtres RFI avec la phase linéaire

Représenter graphiquement la fonction de transfert du filtre RFI défini par :

$$H(z)=z^{-2}+\sqrt{3}z^{-3}+z^{-4}$$

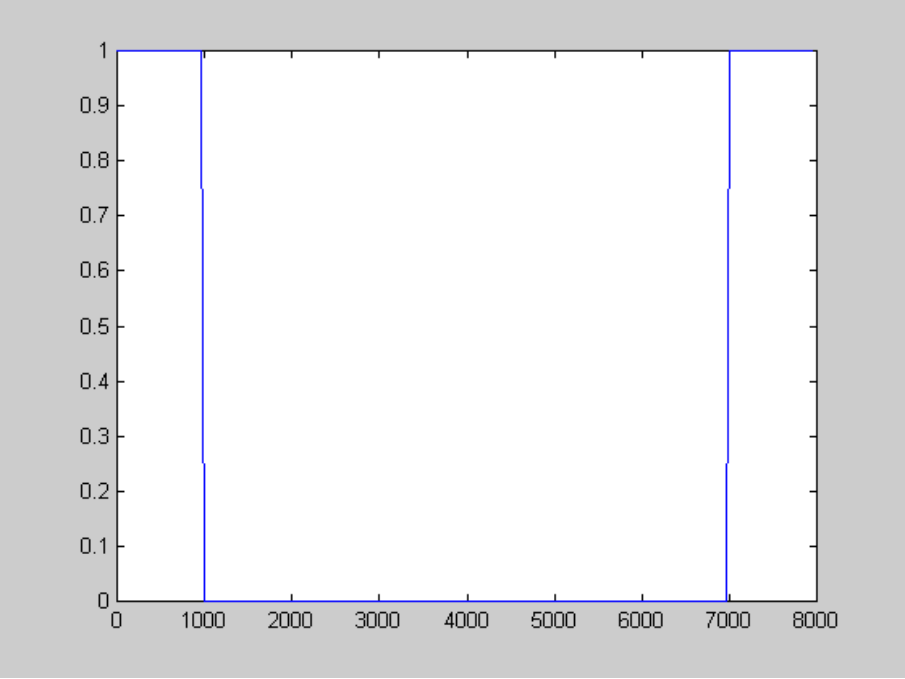
Solution

```
H=[0,0,1,sqrt(3),1,0,0]
H=fft(h,512) ;
W=-pi :2*pi/512 :pi-2*pi/512 ;
plot(w,fftshift(abs(H))),grid
plot(w,fftshift(angle(H))),grid
```

Filtre rectangulaire

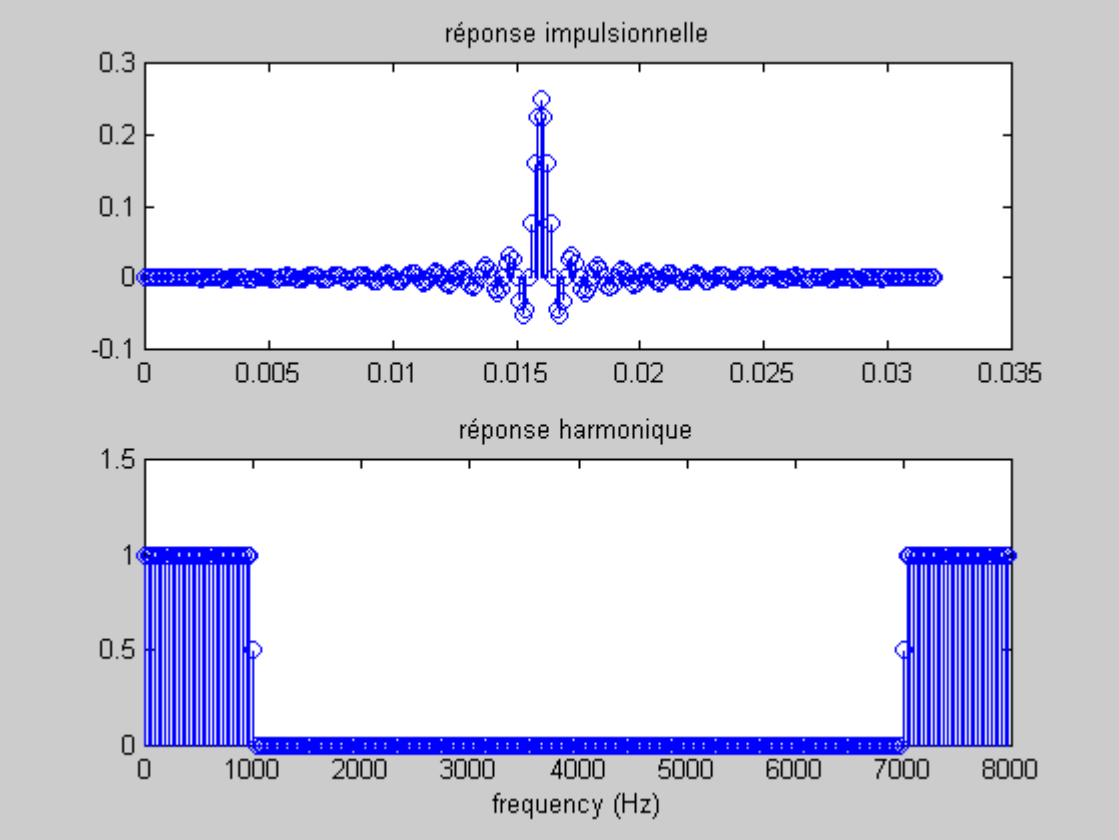
1. On demande de créer un filtre rectangulaire passe bas FIR de longueur $R = 256$ filtrant les fréquences au delà de $f_c = 1kHz$ à appliquer au son numérique du fichier **data.wav**. Calculer avant tout le nombre de points N donnant la largeur de la réponse harmonique.

```
fe = 8000;
R = 256;
N = (R/8000)*1000;
H = [ones(1,N) , zeros(1,R-2*N) , ones(1,N)];
freq = [0:fe/R:fe-fe/R];
plot(freq,H);%le filtre (réponse harmonique)
```



2. Caractériser ce filtre à l'aide de ses réponses harmonique et impulsionnelle.

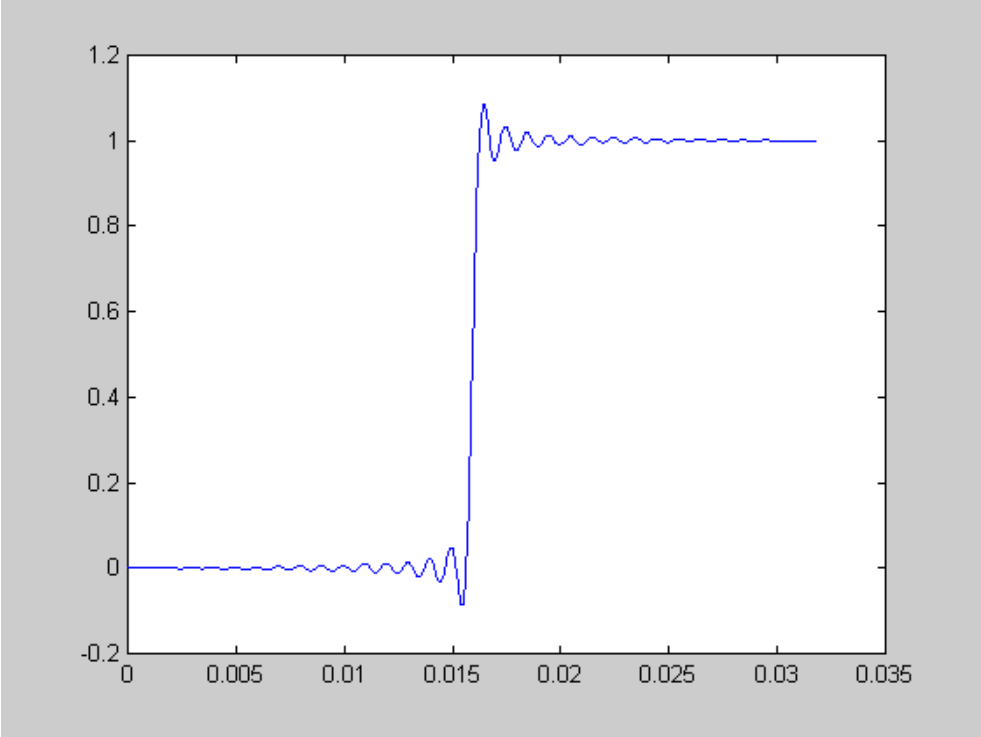
```
freq=[0:R-1]*fe/R;
N= fix(1000*R/fe); % si fe vaut 11 KHz alors N=23
% définition RH du filtre
H=[ones(1,N),zeros(1,R-2*N),ones(1,N)];
h=fftshift(real(ifft(H)));
subplot(2,1,1)
stem([0:R-1]/fe,h)
title('réponse harmonique');
subplot(2,1,2)
% on trace RH sans bode(.)
stem(freq,abs(fft(h)))
title('réponse impulsionnelle');
xlabel('frequency (Hz)')
```



On observe que ce filtre est bien un FIR
et que le gain statique est 1 cf limite quand f tend vers 0 (cf réponse impulsionnelle)

vérification de la stabilité par la réponse indicielle

```
ind = conv(h,ones(1,R));
figure;
plot(temps,ind(1:R));
```

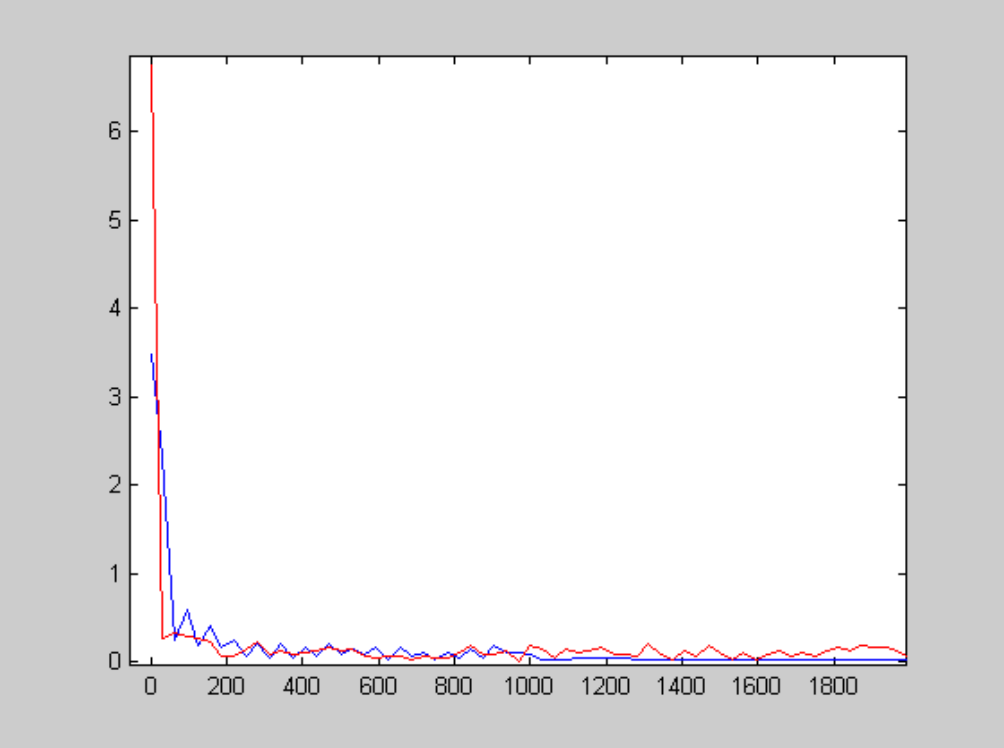


Le filtre présente des rebonds pour tendre vers la stabilité. Il s'apparente a un filtre passe bas dont la fréquence de coupure serrait 1000Hz.

3. Evaluer l'effet du filtre sur **data.wav** sur le spectre de ce son, et utilisant MATLAB.

On aura pris soin de reéchantillonner le signal data a 8000Hz

```
[son,fe] = A partir du fichier indiqué
son  = son(:,1)';
son_f = filter(h,1,son);
plot(freq,abs(fft(son_f,R)));%le son filtré en bleu
hold on;
plot(freq,abs(fft(son,R)),'r');%le son original en rouge
```



On le voit sur le graphique ci-dessus (au dela de 1000Hz le signal bleu est presque nul).

Exemple :

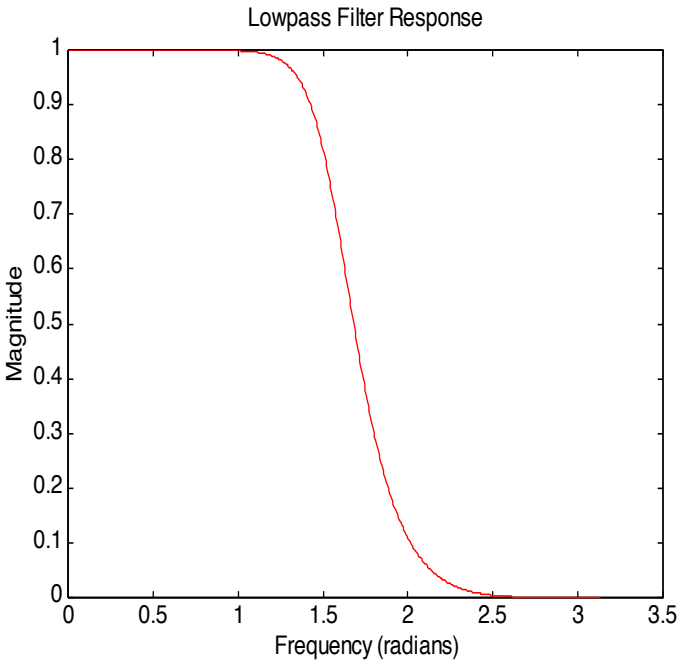
```
[b,a] = butter(5,.5)

b =
    0.0528    0.2639    0.5279    0.5279    0.2639    0.0528
a =
    1.0000    0.0000    0.6334    0.0000    0.0557    0.0000
```

b contient les coefficients du numérateur,
a contient les coefficients du dénominateur

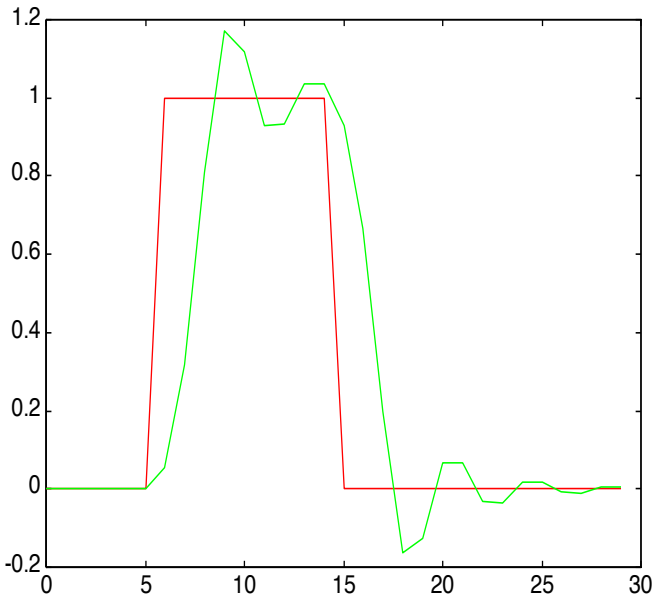
La transformée en Z du filtre peut être calculée autour du cercle $\left(z = e^{j\omega}, \omega \in [0,\pi]\right)$ avec la fonction **freqz**

```
[H,w] = freqz(b,a);
plot(w,abs(H))
ylabel('Magnitude')
xlabel('Frequency (radians)')
title('Lowpass Filter Response')
```



Exemple :
Considérons le filtre **b**,**a** définie plus bas et en entrée on considère un signal carré :

```
n = 30;              % longueur du signal
t = (0:n-1)';        % index vecteur
x = (t>5)&(t<15);    % pulse   t = 6,7,8,... 14
y = filter(h,1,x);
clf, plot(t,x,t,y)
```



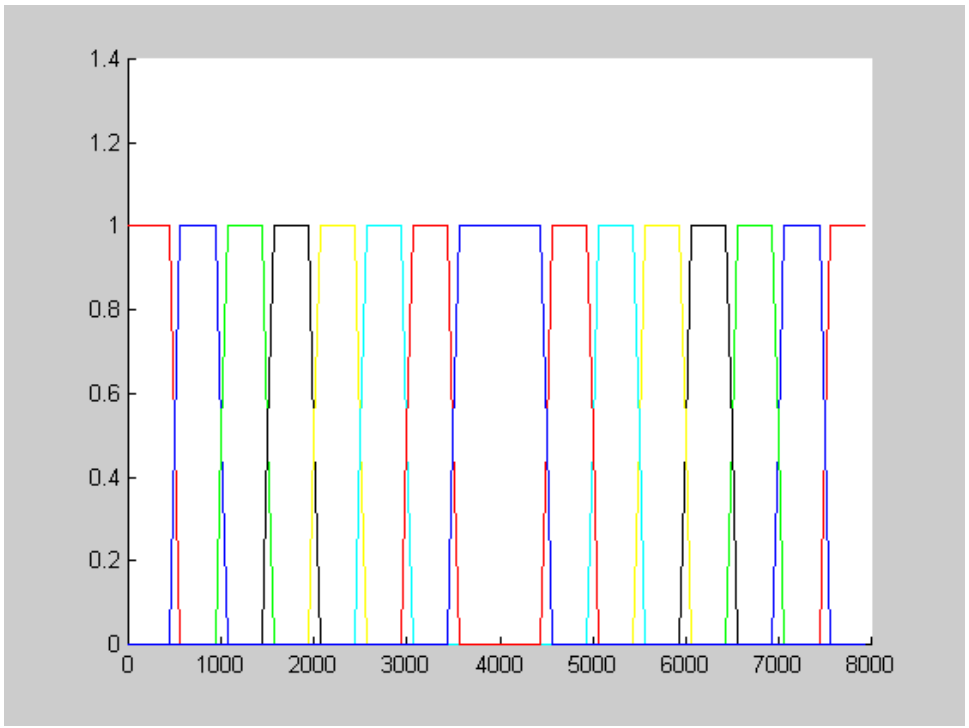
Banc de filtres

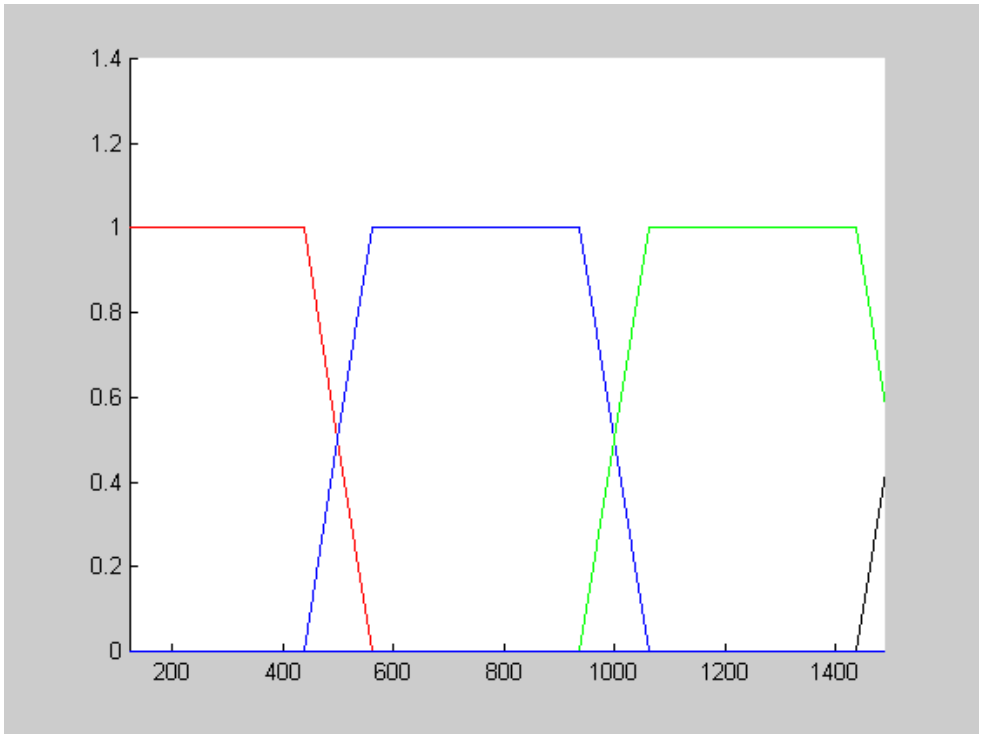
1. Calculer un banc de 8 filtres en adaptant les scripts donnés en cours avec $R = 128$ et $f_e = 8kHz$ pour illustrer).

```
% 2.Banc de filtres
close all;
R= 128;
M= 8; % on veut 8 filtres
fe= 8000;
%la largeur d'un filtre est fe/(2*M)
N= R/(4*M);
n=0:R-1;
H=[ones(1,N),zeros(1,R-2*N),ones(1,N)];
h=fftshift(real(ifft(H)));
for j=0:M-1% réponses impulsionnelles des filtres
 bande(j+1,:)=2*cos((2*j+1)*n*pi/(2*M)).*h;
end
```

Caractériser le résultat en traçant la réponse harmonique du banc de filtres. Evaluer à l'aide du zoom le recouvrement entre deux filtres voisins sur l'axe des fréquences.

```
freq=n*(fe/R);
coul=['r','b','g','k','y','c','r','b'];
hold on
for j=0:M-1 % calcul et tracé des réponses harmoniques
   plot(freq,abs(fft(bande(j+1,:))),coul(j+1)) end
```


1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
1000
2000
3000
4000
5000
6000
7000
8000


1.4
1.2
1
0.8
0.6
0.4
0.2
0
200
400
600
800
1000
1200
1400

ANNEXE

Variables, espace de travail et fonctions

Les variables sous MATLAB

Les noms de variables vérifient les trois principes suivants :

b) ils comportent au maximum 31 caractères alphanumériques
c) le premier caractère est obligatoirement une lettre
d) Matlab fait la différence entre majuscules et minuscules

Il est inutile de :

a) typer les variables
b) dimensionner les variables

Le journal

Toutes les commandes tapées au niveau du prompt MATLAB peuvent être archivées dans un fichier avec la commande diary. Le fichier résultat est un fichier ASCII lisible par n'importe quel éditeur.

```
>> diary journal.txt
>> commande 1
….
>>diary off

type journal
```

Rappel des commandes

Pour rappeler des commandes déjà utilisées :

a) première possibilité : les touches ↑ , ↓ et les commandes s'affichent en ordre chronologique inverse
b) tapez des chaînes de caractères contenues dans les commandes que vous voulez répéter et les touches ↑ , ↓

Fonction Matlab d'intérêt général :

help
who
whos
format
clear
lookfor

dir
cd

Variables et constantes

ans
pi
Inf
NaN
i ou j

Vecteurs et Matrices

linspace
eye
zeros
rand(n)
tril
triu
det
inv

Instruction de control logique

If
End

If
Elseif
End

For
End

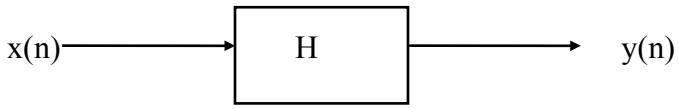
While
End

Opérateurs
<
<=
>
>=
==
~=

Opérateurs logiques
&
|
~

APPLICATION

La deconvolution est l'opération de reconstruction de l'entrée à partir de la fonction de transfert et la sortie



Problème : Connaissant y(n) et H, trouver x(n).

Exemple de problème de deconvolution
A partir des moyennes de température hebdomadaire trouver les températures journalières ?

Notice that **y** is a 'smoothed' version of **x**.

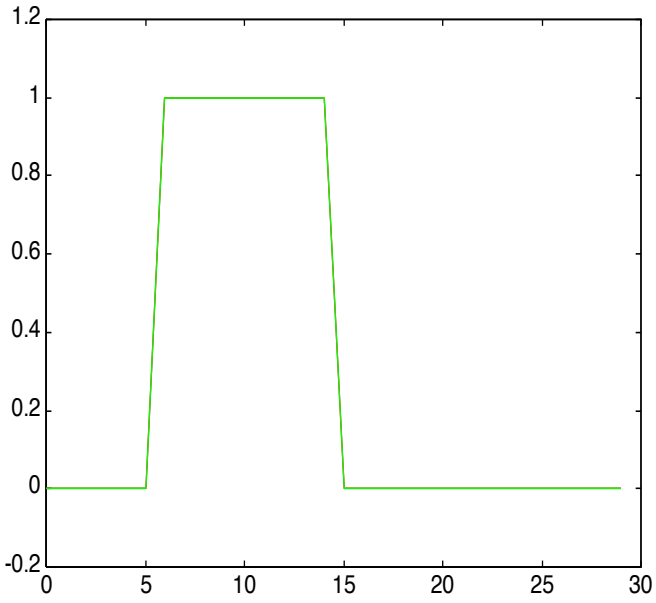
For a length **n** signal, this filter's convolution matrix is

```
C = convmtx(h,n);
C = C(1:n,:);      % keep only first n rows (truncates output)
```

A simple solution is to invert the matrix to recover the input **x** from the output **y**. We use **** to get the least squares, or pseudo inverse, solution.

```
x1 = C\y;
plot(t,x,t,x1)
```

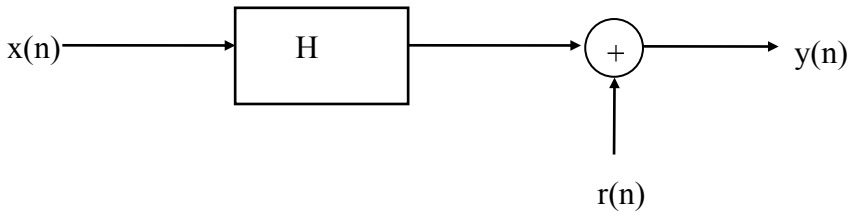
(This is equivalent to `x1 = pinv(C)*y;`)



The input has been recovered EXACTLY now (provided the convolution matrix may be inverted).

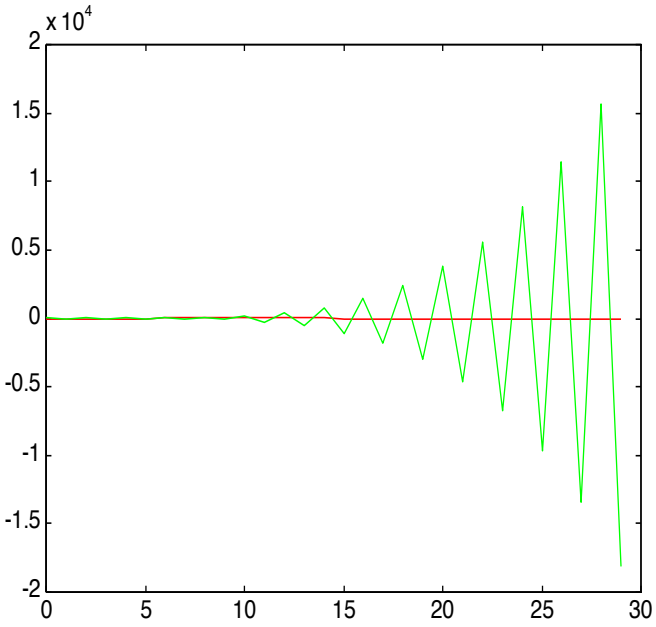
DECONVOLUTION WITH ADDITIVE NOISE

Now consider the case where there is some noise added to the output of the system prior to the measurements.



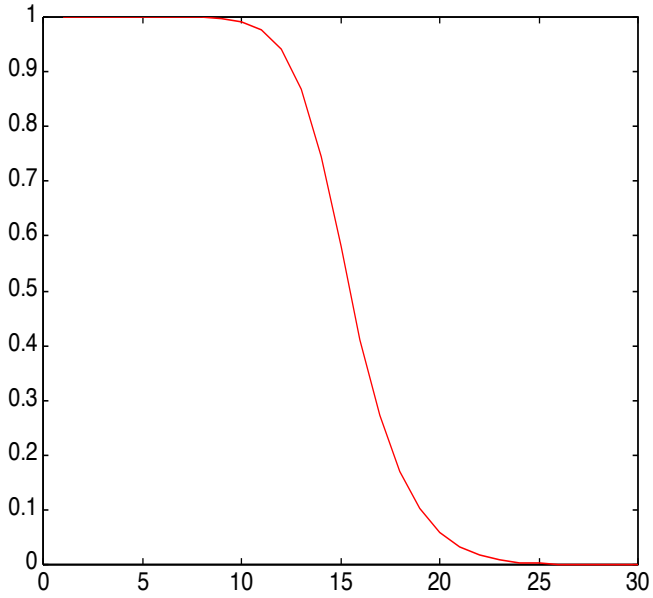
Now the inversion process suffers from 'noise gain'. For example, add some white noise to the output prior to inverting:

```
r = randn(n,1)/100;
x2 = pinv(C)*(y+r);
plot(t,x,t,x2)
```



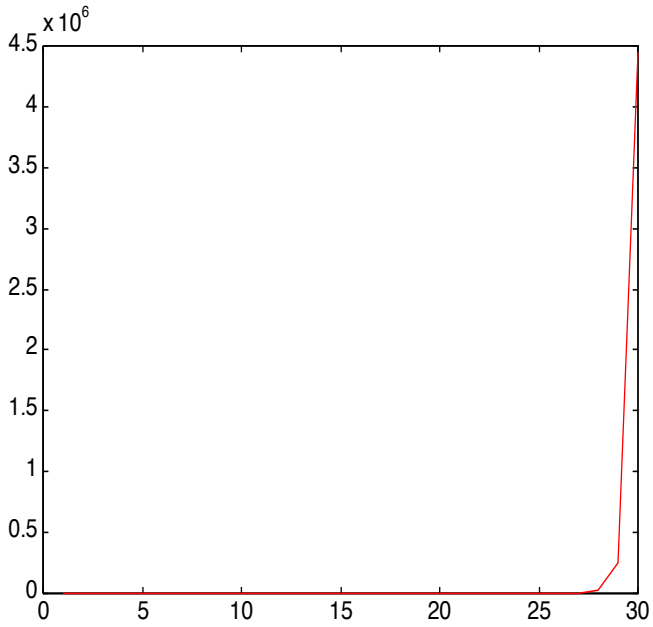
The deconvolved input has gone berserk! Why is this? We only added a bit of noise. The reason is because some of the singular values of our matrix are very very small:

```
s = svd(C);
clf, plot(s)
```



When we invert the output **`y+r`**, we multiply its coordinates in the basis of the singular vectors of **`C`** by the inverse of the singular values.

```
clf, plot(1./s)
```



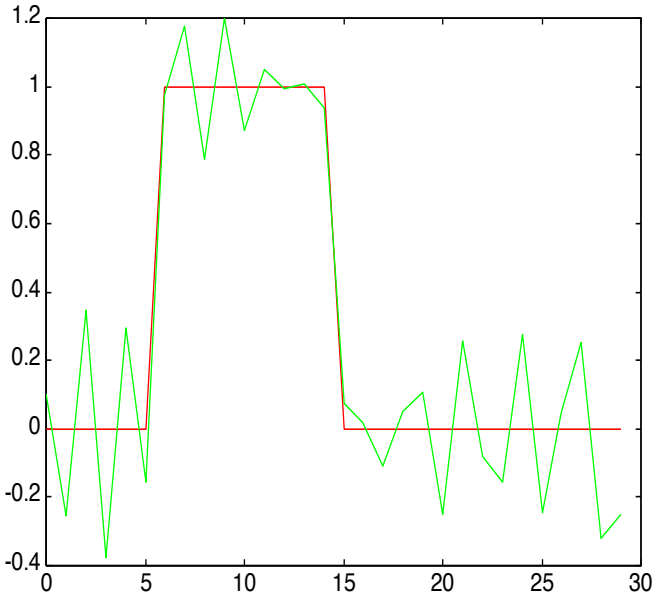
The last 5 or so singular values, when inverted, lead to a HUGE gain in the signal components in the span of those last singular vectors.

TRADING OFF NOISE GAIN FOR RESOLUTION

What can we do about noise gain? We can truncate the inversion to exclude the last singular values that are tiny.

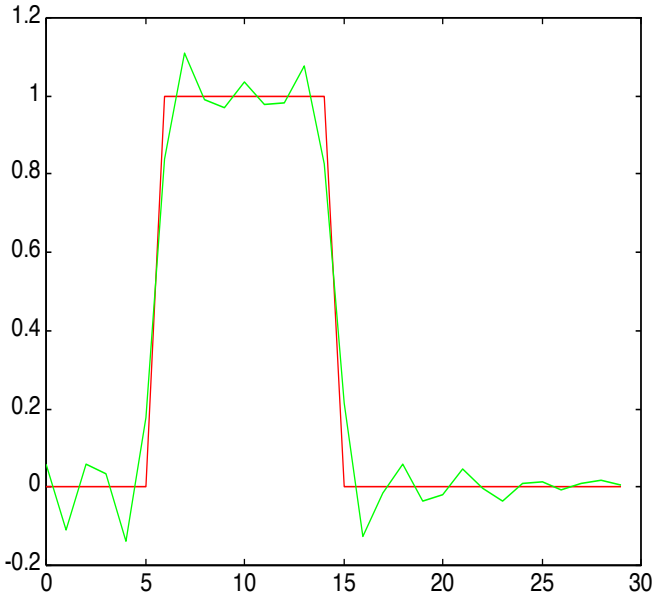
This can be done in the pinv function by passing a tolerance. The pseudo inverse will treat all singular values less than the tolerance as zero.

```
x2 = pinv(C,s(n-5))*(y+r);
plot(t,x,t,x2)
```



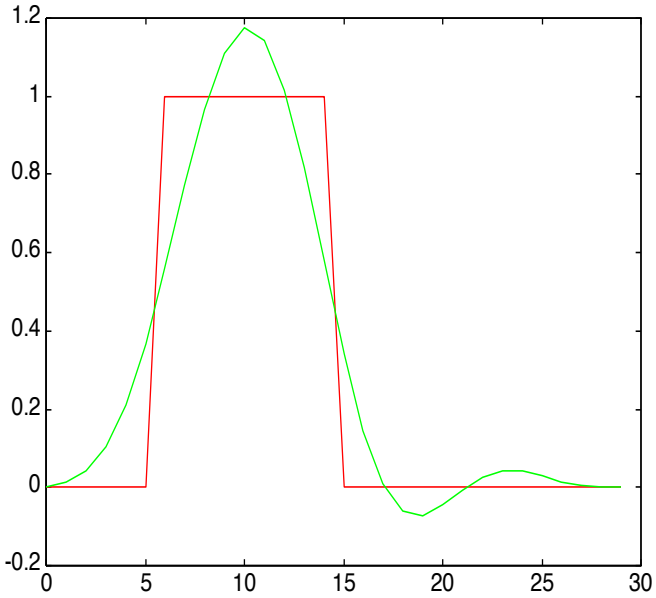
Here we've truncated the last five singular vectors. We still see a lot of ringing, but the edges of the pulse are very exact.

```
x2 = pinv(C,s(n-10))*(y+r);
plot(t,x,t,x2)
```



Taking away the last 10 singular vectors, the input pulse is very closely recovered. The ringing is reduced, and the edges of the pulse are easy to distinguish.

```
x2 = pinv(C,s(5))*(y+r);
plot(t,x,t,x2)
```



This time, taking away all but 5 singular values, we see very little noise effects but the pulse is smeared. We have lost resolution but have also reduced noise gain.

TRY ME

Try this out with other filters, signal lengths, signals, and number of singular values retained.