

Initiation à Matlab :

Langage Matriciel

Dernière mise à jour : 13 janvier 2000

[Hervé CARFANTAN](#)

Ce document a été rédigé par [Hervé Carfantan](#) et il est sa propriété. Il ne devrait pas être reproduit ou distribué sans sa permission écrite.

Informations sur Matlab :

Matlab et ses boîtes à outils *signal processing* et *image processing* sont des produits de [The MathWorks, Inc.](#). De nombreuses informations sont disponibles sur leur site Web <http://www.mathworks.com> en particulier les [contributions d'autres utilisateurs](#). Ainsi que des [références de livres](#) reliés à matlab. Le site de [Mathtools](#): <http://www.mathtools.net> répertorie la plupart des boîtes à outils compatibles Matlab (gratuites et payantes). De nombreux documents d'initiation à Matlab sont disponibles sur internet, par exemple le [Matlab-Primer](#).

Je vous propose également un document résumé Matlab en bref (tout matlab sur une page A4) et un document résumé de Matlab pour le traitement du signal (sur une page A4).

Avertissement

Ce document a pour but de vous initier à Matlab, et de vous montrer l'intérêt et la facilité de son utilisation pour le traitement du signal. Il ne prétend pas présenter toutes les fonctionnalités de Matlab et ne peut en aucun cas remplacer sa documentation . Je vous encourage vivement à consulter cette dernière soit en ligne à partir de Matlab soit, par exemple au [Math Dept, University of Florida](#).

Il a été rédigé originellement à l'attention des étudiants du DEA Automatique et Traitement du Signal de l'université Paris-Sud, aussi ne convient-il pas forcément à ce que vous recherchez. D'autre part, les spécificités de la Version 5 de Matlab (notion d'objet) ne sont pas prise en compte ici et la plupart des instructions fonctionnent également pour les versions précédentes (3 et 4).

Table des matières

[1. Matlab](#)

[1.1. Généralités](#)

- [1.1.1. Introduction](#)
- [1.1.2. Lancer Matlab](#)
- [1.1.3. Langage interprété](#)
- [1.1.4. Variables](#)
- [1.1.5. Variables complexes](#)
- [1.1.6. Vecteurs, matrices et leur manipulation](#)
- [1.1.7. L'opérateur ":"](#)
- [1.1.8. Chaînes de caractères](#)

[1.2. Opérations matricielles](#)[1.3. Affichages graphiques et alphanumériques](#)[1.3.1. Affichage alphanumérique](#)[1.3.2. Affichages graphiques de courbes 1D](#)[1.3.3. Affichages graphiques de courbes 2D](#)[1.3.4. Affichage de plusieurs courbes](#)[1.4. Environnement de travail, scripts et fonctions](#)[1.4.1. Répertoire de travail](#)[1.4.2. Sauvegarde et chargement de variables](#)[1.4.3. Scripts](#)[1.4.4. Fonctions](#)[1.4.5. Outils de mise au point : "déboggage"](#)[1.5. Boucles et contrôles](#)[1.5.1. Contrôle : "if"](#)[1.5.2. Boucle : "for"](#)[1.6. Help / Aide en ligne](#)[2. Matlab pour le traitement du signal](#)[2.1. Fonctions de la version de base de Matlab](#)[2.1.1 Construction de matrices particulières](#)[2.1.2. Fonctions mathématiques particulières](#)[2.1.3. Décomposition de matrices](#)[2.1.4. Fonctions de filtrage et de convolution](#)[2.1.5. Transformation de Fourier](#)[2.1.6. Génération de nombres pseudo-aléatoires](#)[2.2. Signal processing toolbox](#)[2.3. Image processing toolbox](#)[3. Travaux dirigés - travaux pratiques](#)[Exercice 1 : Sur la transformée de Fourier discrète](#)[Exercice 2 : Convolution](#)[Exercice 3 : Régularisation](#)[Exercice 4 : Décomposition en ondelettes discrètes orthogonales](#)

1. Matlab

1.1. Généralités

1.1.1. Introduction

Matlab (abréviation de " **Matrix Laboratory** "), est un environnement informatique conçu pour le calcul matriciel. L'élément de base est une matrice dont la dimension n'a pas à être fixée. Matlab est un outil puissant qui permet la résolution de nombreux problèmes en beaucoup moins de temps qu'il n'en faudrait pour les formuler en C ou en Pascal.

S'il est parfaitement adapté à l'Automatique et au Traitement du Signal, sa facilité d'emploi avec des nombres complexes et ses possibilités d'affichages graphiques en font un outil intéressant pour bien d'autres types

d'applications. De plus, des " toolboxes " (boîtes à outils) sont disponibles dans de nombreux domaines (traitement du signal, traitement d'image, optimisation, contrôle ...).

Matlab peut être considéré comme un langage de programmation au même titre que C, Pascal ou Basic. C'est un langage interprété, c'est-à-dire que les instructions sont exécutées immédiatement après avoir été tapées.

1.1.2. Lancer Matlab

Pour lancer Matlab sur Macintosh, il suffit de cliquer sur Matlab dans le " *menu pomme* " ou de cliquer deux fois sur un fichier écrit sous Matlab. La fenêtre " *command* " apparaît alors, c'est dans cette fenêtre que l'on peut taper les instructions Matlab (à la suite des chevrons `>>`).

Sur un PC suivant que l'on a une version DOS ou Windows de Matlab, on tapera simplement `matlab` à l'invite du DOS ou on cliquera sur l'icône Matlab, sous Unix, il suffira de taper la commande `matlab` dans un terminal.

1.1.3. Langage interprété

Matlab est un langage interprété. Il n'est pas nécessaire de compiler un programme avant de l'exécuter et toute commande tapée dans la fenêtre de commande est immédiatement exécutée (après la frappe de `return`).

```
>> 2+2
ans = 4
>> 5^2
ans = 25
```

La réponse est affichée et stockée dans la variable `ans`.

La plupart des fonctions mathématiques usuelles sont définies dans Matlab, et ceci sous une forme naturelle (`sin`, `cos`, `exp`, ...). de même que certaines constantes (`pi` ...).

```
>> 2*sin(pi/4)
ans = 1.4142
```

1.1.4. Variables

On peut indiquer le nom de la variable dans laquelle le résultat doit être stocké (commence par une lettre, moins de 19 caractères).

Attention, Matlab prend en considération les majuscules (`x` est différent de `X`).

```
>> x = pi/4
x = 0.7854
```

Le nom de cette variable ainsi que le résultat sont affichés.

Un point virgule à la fin de la ligne permet de ne pas afficher ce résultat. On peut taper plusieurs commandes par ligne, séparées par un point virgule.

```
>> x = pi/2; y = sin(x);
```

Lorsqu'une ligne de commande est trop longue on peut l'interrompre par trois points (...) et la poursuivre à la ligne suivante, on peut aussi mettre des commentaires dans une ligne de commande à l'aide du signe "%".

1.1.5. Variables complexes

Matlab travaille indifféremment avec des nombres réels et complexes. Par défaut les variables *i* et *j* sont initialisées à la valeur complexe . Naturellement si vous redéfinissez la variable *i* ou *j* avec une autre valeur elle n'aura plus la même signification.

```
>> z = 3 + 2*i  
z = 3.0000 + 2.0000i
```

Les fonctions usuelles de manipulation des nombres complexes sont prédéfinies dans Matlab : *real*, *imag*, *abs*, *angle* (en radian), *conj*.

```
>> r = abs(z);  
>> theta = angle(z);  
>> y = r*exp(i*theta);
```

1.1.6. Vecteurs, matrices et leur manipulation

En fait, toute variable de Matlab est une matrice (scalaire : matrice 1x1, vecteur : matrice 1xN ou N×1). On peut spécifier directement une matrice sous la forme d'un tableau, l'espace ou la virgule sépare deux éléments d'une même ligne, les points virgules séparent les éléments de lignes distinctes.

```
>> A = [ 1, 2, 3 ; 4, 5, 6 ; 7, 8, 9 ]  
A = 1 2 3  
     4 5 6  
     7 8 9
```

Les éléments d'une matrice peuvent être n'importe quelle expression de Matlab :

```
>> x = [ -1.3, sqrt(3), (1+2+3)*4/5 ]  
x = -1.3000 1.7321 4.8000
```

Les éléments d'une matrice peuvent être référencés par leurs indices :

```
>>x(2)  
ans = 1.7321
```

```
>>x(5) = abs(x(1))

x = -1.3000 1.7321 4.8000 0.0000 1.3000
```

On peut remarquer que la taille du vecteur x a été ajustée en remplaçant les éléments non précisés par 0.

On peut aussi créer des matrices avec les fonctions zeros, ones et eye, ces fonctions créent des matrices de la taille précisée, respectivement remplies de zéros , de un, et de un sur la diagonale et des zéros ailleurs (eye = prononciation anglaise de I comme identité).

```
>> eye(2,3)

ans = 1 0 0

0 1 0

>> ones(1,5)

ans = 1 1 1 1 1
```

On peut avoir des informations sur la taille d'une matrice :

```
>> size(x)

ans = 1 5

>> length(x)

ans = 5
```

On peut ajouter des lignes et des colonnes à des matrices déjà existantes.

```
>> r1 = [10, 11, 12];

>> A = [A ; r1]

A = 1 2 3

4 5 6
7 8 9
10 11 12

>> r2 = zeros(4,1);

>> A = [A, r2]

A = 1 2 3 0

4 5 6 0
7 8 9 0
10 11 12 0
```

On peut avoir le transposé ou le transposé conjugué d'une matrice :

```
>> A' % Transposée conjuguée de A

>> A.' % Transposée de A
```

Mais aussi des retournements horizontaux (`flipud` : Up/Down), verticaux (`fliplr` : Left/Right), ou des rotations de matrices :

```
>> flipud(A) % Retournement vertical de A
>> fliplr(A) % Retournement horizontal de A
>> rot90(A) % Rotation de A de 90 degrés
```

On peut également remettre en forme des matrices et vecteurs sur un nombre de lignes et de colonnes donnés (à condition que le nombre total d'éléments corresponde) :

```
>> B = [ 1 2 3 4 5 6 ];
>> B = reshape(B,2,3) % Remise en forme avec 2 lignes et % 3 colonnes
B = 1 3 5
     2 4 6
```

1.1.7. L'opérateur ":"

L'opérateur ":" , sous Matlab, peut être considéré comme l'opérateur d'énumération. Sa syntaxe usuelle est :

```
deb:pas:fin
```

Il construit un vecteur dont le premier élément est *deb* puis *deb+pas*, *deb+2*pas*... jusqu'à *deb+n*pas* tel que $deb+n*pas \leq fin < deb+(n+1)*pas$.

```
>> x = 0.5:0.1:0.85
x = 0.5000 0.6000 0.7000 0.8000
```

Le pas d'incrémentation peut être omis, 1 est alors pris par défaut :

```
>> x =1:5
x = 1 2 3 4 5
```

On peut aussi utiliser le ":" pour sélectionner des éléments d'un vecteur ou d'une matrice :

```
>> A(1,3) % Troisième élément de la première ligne de A
>> A(1,1:3) % Premier, deuxième et troisième éléments de % la première ligne de A
>> A(1,:) % Tous les éléments de la première ligne
>> A(:,3) % Tous les éléments de la troisième colonne
>> A(:) % Vecteur colonne contenant tous les éléments % de A lus colonne par colonne.
```

Si l'on a :

```
>> A = [ 1, 2, 3 ; 4, 5, 6 ; 7, 8, 9 ];
>> indi = [3, 2, 1];
```

```
>> indj = [1, 3];
```

Alors

```
>> A(indi,indj)
```

```
ans = 7 9
```

```
4 6  
1 3
```

nous donnera les premiers et troisièmes éléments des lignes 3, 2 et 1 de *A*.

Mais il faut cependant remarquer que si l'on a :

```
>> indi = [3, 0, 2];
```

alors

```
>> A(indi,:)
```

```
ans = 1 2 3
```

```
7 8 9
```

nous donne les lignes de *A* dont l'élément de *indi* est non nul (c'est-à-dire ici la première et la troisième ligne).

En effet, lorsqu'un vecteur contenant un zéro est pris comme indice, les éléments du vecteur sont considérés comme des booléens, leur valeur n'est alors plus prise en compte, mais uniquement le fait qu'ils soient nuls ou non.

1.1.8. Chaînes de caractères

Comme on l'a vu précédemment, toute variable de Matlab est une matrice. Cependant, il peut être utile de conserver des chaînes de caractères dans des variables, et cela se fait de façon tout à fait naturelle :

```
>> message = 'bienvenue sur Matlab';
```

On peut alors réaliser des manipulations de même type que pour les vecteurs

```
>> message = [message, ' version 4.1'];
```

```
message = bienvenue sur Matlab version 4.1
```

On peut convertir des nombres en chaîne de caractères à l'aide des fonctions `num2str`, `int2str`, `sprintf`.

Par exemple :

```
>> message = ['pi vaut ', num2str(pi)]
```

```
message = pi vaut 3.142
```

Deux fonctions peuvent être très utiles lors de manipulations de chaînes de caractères : `eval` et `feval`. Si l'on possède une instruction Matlab dans une chaîne de caractères, la commande `eval` évaluera cette chaîne de caractères comme si elle avait été tapée à la ligne de commande de Matlab :

```
>> str = ['x', n, ' = ' pi;'];
```

```
>> eval([ nom_var, '1 = sqrt(-1)' ])
x1 = 0.0000 + 1.0000i
```

De même si l'on possède le nom d'une fonction dans une chaîne de caractères, la fonction `feval` nous permet d'évaluer cette fonction pour certains de ces arguments :

```
>> nom_fonction = 'sin';
>> feval(nom_fonction,pi)
ans = 1.2246e-16 % et oui sin(pi) ne vaut pas zéro !
```

1.2. Opérations matricielles

Les opérations usuelles sont définies de façon naturelle pour les matrices :

```
>> 2*A % Produit par un scalaire
>> A*B % Produit de deux matrices (de dimensions % cohérentes)
>> A^p % Elève la matrice carrée A à la puissance p
>> inv(A) % Inversion d'une matrice carrée inversible (message %
d'alerte éventuel)
>> A.*B % Produit élément par élément de deux matrices
```

Attention : A^*A est différent de $A.^*A$.

```
>> X = A\B % Donne la solution de  $A^*X = B$ 
>> X = B/A % Donne la solution de  $X^*A = B$ 
>> X = A./B % Division éléments par éléments
```

La plupart des fonctions mathématiques n'ayant pas de signification particulière pour des matrices les considèrent comme un simple tableau de valeur, par exemple :

```
>> exp(A)
```

renverra une matrice dans laquelle tous les éléments seront l'exponentielle des éléments de A , de même pour `log`, `sqrt`, `round`, `rem` (correspond au modulo), `sign`. Pour calculer l'exponentielle matricielle, le logarithme matriciel ou la racine carrée d'une matrice on utilisera respectivement `expm`, `logm` et `sqrtm`.

Certaines fonctions de Matlab agissent indépendamment sur chaque colonnes de la matrice. C'est le cas de la fonction `max` (et `min`) :

```
>> max(A)
```

renverra un vecteur ligne dont chaque élément sera le maximum (minimum) de la colonne correspondante. C'est aussi le cas des fonctions `sum`, `prod` et `median` qui donnent respectivement les sommes, produit, et la valeur médiane des colonnes de la matrice. Les fonctions `cumsum`, `cumprod` et `sort` renvoient une matrice dont chaque colonne contient les sommes et produits cumulés et un tri des colonnes de la matrice initiale.

D'autres fonctions sont très utiles sur les matrices :

```
>> poly(A) % Polynôme caractéristique de A
```

```
>> det(A) % Déterminant de A
>> trace(A) % Trace de A
>> [V, D] = eig(A) % Renvoie les valeurs propres et les vecteurs %
propres de A (eigenvalues, eigenvectors)
```

Remarques sur les matrices creuses : Un grand nombre de fonctions Matlab s'appliquent directement à des matrices creuses (c'est-à-dire dont la majeure partie des éléments sont nuls). On peut créer des matrices creuses grâce à la fonction `sparse`. On ne conservera ainsi en mémoire que les éléments non nuls de la matrice, d'où le gain de place mémoire. De même lorsqu'on utilisera ces matrices lors de calculs, Matlab tiendra compte du fait qu'elles soient creuses pour réduire le coût de calcul. Par exemple :

```
>> A = eye(2);
>> B = sparse(A)
B =
(1,1) 1
(2,2) 1
```

On voit bien ici que Matlab ne conserve dans la matrice B que les éléments non nuls (ici les éléments diagonaux) de la matrice A. Ainsi, les deux instructions suivantes donneront le même résultat mais la première calculera quatre multiplications alors que la deuxième n'en calculera que deux car la matrice B est creuse.

```
>> A*[ 1; 1];
>> B*[ 1; 1];
```

1.3. Affichages graphiques et alphanumériques

1.3.1. Affichage alphanumérique

On peut afficher des chaînes de caractères dans la fenêtre de commande :

```
>> disp(message)
pi vaut 3.142
```

Les fonctions `sprintf` et `fprintf` existe également (même syntaxe qu'en langage C).

```
>> fprintf('pi vaut %f\n',pi)
pi vaut 3.141593
```

On peut aussi demander des valeurs à l'utilisateur :

```
>> rep = input('Nombre d''itération de l''algorithme : ');
```

Matlab affichera la chaîne de caractère entrée en paramètre et attendra une réponse de l'utilisateur.

1.3.2. Affichages graphiques de courbes 1D

Matlab permet un grand nombre de types d'affichage 1D et 2D, seuls les plus courant seront décrits ici.

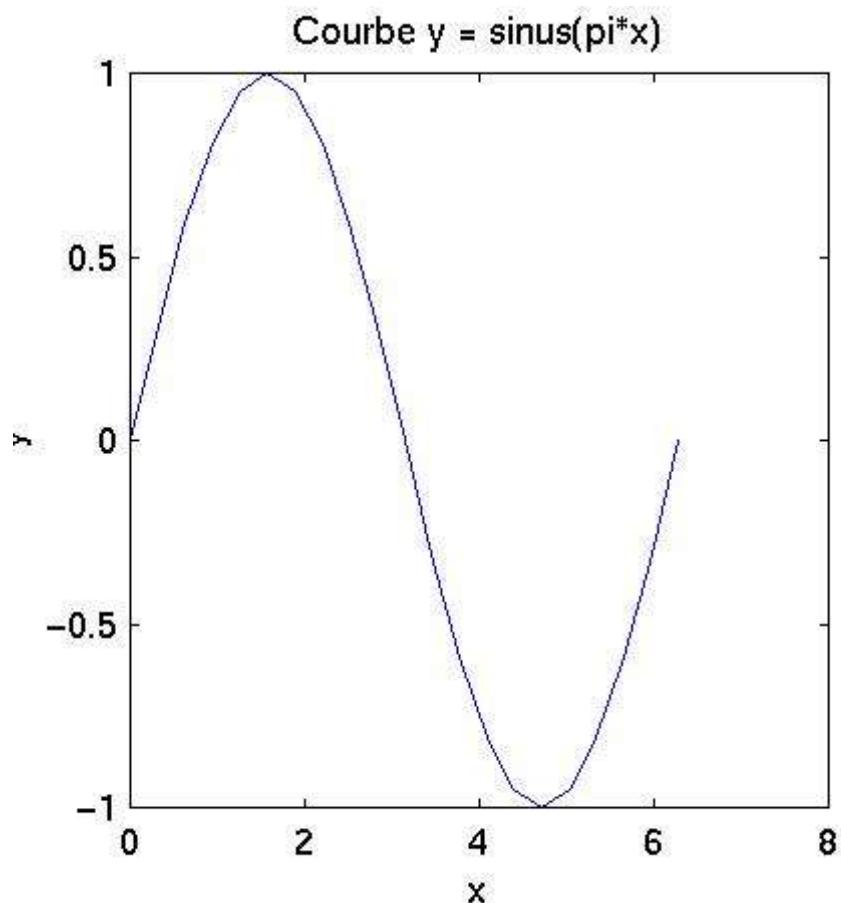
La commande `plot` permet l'affichage d'une courbe 1D :

```
>> x = 0:0.1:2; y = sin(x*pi);
>> plot(x*pi,y) % plot(abscisse,ordonnée)
```

On aurait pu tracer la courbe en semilog ou en log avec les fonctions `semilogx`, `semilogy` et `loglog`.

On peut ajouter un titre aux figures ainsi que des labels aux axes avec les commandes `title`, `xlabel`, `ylabel`:

```
>> title('Courbe y = sinus(pi*x)')
>> xlabel('x'); ylabel('y')
```



1.3.3. Affichages graphiques de courbes 2D

Avec la commande `mesh` on peut aisément avoir une représentation d'une fonction 2D. Cependant il faut construire la grille des coordonnées des points en lesquels on a les valeurs de la fonction à l'aide de la fonction `meshgrid`.

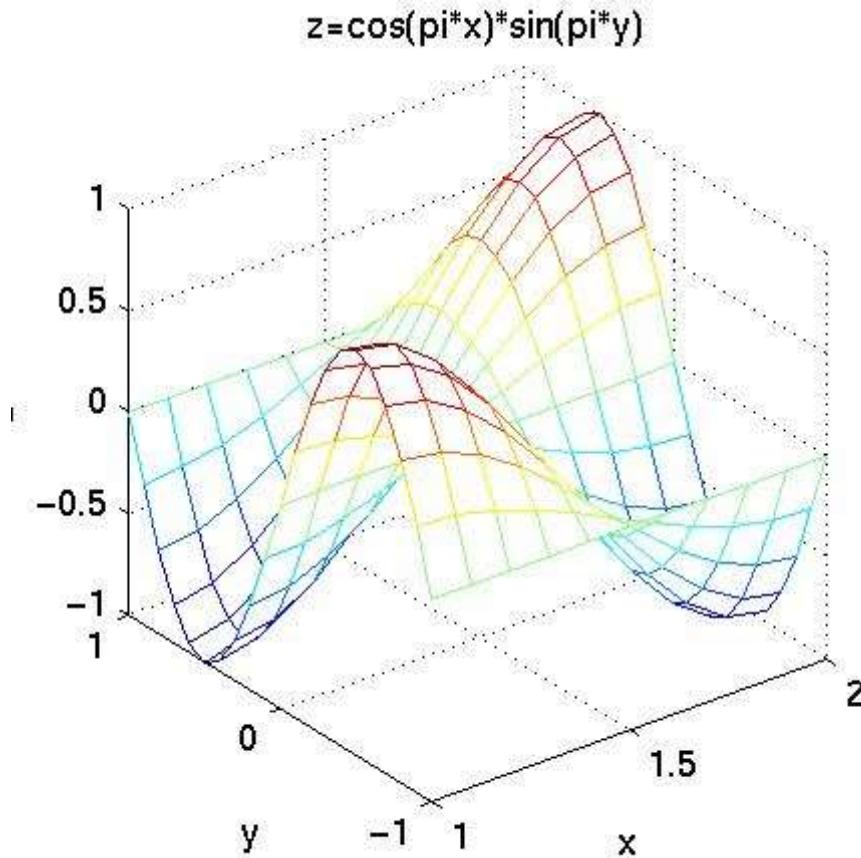
```
>> x = 1:0.1:2; y = -1:0.1:1;
>> [X,Y] = meshgrid(x,y);
```

```
>> mesh(X,Y,cos(pi*X).*sin(pi*Y)) % Coordonnées en x et en y et %
valeurs de la fonction : Z

>> xlabel('x'); ylabel('y'); zlabel('z');

>> title('z=cos(pi*x)*sin(pi*y)')
```

Mais on peut aussi visualiser les matrices avec les fonctions `image`, `meshc`, `contour`...



1.3.4. Affichage de plusieurs courbes

On peut bien évidemment vouloir afficher plusieurs courbes à l'écran. Pour cela deux solutions s'offrent à nous :

On peut effectuer plusieurs affichages sur une même figure en utilisant la commande `subplot` qui subdivise la fenêtre graphique. Sa syntaxe est :

```
subplot(nombre_lignes, nombre_colonnes, numéro_subdivision)
```

Les subdivisions sont numérotées de 1 à `nombre_lignes*nombre_colonnes`, de la gauche vers la droite puis de haut en bas.

Exemple :

```
>> subplot(3,2,1)
>> plot(x,y)
>> subplot(3,2,2)
>> plot(x,y.^2)
```

1	2
3	4
5	6

On peut aussi ouvrir une deuxième fenêtre graphique à l'aide de la commande `figure`.

Le passage d'une fenêtre graphique à une autre pourra alors se faire à la souris ou en précisant le numéro correspondant dans la commande `figure (n)`.

1.4. Environnement de travail, scripts et fonctions

1.4.1. Répertoire de travail

Il est indispensable de travailler dans votre propre répertoire et non dans le répertoire de Matlab. De même dans votre répertoire vous pouvez organiser vos fichiers dans des sous répertoires.

Pour que vos fonctions et scripts soient accessibles à partir de la ligne de commande Matlab il faut au début de chaque session Matlab, déclarer vos répertoires de travail. Ceci se fait à l'aide de la commande `path` :

```
path(path, 'Mac 1:Projet1:Matlab') % Pour les Macs
path(path, 'c:\projet1\matlab') % Pour les PCs
```

Ceci rajoute aux chemins déjà existants (que l'on obtient à l'aide de `path`) le chemin mis entre quotes.

Il faut aussi se placer dans votre répertoire de travail. On peut se déplacer dans l'arborescence du disque de travail à l'aide des commandes `cd` et `dir` (même syntaxe pour les Macs et sous MS-Dos).

En fait lorsqu'on travaille souvent sous matlab avec des fichiers de fonctions et des scripts organisés dans une arborescence sur le disque de travail, il est possible et préférable de réaliser un fichier `startup.m` qui sera exécuté immédiatement en lançant Matlab. Dans ce fichier on peut définir le path, charger d'éventuelles variables, se placer dans le répertoire désiré...

1.4.2. Sauvegarde et chargement de variables

Il peut être utile, lors d'une session de travail, de sauvegarder des variables dans des fichiers du répertoire de travail. Cela peut être réalisé à l'aide de la fonction `save` dont la syntaxe est :

```
save nom_fichier nom_variables
```

exemple :

```
>> save test.mat A, x, y
```

Si le nom des variables est omis, tous l'espace de travail est sauvé; si l'extension du fichier est omise elle sera automatiquement `.mat`, si de plus, le nom du fichier est omis, la sauvegarde se fera dans le fichier `matlab.mat`.

Pour recharger des variables sauveées dans un fichier, il suffit de taper :

```
load nom_fichier
```

Si le nom du fichier est omis, Matlab chargera le fichier *matlab.mat*.

On peut à un moment donné vouloir connaître toutes les variables disponibles dans l'espace de travail. Cela est possible grâce aux commandes `who` et `whos`. La commande `who` nous donne le nom de toutes les variables existantes, tandis que `whos` nous donne leur nom et l'espace mémoire qu'elles occupent.

Lorsque l'on travaille avec des matrices de tailles importantes, ou lorsque la puissance de la machine est limitée, il peut être prudent d'utiliser parfois la commande `pack`. Cette commande consolide l'espace mémoire, qui bien souvent est fragmenté après un certain temps de travail sous Matlab. Ainsi le temps d'allocation et de recherche en mémoire sera réduit.

1.4.3. Scripts

Il est parfois (souvent) souhaitable, pour ne pas avoir à taper plusieurs fois une même séquence d'instructions, de la stocker dans un fichier. Ainsi on pourra réutiliser cette séquence dans une autre session de travail. Un tel fichier est dénommé script.

Sur Macintosh ou sous windows, il suffit d'ouvrir un fichier avec le menu *file, new*, de taper la séquence de commande et de sauver le fichier avec une extension " *.m* " (*nom_fich.m*). En tapant le nom du fichier sous Matlab, la suite d'instructions s'exécute.

Les variables définies dans l'espace de travail sont accessibles pour le script. De même les variables définies (ou modifiées) dans le script sont accessibles dans l'espace de travail.

On peut (doit) mettre des commentaires à l'aide du caractère pour-cent " % ". Toute commande située après " % " n'est pas prise en compte par Matlab, jusqu'à la ligne suivante. De plus Matlab ne voit pas les espaces blancs (lorsqu'il n'y a pas d'ambiguïté), ce qui permet de bien indenter ses fichiers.

1.4.4. Fonctions

On a parfois (souvent) besoin de fonctions qui ne sont pas fournies par Matlab. On peut alors créer de telles fonctions dans un fichier séparé et les appeler de la même façon que les fonctions préexistantes.

La première ligne (hormis les lignes de commentaires) d'une fonction doit impérativement avoir la syntaxe suivante :

```
function [ var de sorties, ... ] = nom_fonction( var d'entrée, ... )
```

Exemple de fonction :

```
function y = sinuscardinal(x)
z = sin(x); % Variable de stockage
y = z./x; % Résultat de la fonction % Il faudra se méfier de la
           % division pour x=0
```

Cette fonction pourra être appelée par :

```
>> sincpi = sinuscardinal(pi);
```

Exemple de fonction à plusieurs variables de sortie :

```
function [mini, maxi] = minetmax(x)
    mini = min(x); % Première variable de sortie
    maxi = max(x); % Deuxième variable de sortie
```

Cette fonction pourra être appelée par :

```
>> [miny, maxy] = minetmax(y);
```

Le nom de la fonction doit impérativement être le même que le nom du fichier dans lequel elle est stockée (sinon Matlab ne prendra pas en compte ce nom mais uniquement celui du fichier).

Les nombres d'arguments en entrée et en sortie ne sont pas fixes et peuvent être récupérés par nargin et nargout. Cela nous permet donc de faire des fonctions Matlab pouvant dépendre d'un nombre variable de paramètres.

Les variables de l'espace de travail ne sont pas accessibles à la fonction sauf si elles sont entrées comme variable d'entrée. De même les variables définies dans une fonction ne sont pas accessibles dans l'espace de travail.

1.4.5. Outils de mise au point : " déboggage "

Lorsqu'on programme dans un langage informatique, il est utile de posséder des outils de mise au point. Matlab étant un langage interprété la mise au point de fonctions peut-être effectuée directement sur la ligne de commande. Cependant Matlab possède des possibilités de " déboggage " utiles pour la réalisation de programmes plus élaborés.

Le plus simple (et existant déjà dans les versions précédentes) est de placer dans la fonction ou le script à mettre au point la commande keyboard qui l'interrompt lors de son exécution et permet de vérifier les valeurs et dimensions des variables ou d'exécuter toute autre commande. Le retour à l'exécution normale est alors prévue avec la commande return.

Des outils plus élaborés sont disponibles depuis la version 4, mais ne peuvent être utilisées que pour mettre au point des fonctions. On peut ainsi, à partir de la ligne de commande, placer des points d'arrêt dans une fonction à l'aide de :

```
dbstop at num_ligne in nom_fich % Arrêt à une ligne précise
dbstop in nom_fich % Arrêt avant que la fonction % s'exécute
```

Pour poursuivre l'exécution de la fonction on pourra utiliser dbstep qui avancera pas à pas ou dbcont qui continue l'exécution jusqu'au prochain point d'arrêt rencontré. dbclear permet d'enlever des points d'arrêt, et dbquit de quitter le mode " déboggage ". Des possibilités supplémentaires sont offertes grâce aux commandes dbstatus, dbstack, dbtype, dbup, dbdown (voir la documentation ou le help pour plus de détails).

1.5. Boucles et contrôles

Comme de nombreux autres langages de programmation, Matlab possède trois types d'instructions de contrôles et de boucles : for, if et while. Nous n'étudierons ici que if et for.

1.5.1 Contrôle : "if"

Cette structure de contrôle est très utilisée pour effectuer des branchements, dans un script ou une fonction, suivant certaines conditions. Sa syntaxe est :

```
if (expression logique)

    suite d'instructions 1;

else

    suite d'instructions 2;

end
```

La condition est exprimée au travers d'une expression logique. Les expressions logiques sont des expressions quelconques pour lesquelles on considère uniquement le fait qu'elles soient nulles (expression fausse) ou non (expression vraie).

On peut aussi construire ces expressions à l'aide des opérateurs logiques et (`&`), ou (`|`), égal (`==`), supérieur (`>,>=`), inférieur (`<,<=`), non (`~`), ou exclusif (`xor`) ou à l'aide de fonctions logiques prédéfinies `exist`, `any`, `find`, `isinf`, `isnan`...

1.5.2 Boucle : "for"

La boucle `for` a pour syntaxe :

```
for i=1:10

    suite d'instructions;

end
```

Cette structure de contrôle est très utile, mais il est recommandé d'éviter au maximum les boucles dans Matlab, car elles sont très lentes par rapport à un traitement matriciel.

Exemple trivial : la boucle suivante

```
for i=1:N, x(i) = i; end
```

est équivalente à

```
x = 1:N;
```

Malheureusement, on ne rencontre pas toujours des cas aussi simples et il est difficile de penser matriciel, c'est à dire penser en Matlab.

Les fonctions `zeros` et `ones`, permettent parfois d'éviter de mettre en oeuvre une boucle.

Par exemple, au lieu de faire la boucle :

```
r = 1:10;

A = []; % initialisation de A à vide

for i=1:5, A = [A ; r]; end
```

On peut écrire

```
r = 1:10;
A = ones(5,1)*r;
```

Une façon plus rusée encore de réaliser ceci est :

```
r = 1:10;
A = r(ones(5,1),:); % Etudiez bien cette instruction
```

On remarque qu'ainsi on évite la multiplication de l'instruction précédente.

De plus certaines fonctions Matlab agissent sur une matrice en tant que suite de vecteurs.

Ainsi

```
maximum = max(A);
```

nous donne le même résultat (mais en moins de temps) que :

```
n = length(A(1,:));
for i=1:n, maximum(i) = max(A(:,i)); end
```

Il faut donc bien réfléchir avant de construire une boucle, afin de savoir si on ne peut vraiment pas faire autrement.

Cependant, il est parfois impossible d'éviter la mise en oeuvre d'une boucle, c'est en général le cas pour les algorithmes itératifs.

Dans ce cas il est nécessaire de bien initialiser les variables que l'on va allouer lors de l'exécution de la boucle. Ainsi si l'on ne pouvait éviter d'effectuer la boucle suivante,

```
for i=1:N, x(i) = i; end
```

il ne faudrait pas l'initialiser par

```
x = [];
```

mais par

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

car l'allocation en mémoire de la variable *x* est réalisée une fois pour toute à l'initialisation et non à chaque itération ou l'on a à augmenter sa dimension.

1.6. Help / Aide en ligne

Matlab est pourvu d'une fonction d'aide très utile : `help`. Ainsi si vous tapez `help` sur la ligne de commande apparaissent tous les répertoires accessibles depuis Matlab ainsi qu'une explication concernant ces répertoires. Si vous tapez `help` suivi du nom d'un répertoire, Matlab affiche une explication brève sur toutes les fonctions et scripts de ce répertoire. Enfin si vous tapez `help` suivi du nom d'un fichier (script ou fonction), apparaît une explication détaillée sur l'utilisation de la fonction ou du script.

De plus vous pouvez créer une aide sur les fonctions et scripts que vous créez et sur vos répertoires.

Pour une fonction, Matlab prendra comme help toutes les lignes de commentaires suivant l'instruction `function` (en première ligne). Pour un script Matlab prendra toutes les lignes de commentaires placés en début de fichier. Pour un répertoire, il faut créer un fichier `Contents.m` dans lequel on place (en commentaires) en première ligne l'aide général sur le répertoire, ensuite les aides sur les différentes fonctions et scripts contenus dans le répertoire.

Ces aides que vous créez pour vos fonctions et scripts, peuvent évidemment être intéressantes pour un utilisateur éventuel, mais peuvent aussi vous être utiles, pour vous rappeler le but de telle fonction ou pour vous souvenir de sa syntaxe.

Exemple simple

```
function [mini, maxi] = minetmax(x)
%
% [mini, maxi] = minetmax(x)
%
% Cette fonction renvoie les éléments minimum et maximum d'un
%
% vecteur
%
% Réalisée le 16 juillet 1997 : H. Carfantan
```

Une autre fonction d'aide très utile est `lookfor`. Si vous ne vous souvenez plus du nom d'une fonction qui réalise une certaine action, par exemple l'inversion d'une matrice, tapez :

```
>> lookfor inverse
INV Matrix inverse.
PINV Pseudoinverse.
IFFT Inverse discrete Fourier transform.
```

et Matlab affiche le nom des fonctions (ainsi qu'une brève description) contenant en première ligne de leur aide le mot `inverse`.

Remarques Importantes :

Il est vivement recommandé, avant d'écrire une fonction, de consulter la documentation ou le `help` afin de savoir s'il n'existe pas déjà une fonction similaire (et sûrement mieux écrite).

De même, avant d'utiliser une fonction matlab pour la première fois, consulter la documentation ou le `help` pour vérifier qu'elle réalise bien ce que vous désirez.

L'aide en ligne est également disponible au [Math Dept, University of Florida](#)

2. Matlab pour le traitement du signal

Comme précisé en introduction, Matlab est un langage parfaitement adapté au traitement du signal, et particulièrement à la recherche. En effet, il existe un grand nombre de fonctions Matlab utiles au traitement du signal, ce qui nous évite d'avoir à les programmer et nous permet de nous focaliser sur nos propres problèmes. De plus il existe des bibliothèques supplémentaires de fonctions appelées *toolboxes* qui viennent enrichir la version de base : les toolboxes " Signal Processing " et " Image Processing " (voir leur documentation).

2.1. Fonctions de la version de base de Matlab

Dès sa version de base, Matlab propose des fonctions très utiles en traitement du signal.

2.1.1. Construction de matrices particulières

Les matrices de Toeplitz peuvent être construites à l'aide de la fonction *toeplitz* :

```
>> c = [ 1 2 3 4 5 ];
>> l = [ 1.5 2.5 3.5 4.5 5.5 ];
>> toeplitz(c,l)
ans = 1.0000 2.5000 3.5000 4.5000 5.5000
      2.0000 1.0000 2.5000 3.5000 4.5000
      3.0000 2.0000 1.0000 2.5000 3.5000
      4.0000 3.0000 2.0000 1.0000 2.5000
      5.0000 4.0000 3.0000 2.0000 1.0000
```

On remarque que les colonnes remportent le conflit sur la diagonale.

Les matrices de Vandermonde peuvent être construites à l'aide de la fonction *vander* :

```
>> x = 2:5;
>> vander(x)
ans = 8 4 2 1
      27 9 3 1
      64 16 4 1
      25 25 5 1
```

On peut aussi construire des matrices de Hankel, Hadamard, Hilbert ...

2.1.2. Fonctions mathématiques particulières

En plus des fonctions mathématiques usuelles, Matlab offre des fonctions plus spécialisées dont il est bon de connaître l'existence : *bessel* (fonction de bessel de première et deuxième sorte), *beta* (fonction beta), *gamma* (fonction gamma).

2.1.3. Décomposition de matrices

Certaines décompositions de matrices sont déjà implémentées :

- La décomposition en valeurs singulières est réalisée par la fonction *svd* (singular value decomposition). Sa syntaxe est :

```
>> [U, S, V] = svd(X);
```

Elle produit une matrice diagonale S de même dimension que X , avec les éléments diagonaux non nuls rangés en ordre décroissant, et des matrices unitaires U et V telles que $X = U*S*V'$.

- La factorisation de Cholesky est réalisée par la fonction *chol* :

```
>> R = chol(X);
```

Elle utilise uniquement la partie triangulaire supérieure de X (et considère que la partie triangulaire inférieure de X est la transposée conjuguée de la précédente). Elle produit un message d'erreur si X n'est pas définie positive, sinon elle produit la matrice triangulaire supérieure R telle que $R'*R=X$.

- La factorisation QR est réalisée par la fonction *qr* :

```
>> [Q, R] = qr(X);
```

produit une matrice triangulaire supérieure Q de même dimension que X et une matrice unitaire R telle que $X = Q*R$.

- Une factorisation LU (Lower, Upper) est réalisée par la fonction *lu* :

```
>> [L, U] = lu(X);
```

exprime une matrice carrée X sous forme d'une matrice triangulaire supérieure U et d'une permutation d'une matrice triangulaire inférieure L , telles que $X = L*U$.

2.1.4. Fonctions de filtrage et de convolution

Une fonction très utile en signal est la fonction *filter* :

```
>> Y = filter(B, A, X);
```

Elle construit le vecteur Y tel que :

$$y(n) = b(1)*x(n) + b(2)*x(n-1) + \dots + b(nb+1)*x(n-nb) - a(2)*y(n-1) - \dots - a(na+1)*y(n-na)$$

Cela correspond donc à un filtrage par un filtre ARMA.

La fonction *roots* qui calcule les zeros d'un polynôme, peut lui être associée afin de trouver les zéros et les pôles d'un ARMA ; Par exemple le polynôme $P(z)=z^3-6z^2-72z-27$ est représenté par le vecteur :

```
>> p=[1 -6 -72 -27];
```

dont les racines sont données par :

```
>> r=roots(p)
```

```
r = 12.1229 -5.7345 -0.3884
```

Pour calculer la valeur du polynôme en certain points on pourra utiliser la fonction *polyval* :

```
>> polyval(p, [ 1, exp(j*pi/4), exp(j*pi/2)])
```

```
ans = 1.0e+002 *
```

```
-1.0400 -0.7862 - 0.5620i -0.2100 - 0.7300i
```

La fonction *poly* effectue l'opération inverse de *roots* : étant donné un polynôme unitaire défini par ses racines, *poly* le développe en donnant les coefficient de ses monômes en puissance décroissante.

```
>> p=poly(r)
p = 1.000 -6.000 -72.000 -27.000
```

D'autre part, si l'argument de *poly* est une matrice, la fonction renvoie les coefficients du polynôme caractéristique de cette matrice.

La fonction *conv* effectue elle la convolution de deux vecteurs :

```
>> Z = conv(X, Y);
```

Cette convolution est complète dans le sens où la longueur de *Z* est la somme des longueurs de *X* et *Y* moins 1. Elle peut aussi être interprétée comme le développement du produit de deux polynômes.

En dimension 2 les fonctions correspondant à *filter* et *conv* sont *filter2* et *conv2*.

Une fonction fournie mais dont l'utilisation doit vous sembler délicate est la fonction *deconv*. Elle est sensée effectuer une déconvolution, ce qui peut sembler ambitieux (pour ceux qui ont suivi un cours sur les problèmes inverses). Néanmoins cette fonction est utile pour la division de polynômes.

2.1.5. Transformation de Fourier

La transformation de Fourier rapide (Fast Fourier Transform) est implémentée sous Matlab et s'appelle *fft*, de même on a la transformation inverse *ifft*; en dimension 2 on a *fft2* et *ifft2*. Matlab nous donne la FFT pour les fréquences normalisées sur [0, 1], pour les représenter sur [-0.5, 0.5] on utilisera la fonction *fftshift*.

2.1.6. Génération de nombres pseudo-aléatoires

Matlab possède deux instructions pour générer des nombres pseudo-aléatoires : *rand* et *randn*.

```
>> Y=rand(p,q);
```

génère une matrice de taille $p \times q$ contenant des nombres uniformément répartis sur l'intervalle [0,1]. De même *randn* génère une matrice de taille $p \times q$ contenant des nombres répartis selon une loi gaussienne de moyenne nulle et de variance unité. Ces générateurs de nombres pseudo-aléatoires sont toujours activés avec la même graine au démarrage de Matlab. On veillera à utiliser l'option *seed* si l'on veut, pour commencer, une autre séquence que celle offerte par défaut.

Matlab possède des fonctions permettant d'analyser des nombres pseudo-aléatoires. Ainsi les fonctions *mean*, *std*, et *cov*, fournissent des estimations de la moyenne, de l'écart-type et de la matrice de covariance de vecteurs et matrices correspondant à des réalisations de variables aléatoires.

2.2. Signal processing toolbox

voir la documentation...

2.3. Image processing toolbox

voir la documentation...

3. Travaux dirigés - travaux pratiques

Exercice 1 : Sur la transformée de Fourier discrète

Ce premier exercice a pour but de vous familiariser avec Matlab. Il est intentionnellement très simple mais néanmoins très instructif, et il vous est conseillé d'en profiter pour faire le lien avec vos connaissances théoriques.

- Créer un signal x de longueur $N=250$ correspondant à une sinusoïde pure de fréquence 0.1Hz échantillonnée à 1Hz. Tracer l'allure du signal.

Calculer et afficher le "spectre du périodogramme" de ce signal correspondant au carré du module de sa transformée de Fourier. Peut-on en déduire la nature du signal ?

Recommencer avec le même signal pour une longueur $N=256$. Observe-t'on le même résultat ? Pourquoi ?

- Prélever un point sur deux des 256 échantillons du signal précédent, puis itérer la procédure. Quel est l'effet de ce sous-échantillonnage sur le spectre du signal ? Jusqu'à quel point peut-on aller sans perdre d'information ? Remarquons que l'on peut visualiser le périodogramme sur un nombre constant de points (quel que soit le nombre d'échantillons du signal) en effectuant un "*bourrage de zéros*".
- On va simuler l'échantillonnage d'un signal continu. Pour cela on va utiliser le signal fourni dans le fichier "*signal.mat*". Visualisez le périodogramme de ce signal (sur 256 points). Prélever un échantillon sur deux de ce signal en laissant les autres échantillons à zéro (puis un sur trois ...). Quel est l'effet produit sur le spectre du signal ?

Exercice 2 : Convolution

Voici l'aide de la fonction conv, elle peut être utile pour cet exercice.

```
>> help conv
```

```
CONV Convolution and polynomial multiplication.
```

```
C = CONV(A, B) convolves vectors A and B. The resulting
vector is length LENGTH(A)+LENGTH(B)-1.
```

```
If A and B are vectors of polynomial coefficients, convolving
them is equivalent to multiplying the two polynomials.
```

```
See also XCORR, DECONV, CONV2.
```

- L'opérateur de convolution est un opérateur linéaire. La commande $y = \text{conv}(x, h)$ peut donc se mettre sous la forme $y = Ax$.

Créer une fonction Matlab permettant de construire la matrice A à partir de la réponse impulsionale h .

Quelles sont les différences entre les deux commandes, avantages et inconvénients.

- La convolution peut aussi se faire dans le domaine fréquentiel. Ecrire une fonction Matlab `convfreq` réalisant cette opération.
- La déconvolution s'apparente, dans le domaine fréquentiel, à une division. Réaliser une fonction permettant la déconvolution d'un signal $y = \text{conv}(x, h)$.

Ajouter du bruit à y : $yb = y + b$ et réaliser la déconvolution du signal yb .

On pourra prendre comme exemple simple (pour un SNR = 20 dB sur les amplitudes) :

```
x = 1:11;
h = [1, 2, 1];
```

Conclusions.

Exercice 3 : Régularisation.

Soit la matrice

```
A = [ 10, 7, 8, 7 ; 7, 5, 6, 5 ; 8, 6, 10, 9 ; 7, 5, 9, 10 ];
```

- Calculer son inverse A^{-1} . On remarque que A est inversible et que A^{-1} est "sympathique".

Prendre le vecteur $x = [1 1 1 1]'$;

Calculer $y = Ax$, puis $x0 = A^{-1}y$.

*On va maintenant légèrement bruiter les mesures y : $yb = y + 0.1 * [-1 -1 1 -1]$;*

Calculer $x1 = A^{-1}yb$.

Conclusions.

Votre conclusion est sans doute que ce problème est mal posé. On va essayer de le régulariser.

Créer un script permettant de régulariser ce problème des différentes façons suivantes :

- Moindres carrés à norme minimale (pseudo inverse),
- Régularisation en tronquant 1, 2 puis 3 valeurs singulières,
- Régularisation par une approche bayésienne :

On sait qu'une régularisation par une approche bayésienne peut parfois se ramener à la minimisation d'un critère de la forme :

$$\| y - Ax \|^2 + \lambda \| Dx \|^2.$$

Ecrire un script Matlab permettant cette régularisation pour $D = \text{Identité}$ et pour :

```
D = 1.0 -0.5 0.0 0.0
     -0.5 1.0 -0.5 0.0
      0.0 -0.5 1.0 -0.5
```

```
0.0 0.0 -0.5 1.0
```

- Comparer ces résultats pour différentes réalisations du bruit et différentes valeurs du paramètre de régularisation λ .

Exercice 4 : Décomposition en ondelettes discrètes orthogonales

On souhaite réaliser un programme effectuant la décomposition en ondelette orthogonale d'un signal 1D suivant l'algorithme de **Mallat**, souvent appelé " *Fast Wavelet Transform*" On choisira par exemple comme base d'ondelettes la base de **Haar**.

- Qu'elles sont les briques élémentaires d'un tel algorithme ? Comment va-t'on les implémenter en Matlab ?
- Comment ces briques élémentaires vont-elles s'imbriquer entre elles ? Comment va-t'on réaliser cela en Matlab.
- Ecrire un script Matlab réalisant une telle décomposition.
- Réalisez la décomposition dans une base d'ondelettes des deux signaux suivants :

```
x1 = [ zeros(1,16), ones(1,32), zeros(1,16) ]
```

```
x2 = [ zeros(1,12), ones(1,32), zeros(1,20) ]
```

Conclusions.

De même on souhaite resynthétiser un signal 1D décomposé dans une base d'ondelettes discrètes orthogonales.

- Mêmes questions que pour la décomposition.
- Comment vous y prendriez vous pour synthétiser une ondelette (par exemple l'ondelette correspondant à la base de **Haar**) sur 64 points ?

[Retour à ma page personnelle](#)