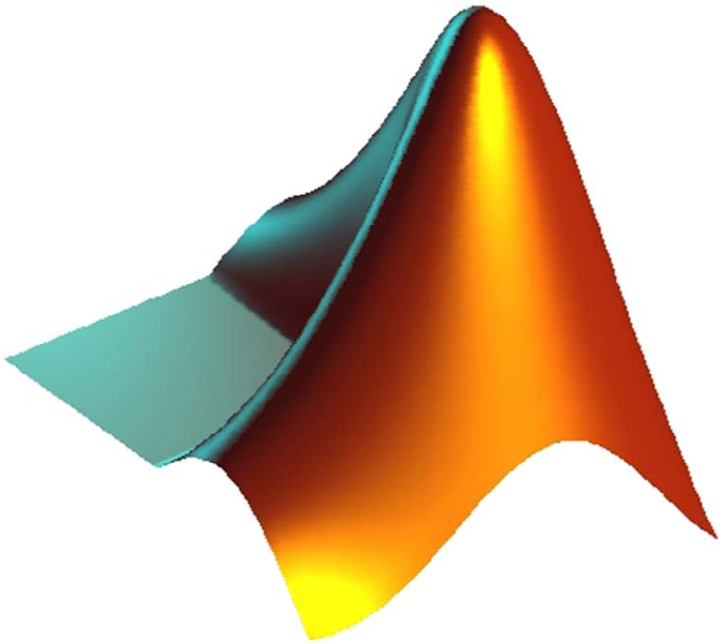
INITIATION AU

LOGICIEL MATLAB

version 6.5



ESIAL 1ère année 2004/2005

Marc Tomczak, Bruno Pinçon, Patrick Sibille UHP, Nancy 1

1. INTRODUCTION

MATLAB (*MATrix LABoratory*) est un logiciel interactif de calcul scientifique et technique. Il est issu des projets *Linpack* et *Eispack*, développés dans les années 70 et visant à constituer des librairies de programmes dédiés au calcul matriciel. Bien qu'écrit à l'origine en Fortran, par *Cleve Moler*, la version actuelle (due à *The MathWorks Inc.*) est en langage C. Elle est complétée par un outil de simulation (*SIMULINK*).
MATLAB est disponible sur plusieurs plates-formes : Sun, Bull, HP, IBM, compatibles PC, Macintosh, ainsi que plusieurs machines parallèles. Un de ses atouts est sa portabilité : la même portion de code peut être utilisée sur différentes plates-formes sans la moindre modification.
L'environnement MATLAB comprend un noyau de base (*built-in functions*) et des bibliothèques de fonctions spécifiques (boîtes à outils ou *toolboxes*), propres à un domaine, comme par exemple la *Signal Processing Toolbox*, dédiée au traitement du signal. Actuellement, plus d'une trentaine de boîtes à outils sont disponibles (commande des systèmes, identification, traitement d'images, réseaux de neurones, logique floue, calcul symbolique, mathématiques financières, statistiques, communications, etc.). L'édition du code source de toutes les fonctions est possible (sauf pour les *built-in functions*), l'utilisateur est donc libre de les modifier ou de s'en inspirer pour écrire ses propres fonctions.
Du point de vue des langages informatiques, MATLAB est un interpréteur, les instructions saisies dans la fenêtre de commande sont exécutées au fur et à mesure des appuis sur la touche *entrée* ou *return*. Cependant, elles peuvent aussi être regroupées dans un fichier programme, reconnaissable à l'extension **.m**. L'exécution de ce dernier peut alors être lancée en tapant son nom dans la fenêtre de commande, sous réserve que le chemin d'accès (le *path*) soit indiqué.
Le "langage" MATLAB contient un minimum de structures de programmation (structure itérative, structure conditionnelle, sous-routine) mais reste très rudimentaire. L'avantage est qu'il est très simple, assez intuitif et très rapide à programmer, offrant une grande tolérance (syntaxe simple, pas d'obligation de définition de types, etc.), ce qui permet un gain appréciable en temps de mise au point. L'ingénieur peut par ce moyen être plus efficace dans l'analyse d'un problème, en concentrant ses efforts sur celui-ci et non pas sur l'outil servant à le résoudre.
Enfin, l'interface graphique de MATLAB est sans conteste l'un des points forts du logiciel et facilite le tracé de courbes et l'obtention de graphiques 2D ou 3D de grande qualité.
Il faut tout de même noter qu'il existe des produits concurrents :

- MatrixX/SystemBuild, développé à partir de la même base par Integrated Systems Inc.
- Scilab/Scicos développé par l'INRIA en France, d'utilisation libre, qui peut être obtenu à l'adresse http://www-rocq.inria.fr/scilab
- Octave, développé par John W. Eaton et d'autres, d'utilisation libre, qui peut être téléchargé à l'adresse http://www.octave.org

Ce dernier est particulièrement compatible avec MATLAB, quant à Scilab, sa philosophie est très proche également.

2. DÉMARRAGE

Au démarrage, 3 fenêtres sont ouvertes :

- la **fenêtre de commande** (*command window*), qui comme son nom l'indique est la fenêtre où sont entrées les instructions ou commandes Matlab. Celle-ci est affectée de son *prompt* ">>", signe qui signifie que le logiciel est en attente d'une commande. Le contenu de cette fenêtre peut être effacé à l'aide de la commande `clc`.
- l'**historique** des commandes exécutées auparavant (*command history*). Il permet d'accéder à d'anciennes commandes. Il est également possible de naviguer parmi ces commandes à l'aide des touches ↑ et ↓ du clavier. Entrer une lettre puis taper sur la touche ↑ permet de rappeler les dernières commandes entrées commençant par cette lettre.
- l'**espace de travail** (*workspace*) qui contient l'ensemble des variables créées lors de la session (si elles n'ont pas été effacées entre-temps) et leur dimension. La gestion de l'espace de travail peut se faire soit par l'intermédiaire du menu ou du clic droit, soit par les commandes `who` et `whos` (affichage des variables existantes, de leur dimension, de la mémoire disponible), `clear` (réinitialisation de l'espace de travail : `clear all`, ou suppression d'une ou plusieurs variables : `clear` <*nom_variable*>), `load` (chargement d'un fichier de données), `save` (sauvegarde des variables de l'espace de travail dans des fichiers de données). Les noms des fichiers de données Matlab sont en général dotés de l'extension **.mat**.

Une quatrième fenêtre se cache derrière le *workspace*, il s'agit du **répertoire courant** (current directory) qui constitue le répertoire de travail où vont s'enregistrer les fichiers créés par l'utilisateur. Ce répertoire peut bien sûr être changé, mais pour qu'il soit reconnu, par exemple lors de l'appel d'un programme qui s'y trouve, le chemin d'accès doit être spécifié dans le *path*.

D'autres fenêtres peuvent apparaître en cours d'utilisation, il s'agit :

- de l'**éditeur** de Matlab, qui permet de taper des programmes ou des fonctions. Bien sûr, tout autre éditeur peut faire l'affaire, mais l'éditeur de Matlab est équipé d'un débogueur ce qui peut être utile.
- des **fenêtres graphiques**, en cas d'utilisation de commandes graphiques.
- de la fenêtre d'**aide**. Celle-ci peut être invoquée à partir du menu, ou directement à partir de la fenêtre de commande (`helpwin`) : >>`helpwin` <*fonction ou commande*>.

Attention : Matlab fait la distinction entre lettres minuscules et majuscules, or, toutes les instructions Matlab sont en **minuscules**, bien que, dans l'aide, elles apparaissent écrites en majuscules afin de les mettre en évidence.

L'aide peut également être invoquée et obtenue dans la fenêtre de commande (`help`), selon le schéma : >>`help` <*fonction ou commande*>.

L'instruction `lookfor` fournit la liste des fonctions et commandes contenant le **mot-clé** spécifié dans la première ligne de leur texte d'aide : >>`lookfor` <*mot-clé*>.

3. QUELQUES GÉNÉRALITÉS

L'instruction `which` <*nom_fonction*> retourne le **chemin d'accès** de la fonction spécifiée. La sélection de ce dernier et un clic droit permettent alors d'éditer le code correspondant.

L'instruction `type` <*nom_fichier*> permet de **visualiser le contenu** de ce fichier.

L'instruction `what` renvoie **la liste des fichiers** spécifiques à Matlab, notamment ceux possédant l'extension **.m**, **dans le répertoire courant**.

La commande `pack` permet de **défragmenter** la mémoire.

Pour **sauvegarder** la session en cours dans un fichier texte, utiliser la commande `diary`.

Enfin, les commandes DOS `cd`, `pwd`, `dir` sont utilisables dans Matlab.

La séquence de touches **CTRL-C** permet de mettre fin à toute opération ou exécution en cours.

Les noms des fichiers de programmes simples (**scripts**) et des **fonctions** doivent être suivis de l'extension **.m**. Ces noms ne doivent comporter ni caractères spéciaux (accents, slash, etc.) ni espaces.

Les variables créées dans un script entrent dans l'espace de travail (ce sont des **variables globales**). Au contraire, les variables créées dans une fonction sont **locales**, les fonctions permettent donc d'effectuer des passages de paramètres. Pour qu'un fichier programme soit reconnu comme une fonction, son entête doit être au format :

`Function` [*liste des arguments de sortie*] =nom_de_fonction(*liste des arguments d'entrée*)

Ces listes utilisent la virgule comme séparateur : `arg1,arg2,…`

Attention : le nom_de_fonction doit être identique au nom du fichier !

Si un script peut être invoqué en tapant simplement son nom (sans l'extension) dans la fenêtre de commande, **l'appel à une fonction** se fait selon le schéma :

[*liste des arguments de sortie*] =nom_de_fonction(*liste des arguments d'entrée*)

Tout ce qui suit le caractère "%" (pourcent) sur une ligne n'est pas interprété par Matlab et est donc considéré comme un **commentaire**. Les premières lignes de commentaires en tête de fichier sont celles qui apparaissent lors d'une demande d'aide sur le fichier en question par la commande `help`.

Lorsqu'un programme comporte une erreur, l'exécution de celui-ci s'arrête au niveau de l'erreur en question. Matlab fournit alors un **message d'erreur** (en rouge), ce dernier est souvent très explicite (bien qu'en anglais) et **ne doit donc pas être négligé lors du débogage !** D'autres messages, en noir, peuvent intervenir, ce sont des **avertissements** (par exemple, pour signaler l'utilisation d'une commande obsolète ou une division par 0).

Les **séparateurs d'instructions** sont les caractères "`,`" (virgule) et "`;`" (point-virgule). Que ce soit dans un fichier programme ou dans la fenêtre de commande, le point-virgule permet d'**éviter l'affichage à l'écran** du résultat de la commande qui le précède. Ces deux séparateurs s'utilisent également lors de la définition de matrices.

Matlab possède des **variables prédéfinies** : `pi`, qui contient une approximation de π, `Inf` pour $+\infty$, `NaN` (Not-a-Number) pour le résultat d'une opération non définie comme `0/0`, `i` et `j` pour $\sqrt{-1}$ (pour travailler sur des complexes), ou encore `ans` qui contient le résultat de la dernière instruction exécutée.
Attention : ces variables peuvent être redéfinies (parfois sans y prêter attention) par l'utilisateur. Ce peut être le cas par exemple avec `i` et `j` qui sont souvent utilisés comme indices de boucle !

En Matlab les calculs sont effectués avec une **arithmétique à précision finie**. Ceci le différencie des logiciels de calcul symbolique tels que Maple, mais la comparaison n'a pas lieu d'être. Calcul numérique et calcul symbolique sont des outils complémentaires du calcul scientifique. Par ailleurs, Matlab possède une boîte à outils optionnelle de calcul symbolique.
La variable prédéfinie `eps` correspond à la **précision relative** des nombres. Elle est égale à 2^{-52} et est donc de l'ordre de 10^{-16}, l'arithmétique sur ordinateur n'étant pas exacte (l'étendue des nombres représentables s'étend de 10^{-308} à 10^{308}).

Bien que tous les calculs soient effectués en double précision, il existe différents **formats d'affichage** des nombres, parmi lesquels `short` (format par défaut : virgule fixe avec 5 chiffres), `long` (virgule fixe avec 15 chiffres), `short e` (virgule flottante et 5 chiffres), `long e` (virgule flottante et 15 chiffres), `rat` (approximation par un ratio d'entiers), etc.

Exemple : `>> format long, pi` → `ans = 3.14159265358979`

Avec Matlab on travaille essentiellement avec un seul type d'objet : **une matrice**. En Matlab, une matrice est un tableau rectangulaire de nombres représentés en virgule flottante avec une double précision, c'est à dire 8 octets pour un nombre réel et 16 octets pour un nombre complexe. Une matrice 1×1 est interprétée comme un scalaire, celle ayant une seule ligne ou une seule colonne comme un vecteur.

Les commandes `format loose` et `format compact` permettent **de régler l'interligne à l'affichage**.

4. PRINCIPALES COMMANDES MATLAB

Opérateurs de base : ceux-ci agissent sur les scalaires et les matrices. Dans le cas matriciel, les dimensions des matrices doivent s'y prêter !

`+`	Addition
`-`	Soustraction
`*`	Multiplication
`/`	Division
`\`	Division à gauche
`^`	Élévation à une puissance

La division à gauche `A\B` donne le résultat de l'opération $A^{-1}B$ (utilisée pour résoudre un système linéaire).

Opérateurs "élément par élément" : ils s'utilisent dans le cas matriciel. Là encore, les dimensions des matrices (ou l'orientation dans le cas de vecteurs) doivent correspondre.

`.*`	Multiplication élément par élément
`./`	Division élément par élément
`.^`	Élévation à une puissance élément par élément

Autres opérateurs matriciels :

`inv``(.)`	Inversion d'une matrice
`'`(`A'`)	Transposition-conjugaison
`.'`(`A.'`)	Transposition simple
`fliplr``(.)`	Modification de l'ordre des colonnes dans le sens gauche-droite
`flipud``(.)`	Modification de l'ordre des lignes dans le sens haut-bas

Voir aussi : `rot90`, `flipdim`, `permute`, `reshape`, …

Dans le cas où la matrice `A` ne contient que des réels, l'opérateur `'` n'effectue qu'une transposition.

Opérateurs relationnels :

`<`	Strictement inférieur
`<=`	Inférieur ou égal
`>`	Strictement supérieur
`>=`	Supérieur ou égal
`==`	Égal
`~=`	Différent

Matlab renvoie 1 pour un événement vrai et 0 pour un événement faux.

Opérateurs logiques :

&	ET logique (AND)
|	OU logique (OR)
~	NON logique (NOT)
xor	OU exclusif (XOR)

Instructions mathématiques de base :

Celles-ci s'appliquent aux scalaires comme aux matrices. Dans le cas matriciel, elles s'appliquent à chaque élément de la matrice.

Sqrt(.)	Racine carrée	**abs(.)**	Valeur absolue ou module
exp(.)	Exponentielle	**angle(.)**	Phase
log(.)	Logarithme naturel	**conj(.)**	Conjugaison complexe
log10(.)	Logarithme décimal	**imag(.)**	Partie imaginaire
log2(.)	Logarithme de base 2	**Isreal(.)**	Vrai si réel
nextpow2(.)	Puissance de 2 supér.	**real(.)**	Partie réelle
cos(.)	Cosinus	**fix(.)**	Arrondi vers zéro
acos(.)	Arc cosinus	**floor(.)**	Arrondi vers -∞
sin(.)	Sinus	**ceil(.)**	Arrondi vers +∞
asin(.)	Arc sinus	**round(.)**	Arrondi au plus proche entier
cot,tan(.)	(Co)Tangente	**sign(.)**	Signe
atan(.)	Arc tangente	**factorial(.)**	Factorielle
sinc(.)	Sinus cardinal	**rem(.)**	Reste division euclidienne
cosh,sinh, tanh(.)	Fonctions hyperboliques	**mod(.)**	Reste modulo

Instructions pour les matrices :

det(.)	Déterminant
diag(.)	Création d'une matrice diagonale ou sélection de la diagonale
zeros(.)	Création d'une matrice de zéros
ones(.)	Création d'une matrice de uns
eye(.)	Matrice identité
size(.)	Dimensions d'une matrice
length(.)	Dimension maximale d'une matrice (longueur d'un vecteur)
expm(.)	Exponentielle matricielle
rank(.)	Rang de la matrice

Voir aussi : `cond`, `norm`, `trace`, `eig`, `poly`, …

Instructions polynomiales :

conv(.)	Produit de polynômes et produit de convolution
deconv(.)	Division polynomiale et déconvolution
polyval(.)	Calcul des valeurs d'un polynôme
roots(.)	Racines d'un polynôme
poly(.)	Définition d'un polynôme à partir des racines

Voir aussi : `polyder`, `polyfit`, `residue`, …

Instructions statistiques :

min(.)	Valeur minimale	**xcov(.)**	Fonction d'auto ou d'intercovariance
max(.)	Valeur maximale	**xcorr(.)**	Fonction d'auto ou d'intercorrélation
mean(.)	Valeur moyenne	**psd(.)**	Estimateur de la densité spectrale de puissance
median(.)	Valeur médiane	**hist(.)**	Histogramme
std(.)	Écart-type	**rand**	Matrice aléatoire uniforme
cov(.)	Covariance	**randn**	Matrice aléatoire gaussienne
var(.)	Variance	**sort(.)**	Tri des éléments

La commande `cov` utilisée sur un vecteur renvoie la variance et est équivalente dans ce cas à la commande `var`.

Caractères spéciaux :

:	Boucle implicite : génération de vecteurs. Indice : "toutes les lignes" ou "toutes les colonnes"	**[.]**	Définition de matrices ou arguments de sortie de fonctions
…	Continuation à la ligne	**(.)**	Expressions math. Ou arguments d'entrée de fonctions
,	Sépare 2 éléments sur une ligne, sépare 2 instructions en autorisant l'affichage du résultat de la première	**;**	Sépare 2 lignes, sépare 2 instructions en interdisant l'affichage
.	Point décimal	**..**	Répertoire parent
'	Délimiteur de chaîne de caractère (à doubler en cas d'apostrophe)	**=**	Affectation

Instructions graphiques :

plot(.)	Graphe simple	**grid**	Grille de graduation
subplot(.)	Plusieurs graphes sur une figure	**hold**	Maintien du graphe
stem(.)	Représentation par impulsions	**axis(.)**	Gestion des axes
bar(.)	Représentation type histogramme	**xlabel(.)**	Légende abscisses
stairs(.)	Représentation en escalier	**ylabel(.)**	Légende ordonnées
figure(.)	Nouvelle figure	**title(.)**	Titre de la figure
clf	Effacer la figure courante	**legend(.)**	Légende de la figure
close(.)	Fermer la figure	**semilogx(.)**	Échelle log sur les abscisses
mesh(.)	Graphe 3D	**semilogy(.)**	Échelle log sur les ordonnées
polar(.)	Graphe en coordonnées polaires	**loglog(.)**	Échelle log sur les 2 axes

Voir aussi : `axes`, `ginput`, `surf`, `zlabel`, …

Instructions de contrôle du temps :

clock	Date et heure sous forme de vecteur
cputime	Temps CPU
date	Date du jour (chaîne de caractères)
now	Date du jour et heure (nombre)
etime(.)	Intervalle de temps écoulé
tic, toc	Chronomètre

Autres instructions (en vrac) :

fft(.)	Transformée de Fourier discrète	**end**	Index de fin par défaut
ifft(.)	TFD inverse	**input(.)**	Entrée de l'utilisateur
fftshift(.)	Échange des 2 moitiés d'un vecteur	**fprintf(.)**	Affichage dans un fichier ou à l'écran
filter(.)	Filtrage	**disp(.)**	Affichage à l'écran (texte et matrice)
boxcar(.)	Fenêtre rectangulaire	**num2str(.)**	Conversion d'un nombre en chaîne de caractères
rectwin(.)	Idem : fenêtre rectangul.	**int2str(.)**	Conversion d'un entier en chaîne de caractères
linspace(.)	Vecteur à espacement linéaire	**strcmp(.)**	Comparaison de chaînes de caractères
logspace(.)	Vecteur à espacement logarithmique	**eval(.)**	Exécution d'une chaîne de caractères comme une commande
sum(.)	Somme des éléments	**any(.)**	Vrai si au moins un élément non-nul
prod(.)	Produit des éléments	**all(.)**	Vrai si tous les éléments sont non-nuls
cumsum(.) **cumprod(.)**	Somme cumulée des éléments, produit cumulé	**find(.)**	Indices d'éléments non-nuls
nargin(.)	Nombre d'arguments d'entrée	**exist(.)**	Vérifie si une fonction ou une variable existe
nargout(.)	Nombre d'arguments de sortie	**isempty(.)**	Vrai pour une matrice vide

Voir aussi les commandes : `str2num`, `isequal`, `isinf`, `isnan`, `issparse`, `isstr`, …

En plus de la fenêtre rectangulaire déjà citée, il existe également plusieurs autres types de fenêtres utilisées en traitement du signal : `bartlett`, `barthannwin`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttalwin`, `parzenwin`, `triang`, `tukeywin`

Instructions de contrôle :

break	Termine l'exécution d'une boucle
continue	Passe à l'itération suivante en sautant les instructions suivantes dans une boucle
else	Sinon, utilisé avec `if`
elseif	Sinon si, utilisé avec `if`
end	Termine `for`, `if` et `while`
error(.)	Termine un programme et retourne un message d'erreur

for	Répétition
if	Instruction conditionnelle
otherwise	Sinon, utilisé avec `switch`
pause	Arrêt momentané d'un programme (attente d'appui sur une touche)
return	Retour à la fonction appelante ou au clavier
switch	Branchement conditionnel
while	Tant que

a) Branchement conditionnel : if … then … else

L'instruction `if` permet d'exécuter un bloc d'instructions en fonction de la valeur logique d'une expression. **Syntaxe :**

```
if <expression>
      <instructions>
end
```

Le groupe d'instructions est exécuté si seulement si l'expression est vraie.

Il est possible d'utiliser des branchements multiples, exemple :

```
if length(x)~=1
      disp('nonscalaire')
elseif isnan(x)
      disp('NaN')
elseif x>0
      disp('positif')
elseif x<0
      disp('negatif')
else
      disp('nul')
end
```

Si `A` et `B` sont deux matrices, il est possible d'utiliser l'expression `A==B` comme test logique. Le bloc d'instructions venant à la suite du `if A==B` est exécuté si et seulement si les deux matrices sont égales. Attention cependant à ce test d'égalité, car il renvoie un message d'erreur si les deux matrices n'ont pas les mêmes dimensions. Il est préférable d'utiliser la fonction `isequal` pour tester l'égalité entre plusieurs matrices.

b) Branchement conditionnel multiple : switch

Syntaxe :

```
switch <expression>
case <valeur1>
      <instructions1>
case <valeur2>
      <instructions2>
...
otherwise <instructions>
end
```

L'expression doit être un scalaire ou une chaîne de caractères. Dans le cas scalaire, elle est comparée successivement avec `valeur1`, `valeur2`, etc. Dès que le premier test `<expression>==valeur` retourne la valeur vraie, le bloc d'instructions qui

suit est exécuté. Si aucun test n'est validé, le bloc qui suit `otherwise` est exécuté. Exemple :

```
switch mod(x,3)
case 0
     disp('multiple de 3')
case 1
     disp('x=3 [1]')
case 2
     disp('x=3 [2]')
otherwise
     disp('autre')
end
```

c) Boucle finie : for

La boucle `for` permet de répéter un bloc d'instructions un certain nombre de fois.

Syntaxe :

```
for <variable>=<expression>
     <instructions>
end
```

Exemple : édition d'une table de multiplication.

```
n=10;
t=1:n;     % le double point est ici utilisé pour effectuer une
           % boucle implicite, créant un vecteur constitué des
           % entiers de 1 à n.
T=[];
for i=1:n
     T=[T;t*i];
end
T
```

Bien sûr, plusieurs boucles peuvent être imbriquées.

Remarque : la variable peut être un vecteur et l'expression une matrice. Exemple :

```
>> for u=eye(4),u',end
ans =
     1     0     0     0
ans =
     0     1     0     0
ans =
     0     0     1     0
ans =
     0     0     0     1
```

d) Boucle infinie : while

Une boucle `while` permet d'exécuter un bloc d'instructions tant qu'une expression logique est vraie. **Syntaxe :**

```
while <expression>
     <instructions>
end
```

Exemple : calcul de la précision machine `eps`.

```
e=1;
while 1+e>1
     e=e/2;
end
epsilon=2*e
```

Remarques :

Bien que ce point soit en cours de correction avec les nouvelles versions de Matlab, dans la plupart des cas **l'exécution des boucles `for` et `while` reste lente comparativement à des instructions matricielles**. Or, il est souvent possible de remplacer une telle boucle par une ou plusieurs instructions matricielles équivalentes.

Exemple : création d'un vecteur contenant 1001 valeurs de la fonction sin(t).

Boucle "classique" :

```
n = 0;
for t = 0:.01:100    % de 0 à 100 par pas de 0,01
    n = n+1;
    y(n) = sin(t);
end
```

Programmation Matlab :

```
t = 0:.01:100;
y = sin(t);
```

Une autre façon d'accélérer un programme Matlab consiste à **pré allouer de l'espace mémoire**, pour une variable dont la taille est censée être augmentée au cours d'une boucle, par exemple en créant une variable de la taille finale souhaitée, mais ne contenant que des valeurs nulles (instruction `zeros`).

Enfin, du point de vue de la rapidité d'exécution, il est toujours préférable de **programmer sous forme de fonctions** plutôt que de scripts. En effet, après le premier appel, une fonction est plus rapide car Matlab l'aura compilé en pseudo-code, alors qu'un script est toujours exécuté ligne par ligne.

5. EXERCICES D'INITIATION :

Exercice 1 : *Matlab utilisé comme une calculatrice*

On travaille directement dans la fenêtre de commande. Définir les scalaires `a` et `b` :

```
>>a=5;
>>b=2
>>a+b
```

Vérifier l'effet du point-virgule en fin de ligne, vérifier le contenu de la variable `ans`. Vérifier le contenu de l'espace de travail.

Tester les autres principaux opérateurs (`-, *, /, \, ^`) sur les scalaires `a` et `b`.

Effacer le contenu de l'espace de travail.

Corrigé :

On vérifie que le point-virgule en fin d'instruction permet d'éviter l'affichage à l'écran du résultat de celle-ci. Après la troisième ligne d'instruction, la variable `ans` contient le dernier résultat c'est-à-dire la valeur `7`. A ce moment, l'espace de travail contient les variables `a`, `b`, et `ans`, qui sont trois scalaires (matrices de dimensions 1x1, codées sur 8 octets chacune). Un double-clic sur l'une des variables permet de visualiser sa valeur. Les opérateurs `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division) et `^` (élévation à une puissance) agissent de façon conventionnelle comme le ferait une calculette de poche. L'opération de "division à gauche" `a\b` correspond ici à l'opération $a^{-1}*b$ c'est-à-dire à `b/a`. Le contenu de l'espace de travail peut être effacé grâce à la commande `clear`.

Exercice 2 : *opérations de base sur des vecteurs*

On travaille toujours dans la fenêtre de commande. Définir les vecteurs `A`, `B`, `C` et `D` :

```
>>A=[1 2 3] ,B=[2 3 4] ,C=[1,2,3], D=[2 3 4]'
```

Vérifier l'égalité entre `A` et `C`. Examiner le contenu de l'espace de travail. Quelle relation y a-t-il entre B et D ? Effectuer chacune des opérations suivantes et commenter les différents résultats :

```
A+B; A-B; A-2; A*B; A.*B; A'*B; A*B'; A*2; A.*2; A\B; A.\B; 2\A;
2.\A; A\2; A.\2; A/B; A./B; A/2; A./2; 2/A; 2./A; B^A; B.^A; A^2;
A.^2; 2^A; 2.^A; (A+i*B).'; (A+j*B)'; Z=A+2i; A(2)*2; A(3)+B(1);
A(4); A(0); A(-1); A(1.5);
```

Réexaminer le contenu de l'espace de travail. Effacer son contenu en conservant `A` et `B` pour l'exercice suivant.

Corrigé :

L'égalité entre les vecteurs `A` et `C` est due à l'équivalence entre les séparateurs "virgule" et "espace", ils permettent donc de séparer des éléments d'une même ligne. On verra par la suite que pour séparer deux lignes, il faut utiliser le "point-virgule". Dans l'espace de travail, il apparaît clairement que les 3 premiers vecteurs définis sont des vecteurs "ligne" donc de dimension 1x3. Au contraire, le vecteur `D`, obtenu par transposition de `B`, est un vecteur "colonne" de dimension 3x1. Ainsi, Matlab prend en compte l'orientation des vecteurs et il en est de même lors d'opérations sur ces derniers.

`A+B` → `ans= 3 5 7` : addition de 2 vecteurs

`A-B` → `ans= -1 -1 -1` : soustraction de 2 vecteurs

`A-2` → `ans=-1 0 1` : équivalent à `A-[2 2 2]`

`A*B` → `??? Error using ==> * Inner matrix dimensions must agree` : les dimensions des vecteurs (ou plutôt ici, l'orientation) sont incompatibles avec une multiplication vectorielle.

`A.*B` → `ans= 2 6 12` : multiplication élément par élément

`A'*B` → `ans=` 2 3 4 : multiplication matricielle
4 6 8
6 9 12

`A*B'` → `ans= 20` : idem (dimension 1x3 * 3x1 = 1x1)

`A*2` → `ans= 2 4 6` : multiplication d'un vecteur par un scalaire

`A.*2` → `ans= 2 4 6` : multiplication élément par élément mais par un scalaire (l'opération équivaut à la précédente)

`A\B` → `ans=` 0 0 0
0 0 0
0.6667 1.0000 1.3333

Il s'agit d'une division à gauche, la réponse est telle que `A*ans = B`

`A.\B` → `ans= 2.0000 1.5000 1.3333` : la division à gauche est faite élément par élément

`2\A` → `ans= 0.5000 1.0000 1.5000` : division à gauche par un scalaire (`2*ans=A`)

`2.\A` → `ans= 0.5000 1.0000 1.5000` : division à gauche élément par élément mais par un scalaire (équivaut à la précédente)

`A\2` → `ans=` 0
0
0.6667 : le résultat est tel que `A*ans = 2`

`A.\2` → `ans= 2.0000 1.0000 0.6667` : idem, mais élément par élément

`A/B` → `ans= 0.6897` : cette opération équivaut à `(B'\A')'`

`A./B` → `ans= 0.5000 0.6667 0.7500` : division de chaque élément de `A` par l'élément correspondant de `B`

`A/2` → `ans= 0.5000 1.0000 1.5000` : division de chaque élément par 2

`A./2` → `ans= 0.5000 1.0000 1.5000` : idem

`2/A` → `??? Error using ==> / Matrix dimensions must agree.`

`2./A` → `ans= 2.0000 1.0000 0.6667` : équivalent à `A.\2`

`B^A` → `??? Error using ==> ^ At least one operand must be scalar`

`B.^A` → `ans= 2 9 64` : chaque élément de `B` à la puissance (élément de `A`)

`A^2` → `??? Error using ==> ^ Matrix must be square` : l'opération `A*A` ne peut se faire que si `A` est carrée.

`A.^2` → `ans= 1 4 9` : les éléments de `A` au carré

`2^A` → `??? Error using ==> ^ Matrix must be square` : l'élévation à une puissance matricielle n'est possible que pour une matrice carrée. En effet, `2^A` est équivalent à `expm(A*log(2))` et `expm(B)`$=\sum_{k=0}^{\infty}\frac{1}{k!}B^k$

`2.^A` → `ans= 2 4 8` : élévation à la puissance par les éléments de `A`

`(A+i*B).'` → `ans=` 1.0000 + 2.0000i
2.0000 + 3.0000i
3.0000 + 4.0000i

Transposition simple de `A+jB` (`j` et `i` sont équivalents)

```
(A+j*B)' → ans=      1.0000 - 2.0000i
                     2.0000 - 3.0000i
                     3.0000 - 4.0000i
```

Transposition et conjugaison de `A+jB (A'-jB')`

`Z=A+2i → Z= 1.0000 + 2.0000i   2.0000 + 2.0000i   3.0000 + 2.0000i`

Ici, il n'est pas nécessaire de faire figurer le signe * devant `i` (ceci peut d'ailleurs accélérer l'exécution dans certains cas)

`A(2)*2 → ans= 4` : `A(2)` désigne le deuxième élément de `A`.

`A(3)+B(1) → ans= 5`

`A(4) → ??? Index exceeds matrix dimensions.` `A` ne comprend que 3 éléments

`A(0) → ??? Subscript indices must either be real positive integers or logicals.`

`A(-1) → ??? Subscript indices must either be real positive integers or logicals.`

`A(1.5) → ??? Subscript indices must either be real positive integers or logicals.`

Sous Matlab, les indices sont des **entiers strictement positifs** (ou des variables de type `logical`).

L'examen de l'espace de travail montre l'existence d'une variable complexe (`Z`). Pour effacer les variables inutiles : `>> clear C D Z ans`

Exercice 3 : *opérations de base sur des matrices, concaténations, extractions*

Si `A11` est une matrice de format `(m1,n1)` (noté aussi `m1xn1`), `A12` une matrice `m1xn2`, `A21` une matrice `m2xn1`, et `A22` une matrice `m2xn2`, alors on peut définir la matrice `A` :

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

par *concaténation* des matrices précédentes. La syntaxe est la suivante :

```
A=[A11 A12; A21 A22]
```

On peut procéder de même avec plus de blocs encore. D'autre part, la fonction `zeros(m,n)` renvoie une matrice de `0` avec m lignes et n colonnes. De plus, comme on l'a déjà vu dans le cas particulier des vecteurs, l'opérateur `'` permet de transposer une matrice (dans Matlab, A^T s'écrit donc `A'`).

Une instruction très utilisée dans Matlab est de la forme :

`ind = deb : inc : fin`

Elle permet de créer un vecteur ligne dont la première composante est `deb`, les composantes suivantes étant obtenues par incrémentation de `inc` jusqu'à ne pas dépasser `fin`. Si `inc` est strictement positif (resp. strictement négatif) et si `deb > fin` (resp. `deb < fin`), aucune composante n'est créée et l'on obtient un objet appelé matrice vide. Par défaut, l'incrément est `1`.

Pour l'extraction de vecteurs ou de matrices, on peut en fait utiliser les expressions `v(ind)`, comme on l'a vu dans l'exercice précédent, et `A(indi,indj)` si les vecteurs `ind`, `indi` et `indj` ne contiennent que des indices entiers bien situés, c'est-à-dire si `ind` ne

contient aucun entier (str.) inférieur à `1` et (str.) supérieur au nombre de composantes de `v` et de même pour les 2 autres relativement au nombre de lignes et de colonnes de `A`.
D'autres possibilités vont être explorées dans cet exercice.

a) Toujours dans la fenêtre de commande, vérifier que les 4 lignes d'instructions suivantes définissent la même matrice :

```
>>M1=[1 2 3;2 3 4;3 4 5]
>>M2=[1 2 3                % entrée
2 3 4                      % entrée
3 4 5]
>>M3=[1:3;2:4;3:5]
>>M4=[A;B;3:1:5]
```

Expliquer l'utilisation du point virgule ainsi que celle du double point.
Obtenir le résultat suivant d'au moins deux manières différentes :

```
M =
     1     2     3     1
     2     3     4     2
     3     4     5     6
```

b) Réinitialiser l'espace de travail et créer la matrice `A` suivante : $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$

Commenter le résultat des commandes suivantes :

```
>> A([1,3],2), A([1,3],[2,1,4])
```

Trouver la commande qui retourne le résultat suivant :

```
B =   5    6   7
      9    10  11
      13   14  15
```

Tester la commande `A(:)` puis expliquer le résultat de la commande :

```
>> A([1 2 3; 4 5 6])
```

Prévoir la réponse à la commande `>> A(10:-1:5)`
Comment créer un vecteur contenant les éléments de `A` dans l'ordre suivant :
`16 12 8 4 15 11` ... ?
Quel est le résultat de l'instruction `[A(1,:),5]` ?
Comment créer un vecteur contenant les éléments de `A` dans l'ordre suivant : `1 2 3 4` ... ?
Effectuer la somme des éléments de la diagonale principale de `A` et vérifier le résultat avec la commande `trace`.
Quel est le résultat de `diag(diag(A))` ?
Créer une matrice identité 4x4 de deux manières différentes (commande `eye` et `diag`).
A l'aide de la commande `reshape`, transformer la matrice `A` en une matrice de dimension 2x8.
A l'aide des commandes `zeros` et `ones`, compléter la matrice `B` de façon à obtenir le résultat :

```
C =
     5     6     7     0
     9    10    11     0
    13    14    15     0
     1     1     1     1
```

A l'aide de la commande `rand`, créer une matrice aléatoire `D` de distribution uniforme et de dimension 4x4.
Vérifier l'équivalence entre les commandes `D\A` et `inv(D)*A`.
Obtenir le déterminant et le rang de la matrice `D`.
Proposer une méthode pour extraire les valeurs positives de la matrice `E=D-.5`.

Corrigé :
a) `M1`, `M2`, `M3` et `M4` définissent bien la même matrice. Dans la définition de `M1`, le point virgule est utilisé pour délimiter des nouvelles lignes. Pour `M2`, l'appui sur la touche entrée permet d'obtenir le même résultat. Dans le cas de `M3`, le point virgule est à nouveau utilisé pour délimiter les différentes lignes, le contenu de celles-ci étant obtenu à l'aide du double point. Ainsi, par exemple, `1:1:3` génère un vecteur ligne contenant les valeurs de `1` à `3` avec un incrément de `1`. On aurait pu d'ailleurs écrire directement `1:3`. Enfin, dans le cas de `M4`, on met à profit l'existence des vecteurs `A` et `B` créés dans l'exercice précédent.
Pour obtenir la matrice `M`, on peut par exemple procéder comme suit :
`>> M=[A 1; B 2; 3:6]` ou bien encore : `>> M=[M4 [1 2 6]']`

b) La commande `clear` permet de réinitialiser l'espace de travail. La matrice `A` peut être générée ainsi : `>> A = [1:4; 5:8; 9:12; 13:16]`
`>> A([1,3],2)` → `ans =  2`
`10`
Cette instruction permet d'extraire les composantes `1` et `3` de la 2ième colonne de `A`.
`>> A([1,3],[2,1,4])` → `ans =  2  1  4`
`10  9  12`
Cette instruction permet d'extraire les composantes `1` et `3` des colonnes `2` puis `1` puis `4` de la matrice `A`.
Pour créer la matrice `B`, on peut utiliser : `>> B=A(2:4,1:3)`
L'instruction `A(:)` crée un vecteur colonne contenant les éléments de `A` énumérés colonne par colonne. Noter que l'instruction `A(7)` retourne la valeur `10`, car les éléments d'une matrice sont numérotés selon ce même ordre d'énumération. Ainsi, l'élément `A(i,j)` est numéroté `i+m*(j-1)`, où `m` est le nombre de lignes de `A` :

1	m+1	2m+1	⋯
2	m+2	2m+2	⋯
⋮	⋮	⋮	⋮
m	2m	3m	⋯

D'où le résultat : `>> A([1 2 3; 4 5 6])` → `ans =  1  5  9`
`13  2  6`
`>> A(10:-1:5)` → `ans = 7  3  14  10  6  2`
(de l'élément n°10 à l'élément n°5)
Pour obtenir un vecteur contenant les éléments de `A` dans l'ordre `16 12 8 4 15 11` ..., il suffit de taper : `>> A(16:-1:1)`, ou, de manière équivalente : `>> A(end:-1:1)`
`>> [A(1,:),5]` → `ans = 1  2  3  4  5` (crée un vecteur ligne constitué de la première ligne de `A` et de la valeur `5`)

Pour créer un vecteur contenant les éléments de A dans l'ordre : 1 2 3 4 ..., il suffit de transposer A au préalable : `>> AT=A'; AT(:)'`
La somme des éléments de la diagonale se calcule selon :
`>> A(1,1)+A(2,2)+A(3,3)+A(4,4)`, résultat que l'on trouve directement avec la commande trace : `>> trace(A)` → `ans = 34`
La commande `diag(A)` permet d'extraire les éléments de la diagonale de A dans un vecteur. Appliquée à un vecteur, cette commande crée une matrice diagonale, d'où le résultat : `>> diag(diag(A))` →

```
ans =     1     0     0     0
          0     6     0     0
          0     0    11     0
          0     0     0    16
```

La matrice identité (4,4) peut être créée selon : `>> diag([1 1 1 1])` ou `>> eye(4)`

```
>> reshape(A,2,8) → ans = 1    3    5    7    9   11   13   15
                          2    4    6    8   10   12   14   16
```

Pour obtenir la matrice C souhaitée : `>> C=[[B zeros(3,1)];ones(1,4)]`

```
>> D=rand(4) → D =    0.9501    0.8913    0.8214    0.9218
                      0.2311    0.7621    0.4447    0.7382
                      0.6068    0.4565    0.6154    0.1763
                      0.4860    0.0185    0.7919    0.4057
>> D\A → ans =      -22.3311  -23.5501  -24.7691  -25.9881
                     4.8461    5.7858    6.7255    7.6651
                     36.9071   39.2159   41.5247   43.8336
                    -13.4709  -14.0956  -14.7202  -15.3448
```

Le même résultat est bien sûr obtenu par : `>> inv(D)*A`
Le déterminant de D est calculé par la commande : `>> det(D)` → `ans = 0.1155`
Le rang se calcule avec la commande : `>> rank(D)` → `ans = 4`

```
>> E=D-.5 → E =     0.4501    0.3913    0.3214    0.4218
                   -0.2689    0.2621   -0.0553    0.2382
                    0.1068   -0.0435    0.1154   -0.3237
                   -0.0140   -0.4815    0.2919   -0.0943
```

Le résultat d'une opération relationnelle ou logique est du type `logical`. Un tableau de type `logical` peut être utilisé comme tableau d'indices d'une matrice de même dimension. Pour extraire les valeurs positives de E, on peut donc procéder comme suit :

```
>> Epos=E>0 → Epos =     1     1     1     1
                         0     1     0     1
                         1     0     1     0
                         0     0     1     0

>> E(Epos)' → ans =
  0.4501 0.1068 0.3913 0.2621 0.3214 0.1154 0.2919 0.4218 0.2382
```

L'opération `Epos=E>0` retourne une matrice de même dimension telle que `Epos(i,j)=1` si `E(i,j)>0` et `Epos(i,j)=0`. La matrice Epos est utilisée ensuite pour extraire les valeurs positives de E.

Exercice 4 : *premier script, première fonction*

Soit : $A = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}, x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

Ecrire un script (que l'on pourra appeler `script1.m`), qui :

- permette de donner une valeur à l'angle θ,
- permette de définir la matrice A et le vecteur x,
- calcule y=Ax.

Expérimenter votre script avec plusieurs valeurs pour θ et pour le vecteur x.
Comment :

- extraire la deuxième composante de x ?
- extraire l'élément $a_{1,2}$?
- extraire la première ligne de A ?
- extraire la deuxième colonne de A ?

Modifier le script pour en faire une fonction dont l'argument d'entrée est la valeur de θ, et tester la fonction.

Corrigé :

```
%script1
theta=pi/2;
x=[1;2];
A=[cos(theta) -sin(theta); sin(theta) cos(theta)];
y=A*x
```

Pour exécuter ce script, on peut taper son nom dans la fenêtre de commande ou bien, dans l'éditeur, on peut utiliser le menu "Debug" puis le sous-menu "Run" (touche F5), ou encore cliquer sur l'icône "Run" du menu (petit fichier avec flèche bleue vers le bas).
Deuxième composante de x : `x(2)`
Élément $a_{1,2}$: `A(1,2)`
Première ligne de A : `A(1,:)`
Deuxième colonne de A : `A(:,2)`
Pour en faire une fonction, il suffit de modifier l'en-tête du fichier selon :

```
function [z]=exo1(theta);
x=[1;2];
A=[cos(theta) -sin(theta); sin(theta) cos(theta)];
z=A*x;
```

Pour invoquer la fonction, il faut alors taper : `>> y=script1(pi);`

Exercice 5 : *encore une matrice*

Après avoir consulté le `help` et expérimenté la commande `diag(a,k)`, où `a` est un vecteur et `k` un entier relatif, écrire un script qui génère la matrice d'ordre `n` suivante :

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

Corrigé :

La valeur de n étant quelconque, on peut écrire une fonction avec n comme paramètre d'entrée. Une solution possible est donnée par :

```
function [V]=matdiag(n);
v=ones(1,n);
V=2*diag([v 1]);
V2=-diag(v,1);
V3=-diag(v,-1);
V=V+V2+V3;
V=V(1:n,1:n)
```

Ne pas oublier que le nom de la fonction doit correspondre au nom du fichier.

Exercice 6 : *premières boucles, instruction* `if`*, commandes* `find`*,* `eval`*,* `num2str`

a) Recopier dans un fichier appelé `erathostene.m` le script suivant. Faire exécuter ce script et expliquer l'algorithme.

```
n = 49;
T = ones(1,n);
for k = 2:sqrt(n)
     if T(k)==1
           T(2*k:k:n)=0;
     end
end
find(T)
```

b) A l'aide des commandes `eval` et `num2str`, et d'une boucle `for`, écrire un script qui génère automatiquement des matrices appelées `I2` à `I10` et correspondant aux matrices identités de dimensions 2x2, 3x3, …, 10x10.

Corrigé :

a) Cet algorithme permet de trouver les nombres premiers inférieurs à 49. La commande `find` renvoie les indices des éléments non nuls.

b) Consulter le help des fonctions `eval` (exécution du contenu d'une chaîne de caractères comme une commande Matlab) et `num2str` (conversion d'un nombre en chaîne de caractère).

```
for k=2:10
    eval(['I',num2str(k),'=eye(k)'])
end
```

Exercice 7 : *polynômes, premier graphe*

Dans MATLAB, un polynôme p(x) est représenté par la liste de ses coefficients dans l'ordre des puissances décroissantes.

Exemple : $p(x) = x^3 - 8x^2 - 32x - 13$ est représenté par : `p=[1 -8 -32 -13]`

En utilisant la commande `polyval`, évaluer le polynôme p(x) en `x = 2` et en `x =` π.
Déterminer les racines du polynôme p(x) (commande `roots`).
Soit le polynôme $q(x)=4x^2+3$, déterminer les coefficients du polynôme résultant du produit p(x).q(x), puis les coefficients et le reste de la division de p(x) par q(x) (commandes `conv` et `deconv`).
Obtenir le graphe de p(x) sur l'intervalle [-5, 5] avec un pas de 0,01 (commande `plot`).

Corrigé :

```
>> polyval(p,2) → ans =  -101
>> polyval(p,pi) → ans = -161.4815
>> roots(p) → ans =  11.0129
                     -2.5500
                     -0.4629
>> q=[4 0 3];
>> conv(p,q) → ans = 4   -32  -125   -76   -96   -39
>> [f,r]=deconv(p,q)
     f =   0.2500   -2.0000
     r =   0    0   -32.7500   -7.0000
```

Ce qu'on peut "traduire" par : p(x) = q(x).(0,25x - 2) - 32,75x - 7

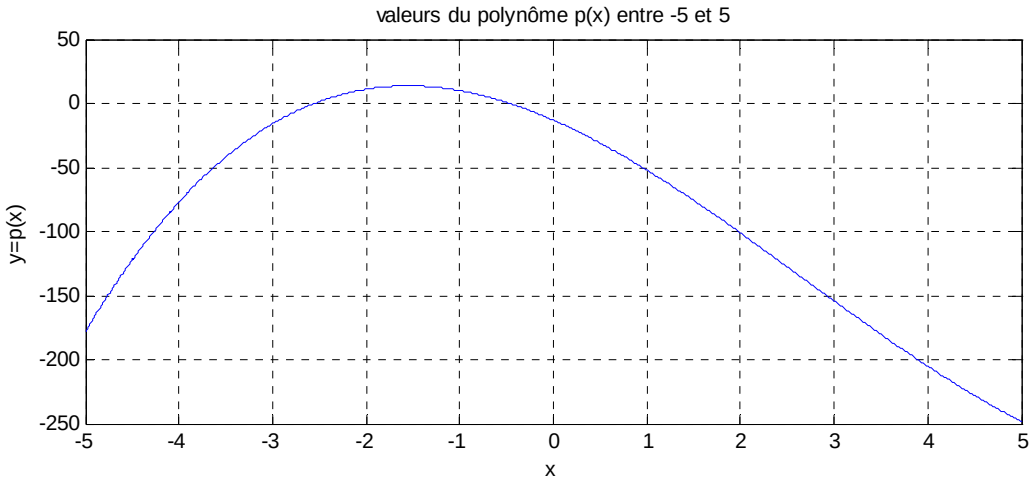
```
>> x=-5:.01:5;
```

Le pas de 0.01 peut bien sûr être modifié si nécessaire.

```
>> y=polyval(p,x);
```

Le vecteur y contient les images par p de tous les éléments de x.

```
>> plot(x,y), title('valeurs du polynôme p(x) entre -5 et 5'), xlabel('x'), ylabel('y=p(x)'), grid
```



Exercice 8 : *`linspace`, opérations élément par élément, graphes multiples*

La fonction `linspace(a,b,n)` permet d'obtenir un vecteur ligne avec `n` composantes régulièrement réparties entre `a` et `b`.

Toutes les fonctions mathématiques classiques (sin, cos, exp,…) sont connues de Matlab et s'appliquent aussi à des vecteurs ou matrices, au sens élément par élément. Par exemple, `w=sin(v)` donnera un vecteur de même format que `v` et tel que $w_i=\sin(v_i)$. Enfin, si deux matrices `A` et `B` sont de même format alors `C=A.*B` et `D=A./B` donnent des matrices de même format que `A` et `B` avec $c_{i,j}=a_{i,j}b_{i,j}$ et $d_{i,j}=a_{i,j}/b_{i,j}$. Il est toléré que l'une des deux matrices soit un scalaire. Ceci fonctionne également avec l'opérateur de puissance `^`, comme on l'a vu dans le cas des vecteurs.

a) Définir un vecteur `x` avec 100 composantes permettant de mailler régulièrement l'intervalle `[0,2pi]`. Définir ensuite le vecteur `y` tel que $y_i=\sin(x_i)$ puis tracer la courbe.

b) Même question avec successivement :

$$f_2(x) = (x-1)(x+1),\ x \in [-2,2]$$

$$f_3(x) = \frac{1}{1+x^2},\ x \in [-3,3]$$

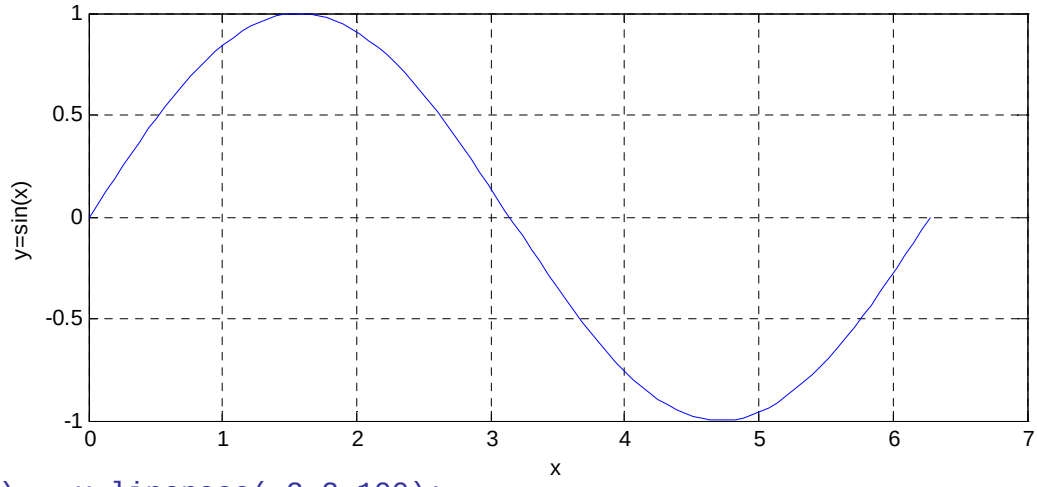
$$f_4(x) = \cos\left(\frac{1}{0.1+x^2}\right),\ x \in [-3,3]$$

Il est possible d'adapter le nombre de points du maillage au besoin. Représenter f_3 et f_4 sur la même figure.

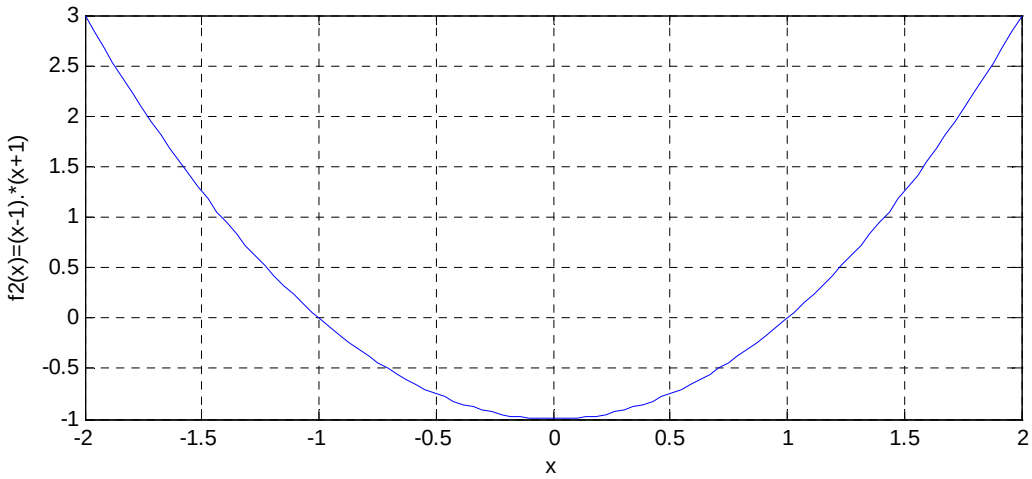
Pour créer plusieurs figures au fur et à mesure des besoins, on pourra utiliser la commande `figure(numéro de figure)`.

Corrigé :

a)
```
x=linspace(0,2*pi,100);
y=sin(x);
figure(1)
plot(x,y),grid, xlabel('x'), ylabel('y=sin(x)')
```

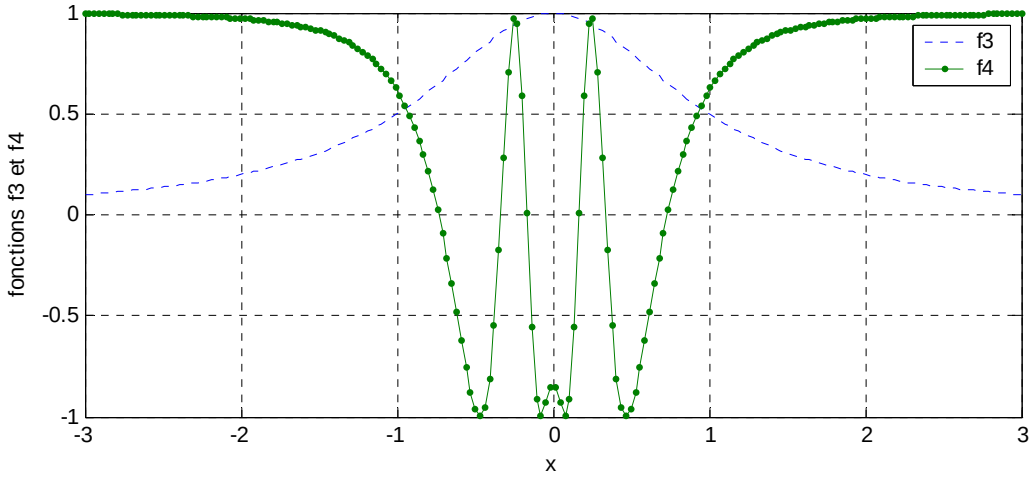


b)
```
x=linspace(-2,2,100);
f2=(x-1).*(x+1);
figure(2)
plot(x,f2),grid, xlabel('x'), ylabel('f2(x)=(x-1).*(x+1)')
```




```
x=linspace(-3,3,200);
f3=1./(1+x.^2);
f4=cos(1./(.1+x.^2));
```

```
figure(3)
plot(x,f3,':',x,f4,'.-'),grid, xlabel('x'),
ylabel('fonctions f3 et f4')
legend('f3','f4')
```



Les options du dernier `plot` permettent de choisir le type de tracé (continu, pointillé, etc.). Pour tracer les 2 vecteurs f3 et f4 sur le même graphe, une autre possibilité est :
`plot(x,[f3' f4'])`
Attention, dans ce cas, si `f3` et `f4` ne sont pas transposés, cette commande donne lieu à une erreur puisque ces deux vecteurs vont être concaténés en un seul vecteur ligne de taille différente de `x`. De plus, même si l'on enlève le `x` (`plot([f3 f4])`), le résultat ne sera pas celui souhaité : on aura en fait le tracé de `f4` à la suite de `f3`.

Exercice 9 : *résolution d'une équation du second degré*

Ecrire une fonction pour résoudre une équation du second degré, sur le modèle :

```
function [x1,x2]=resoud_eq_2d(a,b,c)
% calcul des racines de ax^2+bx+c = 0
% a, b et c peuvent être réels ou complexes et a doit être non nul
```

Corrigé :

```
function [x1,x2]=resoud_eq_2d(a,b,c)
if a==0
    error('a doit etre non nul')
end
delta=b^2-4*a*c;
x1=(-b+sqrt(delta))/(2*a);
x2=(-b-sqrt(delta))/(2*a);
```

La commande `error` permet de renvoyer un message d'erreur explicatif à l'utilisateur.

Exercice 10 : *création d'un signal*

On se propose d'écrire un script qui permette de simuler le signal de mesure fourni par un débitmètre. Ce signal est échantillonné à une fréquence de 100 Hz, pendant 12 secondes. On suppose qu'il est de moyenne nulle pendant les 4 premières secondes, puis de moyenne 10 lors des 2 secondes suivantes, de moyenne nulle lors des 3 secondes suivantes, puis de moyenne 15 pendant les 2 secondes suivantes, et enfin à nouveau de moyenne nulle

pendant la dernière seconde. Le bruit qui se superpose à la mesure est supposé gaussien, d'écart-type égal à 2.

a) Créer tout d'abord un vecteur `t` contenant la base de temps (de `0` à `12` s.).
b) Créer ensuite un vecteur `s` contenant le signal non-bruité.
c) Engendrer un vecteur `b` contenant les valeurs du bruit et effectuer les modifications nécessaires pour que ce bruit soit de moyenne nulle et de variance égale à 4. Vérifier les caractéristiques du bruit à l'aide des commandes `mean`, `std`, `var` et `hist`.
d) Créer enfin le vecteur `sb` par superposition de `s` et `b`.
e) Représenter le résultat en ayant soin de graduer l'axe des temps et de munir le graphe d'un titre, et d'un nom pour chacun des 2 axes.
f) Représenter sur un même graphe le signal bruité et l'original non-bruité, toujours en fonction du temps.

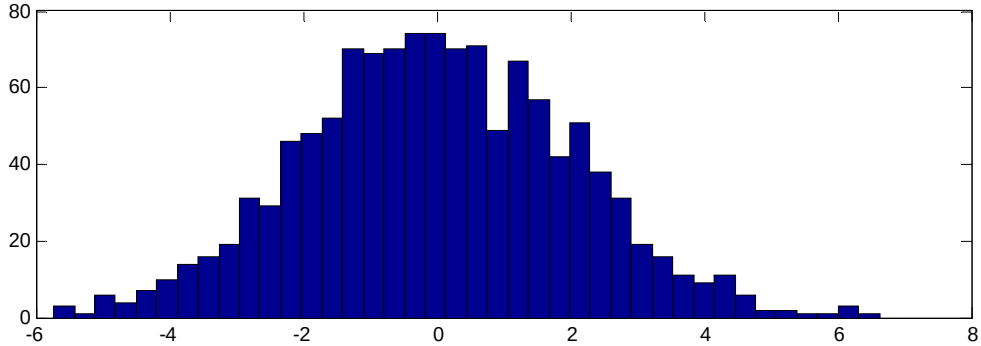
Conserver le contenu de l'espace de travail pour l'exercice suivant.

Corrigé :

a) `fe=100; Te=1/fe; t=0:Te:12;`
`Te` désigne la période d'échantillonnage, celle-ci joue le même rôle que le pas de maillage des exercices 7 et 8. La commande `length` permet de connaître la longueur du vecteur `t` créé : `length(t)` → `>> ans = 1201`

b) `s=[zeros(1,4*fe+1) 10*ones(1,2*fe) zeros(1,3*fe) 15*ones(1,2*fe) zeros(1,fe)];`
Le signal devant être de même dimension que le temps (1201), il faut rajouter un échantillon, on a choisi ici de le placer dans le premier segment.

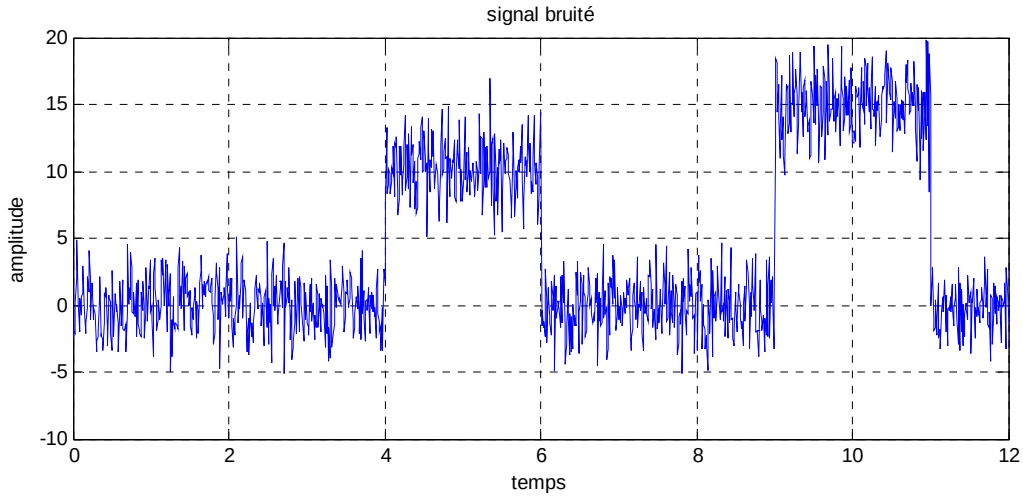
c) `b=randn(size(t));` La commande `randn` crée un vecteur aléatoire de distribution gaussienne, de moyenne à peu près nulle et de variance proche de 1. La commande `size` renvoie les dimensions du vecteur `t`. Une autre solution aurait consisté à écrire :
`b=randn(1,length(t));`
`b=b-mean(b);`
Cette commande permet de fixer la moyenne à une valeur la plus proche possible de zéro : `mean(b)` → `ans = 9.2442e-018`
`b=b/std(b);`
Cette instruction permet de fixer la variance (et l'écart-type) à 1 :
`std(b)` → `ans = 1` `var(b)` → `ans = 1`
`b=b*2;` A présent, l'écart-type est de 2 et la variance de 4 :
`std(b)` → `ans = 2` `var(b)` → `ans = 4`
`hist(b,40)`



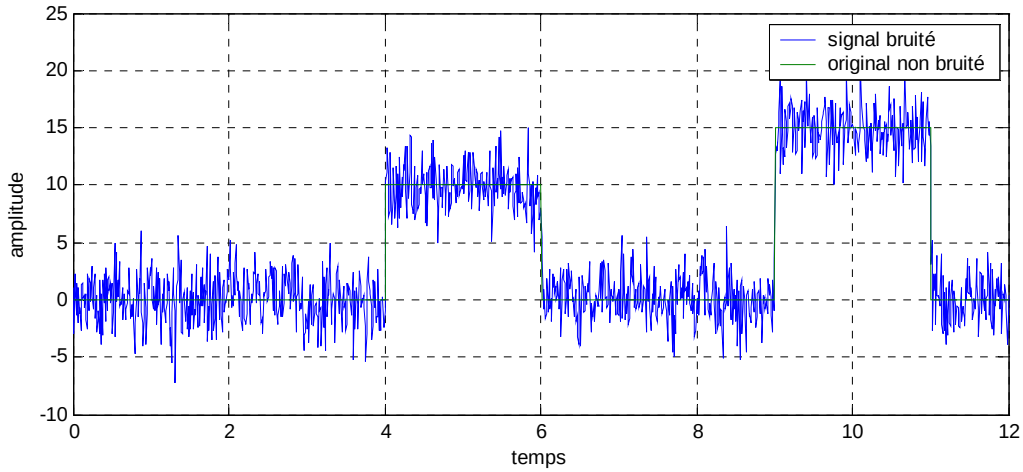
Cette dernière commande permet de vérifier que la distribution est gaussienne.

d) `sb=s+b;`

e)
```
plot(t,sb),grid,title('signal bruité')
xlabel('temps'), ylabel('amplitude')
```



f)
```
plot(t,[sb' s']),grid, xlabel('temps'), ylabel('amplitude')
legend('signal bruité','original non bruité')
```



Exercice 11 : *filtre moyenneur*

a) Soit l'estimateur de la moyenne pour un signal x de N échantillons : $\hat{m} = \frac{1}{N}\sum_{n=1}^{N} x(n)$.

On souhaite développer une version séquentielle de cet estimateur, c'est-à-dire qui fournisse une nouvelle estimation au fur et à mesure de l'acquisition des nouveaux échantillons du signal x(n). Etablir l'expression de $\hat{m}(n)$ puis de $\hat{m}(n+1)$ et en déduire la relation de récurrence entre les deux. Ecrire la fonction Matlab correspondante et la tester sur le signal de l'exercice précédent (argument d'entrée : le signal, argument de sortie : le vecteur des moyennes successives).

b) Proposer une modification de l'algorithme précédent qui permette d'obtenir un estimateur de moyenne sur fenêtre glissante de durée donnée L (donc sur les L derniers échantillons). Le paramètre L est bien sûr un argument d'entrée de la fonction. Tester la nouvelle fonction sur le signal de l'exercice précédent. Lors de l'initialisation de l'algorithme, on pourra utiliser la fonction précédente.

Corrigé :

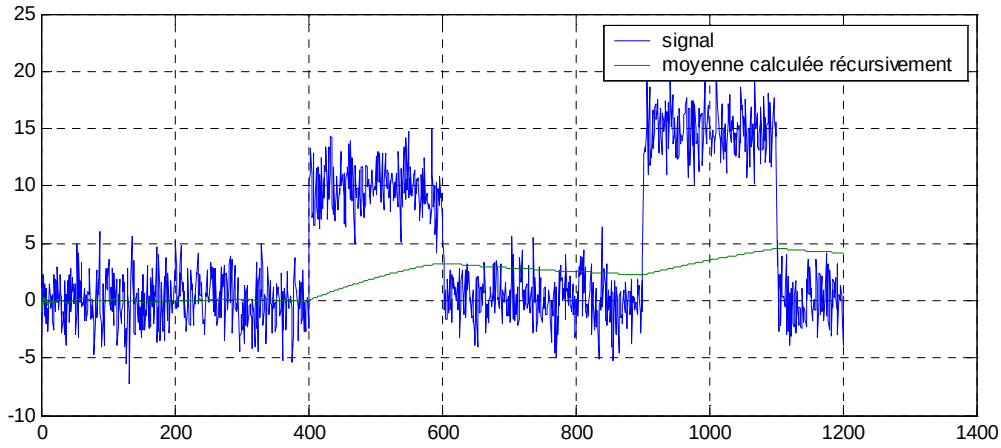
a) On a, de façon évidente : $\hat{m}(n)=\frac{1}{n}\sum_{i=1}^{n}x(i)$. De même, on a : $\hat{m}(n+1)=\frac{1}{n+1}\sum_{i=1}^{n+1}x(i)$.

Cette dernière quantité peut s'écrire :

$$\hat{m}(n+1)=\frac{x(n+1)}{n+1}+\frac{1}{n+1}\sum_{i=1}^{n}x(i)=\frac{x(n+1)+n.\hat{m}(n)}{n+1}$$

On en déduit la fonction Matlab :

```
function [m]=moyrec(x);
% moyenne récursive
[l,c]=size(x);
N=length(x);
if c~=N
    x=x';
end
m=[x(1) zeros(1,N-1)];
for n=1:N-1
    m(n+1)=(x(n+1)+n*m(n))/(n+1);
end
plot([x' m']), grid
legend('signal','moyenne calculée récursivement')
```



b) On a cette fois : $\hat{m}(n)=\frac{1}{L}\sum_{i=1}^{L}x(n-i+1)=\frac{1}{L}\sum_{i=1}^{L-1}x(n-i+1)+\frac{x(n-L+1)}{L}$. Or :

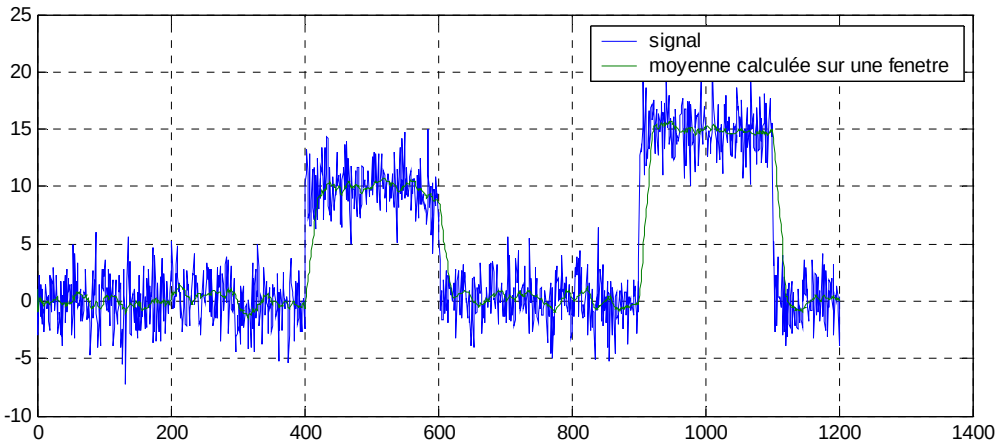
$$\hat{m}(n+1)=\frac{1}{L}\sum_{i=1}^{L}x(n-i+2)=\frac{1}{L}\sum_{i=2}^{L-1}x(n-i+2)+\frac{x(n+1)}{L}=\frac{1}{L}\sum_{i=1}^{L-1}x(n-i+1)+\frac{x(n+1)}{L}$$

$$=\hat{m}(n)+\frac{x(n+1)-x(n-L+1)}{L}.$$

D'où la fonction Matlab :

```
function [m]=moygliss(x,L);
% moyenne glissante sur fenetre de longueur L
[l,c]=size(x);
N=length(x);
if c~=N
    x=x';
end
```

```
m=[x(1) zeros(1,N-1)];
for n=1:L-1
    m(n+1)=(x(n+1)+n*m(n))/(n+1);
end
%m=[x(1:L) zeros(1,N-L)];
for n=L:N-1
    m(n+1)=m(n)+(x(n+1)-x(n-L+1))/L;
end
plot([x' m']),grid
legend('signal','moyenne calculée sur une fenetre')
```



Noter l'utilisation d'un calcul de moyenne récursive pour l'initialisation de l'algorithme (calcul des moyennes des L premiers points). Pour faire plus simple, on peut se contenter de fixer ces points à zéro.