

Introduction à Matlab

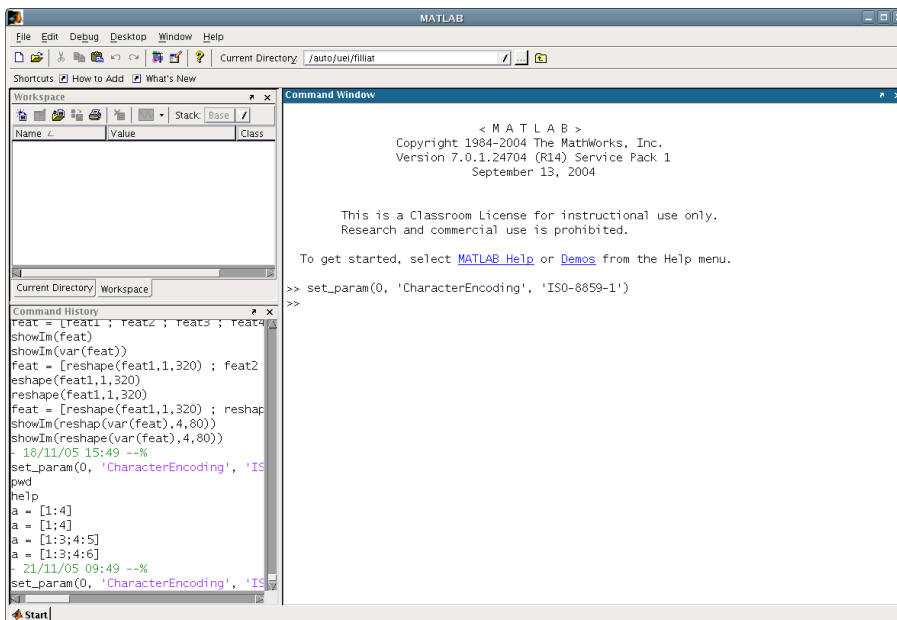
TD 01

david.filliat@ensta.fr

1 Lancement de Matlab

Matlab est un langage interprété, c'est à dire qu'il exécute directement (sans compilation) les commandes que vous entrez dans la *fenêtre de commandes*. Pour pouvoir l'utiliser, vous devez donc lancer l'interpréteur par la commande `matlab` ↵

L'application vous offre plusieurs fenêtres dont une fenêtre principale contenant la fenêtre de commandes avec le *prompt* : `>>`. C'est dans cette fenêtre que vous entrerez toutes les commandes matlab. Il est à noter que toutes les commandes sont en **minuscules** et en **anglais**. Lorsque l'on entre une commande, matlab affiche systématiquement le résultat de cette commande dans cette même fenêtre.



Le nombre de fonctions de matlab étant énorme, vous devrez utiliser l'aide quasiment en permanence. Deux méthodes sont possibles pour cela, soit en mode texte, soit via l'interface graphique. En mode texte, la commande `help` vous donne un aperçu des commandes disponibles :

```
>> help
HELP topics
```

matlab/general	- General purpose commands.
matlab/ops	- Operators and special characters.
matlab/lang	- Programming language constructs.
matlab/elmat	- Elementary matrices and matrix manipulation.
matlab/elfun	- Elementary math functions.
matlab/specfun	- Specialized math functions.
matlab/matfun	- Matrix functions - numerical linear algebra.
matlab/datafun	- Data analysis and Fourier transforms.
matlab/polyfun	- Interpolation and polynomials.
matlab/funfun	- Function functions and ODE solvers.
matlab/sparfun	- Sparse matrices.
matlab/scribe	- Annotation and Plot Editing.

```
matlab/graph2d      - Two dimensional graphs.
matlab/graph3d     - Three dimensional graphs.
....
```

Pour obtenir les informations concernant une section particulières, entrez `help section` :

```
>> help elfun
Elementary math functions.

Trigonometric.
sin      - Sine.
sind    - Sine of argument in degrees.
sinh    - Hyperbolic sine.
asin    - Inverse sine.
asind   - Inverse sine, result in degrees.
asinh   - Inverse hyperbolic sine.
cos     - Cosine.
cosd   - Cosine of argument in degrees.
cosh    - Hyperbolic cosine.
acos   - Inverse cosine.
acosd  - Inverse cosine, result in degrees.
...

```

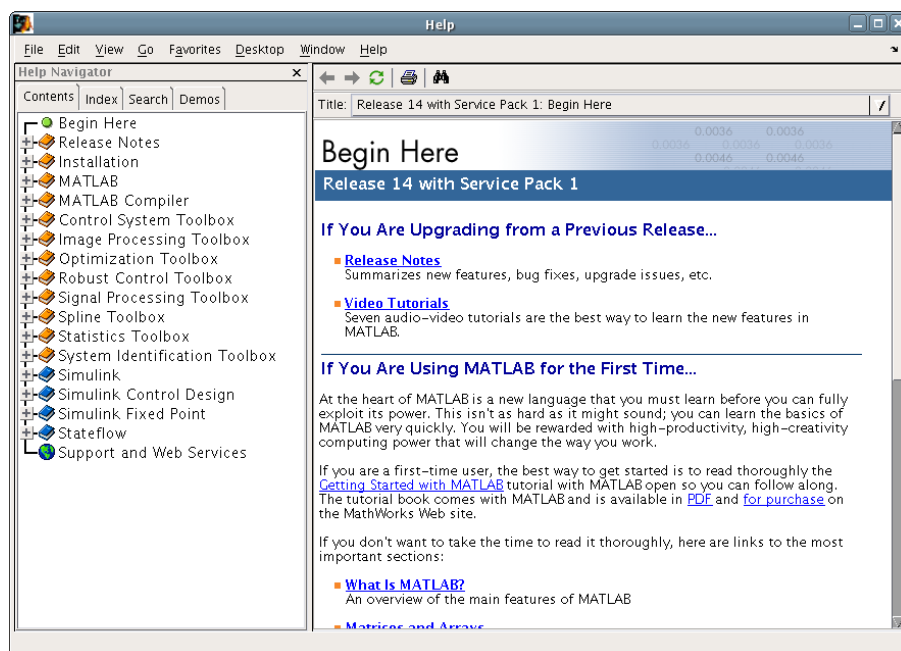
Pour avoir de l'aide directement sur une commande, entrez `help commande` :

```
>> help sin
SIN      Sine.
SIN(X) is the sine of the elements of X.
See also asin, sind.

Reference page in Help browser
doc sin

```

Vous pouvez également accéder à l'aide via l'interface graphique et ses onglets *index* et *recherche* :



Exercice 1 Trouvez dans l'aide le nom de la fonction renvoyant les valeurs propres (eigenvalue en anglais) d'une matrice.

2 Manipulation de variables

Matlab gère les nombres entiers, réels, complexes, les chaînes de caractères ainsi que les tableaux de nombres de façon transparente. Il est inutile de déclarer préalablement le type de la variable que l'on manipule, même pour les tableaux et les matrices, il suffit simplement d'assigner une valeur au nom de la variable avec l'instruction `=` :

```
>> a=10
a =
    10
```

La réponse à une commande de ce type est le nom de la variable ainsi que la valeur contenue dans cette variable. Toutes les variables utilisées restent présentes en mémoire et peuvent être rappelées.

La plupart des commandes que nous utilisons en matlab affectent des valeurs à des variables. Lorsque ce n'est pas le cas, le résultat de la commande est automatiquement affectée à la variable `ans` qui peut être par la suite utilisée comme une variable normale :

```
>> 10
ans =
    10
>> a = ans + 10
a =
    20
```

Attention cependant car une autre commande sans affectation écrasera l'ancienne valeur de la variable `ans`.

Matlab conservera en permanence en mémoire les variables que vous avez créées. Ces variables sont affichées dans la fenêtre *workspace* de l'interface graphique. La commande `who` en ligne de commande permet d'avoir la liste de ces variables en mode texte. La commande `clear all` permet de toutes les supprimer.

Enfin une troisième fenêtre contient l'*historique* des commandes. Il est possible de relancer ou modifier une ancienne commande en cliquant sur cette commande dans la fenêtre historique ou en tapant sur la flèche du haut dans la fenêtre de commande.

2.1 Scalaires

Le type de scalaire manipulé est transparent pour l'utilisateur. Ce type peut être entier, réel ou complexe :

```
>> a=1
a =
    1
>> b=1.02
b =
    1.0200
>> x=1.45e4
x =
    14500
>> c=1+2.4i
c =
    1.0000 + 2.4000i
```

la constante `i` est le nombre imaginaire prédéclaré, de même que certaines constantes (`e`, `pi`,...).

2.2 Vecteurs

Un vecteur ligne se déclare entre crochets en séparant les éléments avec des *espaces* ou des *virgules* :

```
>> v = [ 1 2 4 ]
v =
     1     2     4
>> w = [ 3, 4.6 , 1+3i ]
w =
 3.0000      4.6000      1.0000 + 3.0000i
```

Pour un vecteur colonne, le séparateur est le *point-virgule* :

```
>> z = [3;5;6]
z =
     3
     5
     6
```

Il est également possible d'utiliser l'opération de transposition *'* :

```
>> z = [3 5 6]
z =
     3
     5
     6
```

L'accès aux valeurs des vecteurs (pour les lire ou les écrire) se fait à l'aide des *parenthèses* pour indiquer l'élément souhaité dans le vecteur. Un indice en dehors d'un tableau entraîne un erreur :

```
>> v(1)
ans =
     1
>> v(2) = 7
v =
     1     7     4
>> v(6)
??? Index exceeds matrix dimensions.
```

Il existe des commandes pour créer des vecteurs de manière automatique. Par exemple *deux - points* permet de créer des séquences de nombres en indiquant en option l'intervalle entre ces nombres :

```
>> 1:4
ans =
     1     2     3     4
>> 1:1.5:6
ans =
 1.0000  2.5000  4.0000  5.5000
```

Exercice 2 Créez le vecteur [9 7 5 3 1]

Exercice 3 Créez le vecteur :

```
10.0000
 9.5000
 9.0000
 8.5000
 8.0000
```

2.3 Matrices

Les matrices se déclarent comme les vecteurs, en séparant les colonnes par *espace* et les lignes par *point- virgule* :

```
>> A = [ 1 3; 4 2]
A =
     1     3
     4     2
```

L'accès aux valeurs se fait grâce aux parenthèses en précisant d'abord **la ligne puis la colonne** :

```
>> A(2,1)
ans =
     4
```

Il existe différentes méthodes pour créer automatiquement des matrices. Il est possible comme pour les vecteurs d'utiliser la syntaxe `:` pour créer des suites de nombres en ligne. Il existe aussi des fonctions renvoyant une matrice identité de taille N : `eye(N)`, une matrice de 1 ou de 0 : `ones(N,M)` et `zeros(N,M)` (si l'on ne met pas M, une matrice carrée de taille N est renvoyée) :

```
>> eye(2)
ans =
     1     0
     0     1
>> ones(2,6)
ans =
     1     1     1     1     1     1
     1     1     1     1     1     1
```

Exercice 4 Créez la matrice :

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Enfin, il est possible de connaître la taille d'une matrice ou d'un vecteur avec la commande `size()`, qui retourne le nombre de lignes et de colonnes.

```
>> size(ans)
ans =
     2     6
```

Matrices creuses

Une matrice creuse est une matrice présentant un grand nombre d'éléments nuls qu'il n'est pas nécessaire de stocker, permettant de gagner à la fois de la place mémoire et du temps de calcul. Matlab gère de façon transparente les matrices creuses à l'aide de pointeurs que nous ne décrirons pas ici.

Contrairement aux variables classiques on doit déclarer explicitement le type `sparse` pour spécifier qu'une matrice est creuse. Lors de sa création une matrice est initialisée à zéro :

```
>> AC = sparse(1000,2000)
AC =
All zero sparse: 1000-by-2000
```

Il est ensuite possible de mettre des valeurs aux positions souhaitées. On remarquera que le résultat affiché comporte à la fois la position et la valeur des éléments non nuls :

```
>> AC(23,54)=1
AC =
(23,54)          1
```

Toutes les opérations standard sur les matrices s'appliquent aux matrices creuses. Pour des informations supplémentaires sur le type creux taper `help sparsfun`.

2.4 Chaînes de caractères

Les chaînes de caractères se manipulent comme des vecteurs. Elles sont déclarées avec des guillemets simples ' :

```
>> s='Hello'
s =
Hello
>> s(2)
ans =
e
```

3 Opérations élémentaires

3.1 Opérations mathématiques

Les opérations sur les scalaires sont standards : addition +, soustraction -, multiplication *, division /, puissance ^. La racine carrée s'obtient par la fonction `sqrt`. On dispose de toutes les fonctions usuelles sur les scalaires : faire `help elfun` pour de plus amples détails. Attention, les fonctions peuvent renvoyer des complexes :

```
>> sqrt(-1)
ans =
    0 + 1.0000i
```

En ce qui concerne les vecteurs et matrices ces opérateurs se prolongent **au sens du calcul vectoriel et matriciel**. En particulier, il faut veiller à la compatibilité des tailles des objets entre eux.

```
>> u=1:3
u =
     1     2     3

>> v = [1 0 - 1]
v =
     1     0    - 1

>> u+v
ans =
     2     2     2

>> v'
ans =
     1
     0
    - 1

>> u*v'
ans =
    - 2

>> v'*u
ans =
     1     2     3
     0     0     0
    - 1    - 2    - 3
```

Il est également possible de multiplier une matrice par un scalaire.

Exercice 5 Créez la matrice :

```
0 4 4 4
4 0 4 4
4 4 0 4
4 4 4 0
```

Attention : Pour les divisions de matrices, il faut faire attention au sens de la division. En matlab, il est possible de **diviser à gauche** avec la commande `\`. Par exemple, si $A*B=C$, on pourra écrire directement $B=A\C$. Attention, dans ce cas, éviter d'utiliser $B=inv(A)*C$ car la division à gauche permet de faire un grand nombre d'optimisations (par exemple dans le cas des matrices bloc-diagonales) que l'inversion de matrice simple ne fera pas.

Exercice 6 A l'aide de son écriture matricielle, résoudre le système :

$$2x + 3y + 4z = 3 \quad (1)$$

$$x - y - z = 0 \quad (2)$$

$$-x + 4y + z = 5 \quad (3)$$

On peut effectuer des opérations tensorielles (composante par composante) sur les vecteurs et matrices par l'adjonction d'un point à l'opérande :

```
>> u = 1:3
u =
    1     2     3

>> u.*u
ans =
    1     4     9

>> u.^3
ans =
    1     8    27
```

Cette fonctionnalité est particulièrement importante pour écrire des fonctions génériques qui fonctionneront de la même manière sur des scalaires et des vecteurs.

De même, toutes les fonctions scalaires peuvent s'utiliser sur des vecteurs ou des matrices :

```
>> h = 0:pi/4:pi
h =
    0    0.7854    1.5708    2.3562    3.1416

>> sin(h)
ans =
    0    0.7071    1.0000    0.7071    0.0000
```

Pour les nombreuses opérations sur les matrices (inverse, puissance, trace, déterminant, factorisation, ...) faire `help elmat` et `help matfun`.

3.2 Manipulations de variables

Pour créer et manipuler simplement des matrices ou des vecteurs, il est possible de concaténer des éléments en les mettant cote à cote dans un vecteur ou une matrice. **Attention** aux dimensions des objets :

```

>> A = eye(2)
A =
    1    0
    0    1

>> B = 2*ones(2)
B =
    2    2
    2    2

>> [A B]
ans =
    1    0    2    2
    0    1    2    2

>> [A;B]
ans =
    1    0
    0    1
    2    2
    2    2

```

Exercice 7 Créez la matrice :

```

1.0000    3.4000    0    0    0    0    5.0000
1.0000    0    3.4000    0    0    0    6.0000
1.0000    0    0    3.4000    0    0    7.0000
1.0000    0    0    0    3.4000    0    8.0000
1.0000    0    0    0    0    3.4000    9.0000

```

Il existe également une syntaxe utilisant *deux- points* pour extraire ou manipuler des parties de matrices comme des éléments. `:` spécifie une colonne ou ligne complète, `3 :5` signifie ligne ou colonne de 3 à 5 etc ... :

```

>> A=eye(5)
A =
    1    0    0    0    0
    0    1    0    0    0
    0    0    1    0    0
    0    0    0    1    0
    0    0    0    0    1

>> A(:,2)
ans =
    0
    1
    0
    0
    0

>> A(2:5,1:3)
ans =
    0    1    0
    0    0    1
    0    0    0
    0    0    0

```



```
>> A(1:2, :)=2
A =
     2     2     2     2     2
     2     2     2     2     2
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Exercice 8 Créez la matrice :

```
     1     0     7     0
     0     1     7     0
     0     0     7     0
     0     0     7     1
```

Exercice 9 Inversez les deux colonnes centrales dans la matrice précédente pour obtenir :

```
     1     7     0     0
     0     7     1     0
     0     7     0     0
     0     7     0     1
```

4 Programmation et utilisation des fonctions


4.1 Commandes d'environnement

L'utilisation de Matlab avec la simple ligne de commande comme nous venons de le voir est rapidement limitée. On utilise en général un mode de programmation qui consiste à écrire des scripts que Matlab pourra ensuite exécuter comme des commandes existantes.

Pour que Matlab retrouve vos scripts, il met à votre disposition plusieurs commandes d'environnement d'inspiration Unix :

- `path` : permet de savoir quels sont les dossiers auxquels Matlab a accès et de spécifier de nouveaux dossiers Unix où se trouvent vos ressources personnelles. Pour référencer un nouveau dossier, taper : `addpath /mesfichiersmatlab` ce qui indique à Matlab qu'il peut trouver des scripts dans le dossier `/mesfichiersmatlab` durant la session en cours.
- `cd` : positionne Matlab dans un dossier Unix, par exemple : `cd /mesfichiersmatlab`. **Matlab utilise en priorité les scripts se trouvant dans le dossier courant.**
- `dir` ou `ls` permet d'avoir la liste des fichiers du répertoire courant.

4.2 Scripts

Un script est un simple fichier texte avec l'extension `.m` qui contient une suite de commandes Matlab. Ce fichier peut être créé avec n'importe quel éditeur de texte et doit être placé dans le répertoire courant ou dans un répertoire du `path`. Matlab contient un éditeur de script intégré que vous pouvez utiliser en cliquant sur l'icône .

Les commandes à mettre dans les scripts sont les mêmes que celles que vous auriez mis en ligne de commande. Par défaut, le résultat de ces commandes s'affiche sur la fenêtre d'exécution, ce qui devient rapidement illisible. Il est possible de mettre un *point- virgule* en fin de ligne pour que la commande n'affiche rien.

Par exemple, créez un fichier `premierscript.m` dans le répertoire courant qui contient les lignes suivantes :

```
A=2*eye(4);
B=4*ones(4);
A*B
```

Vous pouvez ensuite exécuter ce script en entrant son nom sur la ligne de commande :

```
>> premierscript

ans =

     8     8     8     8
     8     8     8     8
     8     8     8     8
     8     8     8     8
```

Attention à ne pas donner à vos scripts des noms de commandes pré-définies.

Par ailleurs, il est très important d'ajouter des commentaires dans les scripts afin de les rendre plus lisibles par d'autres, ou par vous-même dans quelques semaines... Une ligne de commentaire commençant par % est ignorée dans les scripts.

4.3 Fonctions

Utilisé de cette manière, les scripts permettent simplement de mémoriser une suite de commande. Pour pouvoir faire des programmes modulaires, vous devez utiliser des fonctions. La syntaxe d'une fonction matlab est la suivante :

```
function [args1,args2,...] = nomfonction(arge1,arge2,...)
    instructions
```

args1,args2,... sont les arguments de sortie de la fonction et peuvent être de n'importe quel type

arge1,arge2,... sont les arguments d'entrée de la fonction et peuvent être de n'importe quel type

instructions est un bloc d'instructions quelconque devant affecter les arguments de sortie *args1,args2,...*

Lorsqu'il n'y a qu'un seul argument de sortie, on peut utiliser la syntaxe plus simple :

```
function args = nomfonction(arge1,arge2,...)
```

Pour appeler une fonction on utilise de code suivant :

```
[vars1,vars2,...] = nomfonction(vare1,vare2,...)
```

en faisant attention à la compatibilité des variables d'entrées *vare1,vare2,...* avec les arguments d'entrée *arge1,arge2,...*

Une fonction doit être enregistrée comme un script, dans un fichier *.m* du même nom.

Par exemple, un fichier *norme.m* contenant le code :

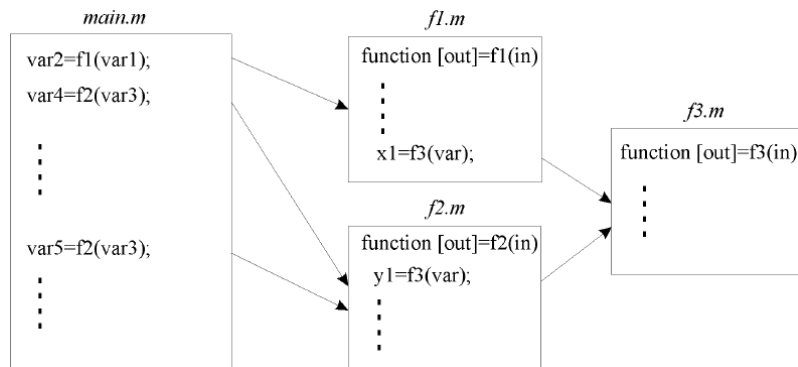
```
function n = norme(u)
n = sqrt(u*u');
```

permettra d'utiliser la fonction suivante :

```
>> norme([1 1])
ans =
    1.4142
```

Exercice 10 Écrire une fonction matlab *produits* prenant deux vecteurs lignes *u* et *v* en entrée et fournissant en sortie les deux produits $u*v'$ et $v'*u$

4.4 Structuration des programmes



Toute bonne programmation repose sur l'écriture d'un script principal qui fait appel à des fonctions autonomes. Cela permet :

- d'améliorer la lisibilité
- de tester indépendamment des parties de programmation
- d'augmenter le degré de généralité (utilisation d'une même fonction à divers endroits du programme, voire réutilisation des fonctions dans d'autres applications)

Attention à bien nommer les fichiers *.m* du même nom que la fonction qu'ils contiennent.

Par défaut, une variable n'est connue que dans le script dans lequel elle a été définie. En particulier, les variables du script principal ne sont pas connues dans les autres scripts. On doit donc transmettre en arguments d'une fonction toutes les variables dont on a besoin pour son exécution. Dans certains cas, on peut également utiliser la notion de variable globale qui permet de rendre visible des variables d'un script à l'autre. On la déclare comme globale dans le script principal ainsi que dans les scripts dans lesquels on désire l'utiliser à l'aide de la commande `global` :

```
global varg
```

4.5 Structures de contrôle

La programmation fait souvent usage de tests conditionnels ou de boucles. Toutes ces structures sont disponibles en matlab :

◆ Syntaxe du test (if)

```
if expression booléenne
    instructions
end
```

```
if expression booléenne
    instructions
else
    instructions
end
```

```
if expression booléenne
    instructions
elseif expression booléenne
    instructions
else
    instructions
end
```

◆ Syntaxe du branchement (switch)

```
switch expression (expression est un scalaire ou une chaîne de caractères)
    case value1 (instructions effectuées si expression=value1)
        instructions
    case value2
        instructions
    ...
    otherwise
        instructions
end
```

♦ Syntaxe de boucle (while et for)

```
while expression      for indice=debut:pas:fin  (si le pas n'est pas précisé, par défaut il vaut 1)
  instructions        instructions
end                   end
```

Pour sortir d'un test ou d'une boucle, on utilise la commande **break** (voir l'aide en ligne).

Ces structures font appel aux tests suivants :

<	strictement inférieur à
<=	inférieur ou égal à
>	strictement supérieur à
>=	supérieur ou égal à
==	égal à
~=	différent de
&	et logique (and)
	ou logique (or)
~	non logique (not)

Les résultats d'un test sous matlab sont 1 pour vrai et 0 pour faux :

```
>> (1>3) | (- 1~=0)
ans =
     1
```

Il existe d'autres fonctions booléennes, par exemple **xor**, **isfinite**, **isnan**, **isinf** ... dont on trouvera la description en faisant [help ops](#).

Attention cependant dans l'utilisation des boucles **for**. Ces boucles sont très inefficaces en Matlab et doivent être réservées au cas où on ne peut pas faire autrement. Il faut privilégier au maximum l'utilisation des fonctions vectorielles. Par exemple, pour appliquer une fonction **f** sur les entiers de 1 à 1000, ne **jamais** faire :

```
for i=1:1000
    X(i) = f(i);
end
```

mais utiliser :

```
t = 1:1000
X = f(t);
```

Exercice 11 Écrire une fonction *suite* qui prend un nombre N en argument et renvoie un vecteur ligne $[1 \dots N]$, si N est positif et un vecteur ligne $[-N \dots 0]$ si N est négatif.

Exercice 12 Écrire une fonction *insere* qui prend en entrée un nombre a et un vecteur X et qui renvoie en sortie le vecteur $[x_1 \ a \ x_2 \ a \ x_3 \ a \ \dots \ x_n]$

Références

Ce document est fortement basé sur un document de Patrick Ciarlet et Eric Lunéville :
<http://www.ensta.fr/~ciarlet/Doc-Matlab/Doc-Matlab-Couleur.pdf>.