CETS8AH - Introduction à MATLAB
TD 1

E. Kerrien

28 janvier 2019

Ce TP est une initiation au logiciel MATLAB; les exercices ne seront pas notés.
Les fichiers relatifs à ce cours sont disponibles sur `http://members.loria.fr/EKerrien/teaching/CETS8AH`.

Au début du TP, vous récupérerez le fichier `pts.txt` (`https://members.loria.fr/EKerrien/files/data/pts.txt`) et l'archive d'images `images.zip` (`https://members.loria.fr/EKerrien/files/data/images.zip`).

Vous créerez un répertoire pour ce TP dans lequel vous mettrez le fichier `pts.txt` et décomprimerez l'archive d'images. Vous vous placerez dans ce répertoire pour réaliser ce TP (voir le bandeau de répertoire dans la partie supérieure, ainsi que le *Current folder* dans la colonne de gauche, voir la fenêtre MATLAB sur la figure 1).

Vous n'arriverez peut-être pas à tout faire pendant le temps du TP. Les exercices, s'ils sont nombreux, ne sont cependant pas difficiles. Aussi, prenez le temps de tester toutes les commandes indiquées car cela vous permettra d'être bien plus efficaces lors des TP suivants qui seront plus orientés analyse d'image. N'hésitez pas à me contacter par mail à `Erwan.Kerrien@inria.fr` pour toute question que vous pourrez avoir par la suite.

1 Manipulations de base

1.1 Introduction

MATLAB est une abréviation de MATrix LABoratory. C'est un logiciel de calcul numérique très convivial qui ne demande que peu de compétences en programmation. Comme son nom l'indique, il est très efficace dans la manipulation de vecteurs et de matrices. La figure 1 montre la fenêtre MATLAB tel qu'elle se présente à son lancement (version R2018a).
Deux choses importantes à savoir:

- L'aide en ligne est très complète et très bien faite. Pour y accéder: via le menu 'Resources'→'Help' (en haut à droite, voir figure 1), le champ d'aide (tout en haut à droite), la touche 'F1' ou les commandes `help [<commande>]`, `doc [<commande>]` et `lookfor <mot-clé>`. L'aide de MATLAB est également disponible sur internet : `https://fr.mathworks.com/help/matlab/index.html`. D'expérience, c'est ce dernier moyen qui est à privilégier[1].

- la philosophie du logiciel est de manipuler des vecteurs et des matrices, là où un langage de programmation "classique" ne manipule que des scalaires. De ce fait, les boucles sont à éviter autant que possible: il n'est pas forcément immédiat de maîtriser ce mode de pensée mais c'est indispensable pour être efficace.

Dans tout le texte du TP, il vous est demandé de taper les commandes données en exemple. Ces commandes sont à entrer dans la *console* : c'est la partie nommée 'Command window' (partie basse au centre, voir figure 1). Elles suivent les >> et l'exemple indique toujours la réponse de MATLAB. >> est une invite de commande dont l'affichage indique que MATLAB a fini de traiter la commande précédente et est ainsi prêt à recevoir la suivante.

Comme tout langage de programmation, MATLAB utilise des variables pour stocker des valeurs sous forme principalement de scalaires, de vecteurs, ou de matrices (d'autres formes, telles des structures existent aussi). Ces variables sont identifiées par un nom choisi par le programmeur. Ce nom est une chaîne de caractères, en général composée de lettres, mais des chiffres peuvent également être employés, sauf en première position. Par exemple :

[1] MATLAB souffre parfois de lenteurs pour accéder à l'aide interne.

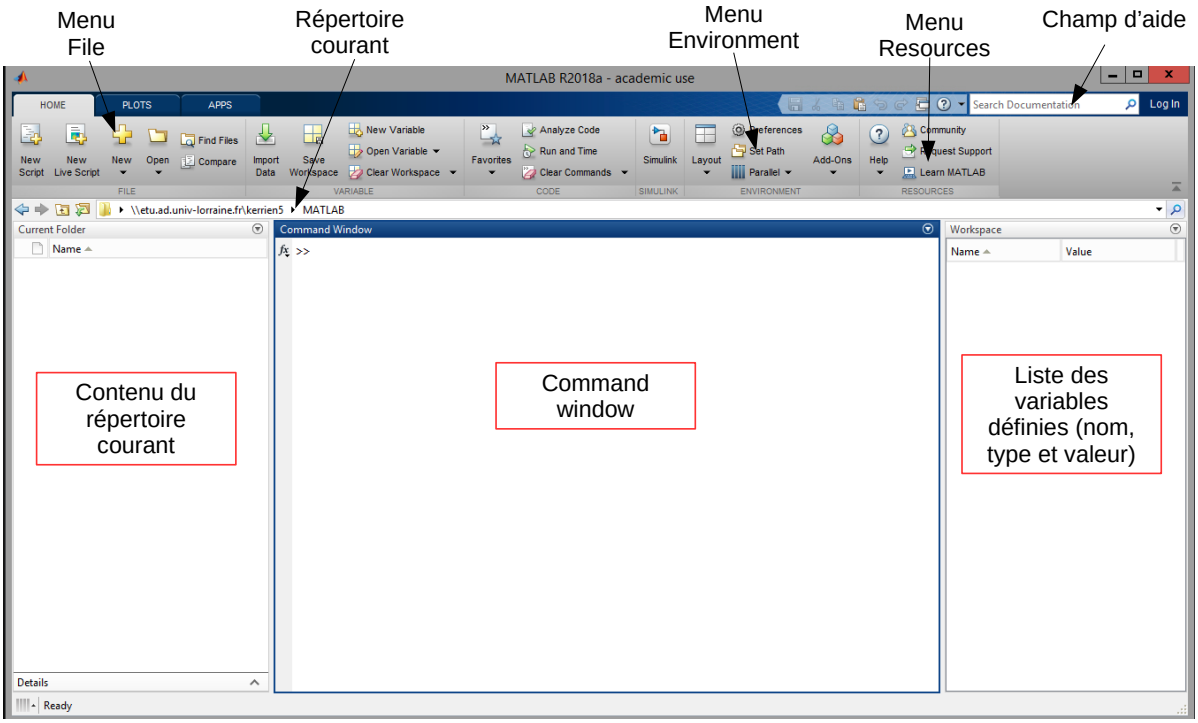

Figure 1: Fenêtre MATLAB au lancement de l'application (version R2018a)

```
>> A1=1
A1 =
     1
```

est une expression valable qui stocke le scalaire 1 dans la variable `A1`. Mais l'expression

```
>> 1A=1
 1A=1
  ^
Error: Invalid expression. Check for missing multiplication operator, missing or
unbalanced delimiters, or other syntax error. To construct matrices, use brackets
instead of parentheses.
```

conduit à une erreur (`Error: Invalid expression.`). À noter également que les variables nommées `A1` et `a1` sont différentes.

Comme dit plus haut, si vous ne voulez pas avoir de retour d'affichage de la première commande, vous pouvez compléter cette commande par un `;`

```
>> A1=1;
```

Et si plus tard vous voulez afficher la valeur stockée dans la variable `A1`, il vous suffit d'entrer le nom de cette variable, sans `;` final

```
>> A1
A1 =
     1
```

La commande `whos` permet à tout instant d'avoir la liste des variables avec leur type (scalaire, vecteur, matrice, etc.). La liste des variables est également disponibles dans la fenêtre 'Workspace' (partie basse à droite, voir figure 1). Vous pouvez choisir les colonnes à afficher : "Class" pour le type, "Bytes" pour la taille occupée en mémoire, etc.

Vous verrez à l'occasion apparaître une variable que vous n'aurez pas définie. Il s'agit de la variable nommée `ans`. Elle est créée automatiquement dès lors que vous tapez une expression qui renvoie une valeur que vous ne stockez pas explicitement dans une variable. Par exemple, si vous tapez :

```
>> 2*3
```

vous verrez comme résultat

```
ans =
      6
```

Enfin, si vous voulez libérer de la mémoire en vous débarrassant d'une variable devenue inutile, la commande `clear` vous permet de le faire :

```
>> clear ans
```

1.2 Scalaires

1.2.1 Types numériques

Le scalaire est l'entité la plus simple à manipuler : il s'agit d'un nombre, soit entier, soit réel. MATLAB gère également les nombres complexes. Selon son *type*, un scalaire peut être stocké en mémoire sous des formes différentes. Chaque type requiert un certain nombre d'octets pour être stocké. Les types numériques connus de MATLAB sont listés dans l'aide: 'MATLAB'→'Language Fundamentals'→'Data Types'→'Numeric Types'. Dans le Workspace, le type des variables est indiqué dans la colonne 'Class'.

Si on prend l'exemple des valeurs des pixels d'une image, celles-ci sont en général comprises entre 0 et 255. On peut donc stocker chacune d'entre elles sur un seul octet, alors qu'il en faudrait 8 pour les stocker sous forme de réels. Cependant, cela demande de prendre certaines précautions lors des manipulations algébriques. En effet, 200 peut se stocker sur un seul octet, mais 200+200 requiert au moins 2 octets. Voici comment MATLAB gère ce cas :

```
>> B=uint8(200)
B =
  uint8
   200
>> B+B
ans =
  uint8
   255
```

La variable `B` est de classe `uint8` (valeur entière non signée codée sur un octet, c'est-à-dire un nombre compris entre 0 et 255) et de valeur 200. `B+B` est donc aussi de type `uint8` et ne peut donc excéder 255. On voit que le résultat est faux : si la valeur à stocker dépasse 255, alors c'est 255 qui est stocké.

De même si la résultat est plus petit que 0, alors c'est 0 qui est renvoyé :

```
>> C=uint8(201)
C =
  uint8
   201
>> B-C
ans =
  uint8
   0
```

Une solution pour éviter de nombreux problèmes consiste donc à convertir (*caster*) le type de toutes les variables numériques en réels grande précision c'est-à-dire un `double` :

```
>> Bd=double(B)
Bd =
   200
>> Bd+Bd
ans =
   400
```

`Bd` est un cast de `B` en double et vaut donc 200, mais compris comme un réel grande précision, codé sur 8 octets. La nombre maximal représentable en double est bien plus grand et `Bd+Bd` est donc tel qu'on pourrait s'y attendre. Il est également possible de faire `B=double(B)` pour changer le type de `B` sans définir de nouvelle variable (mais l'ancienne variable `B` de type `uint8` est écrasée par la nouvelle `B` de type `double`). Vous pouvez remarquer dans le Workspace que la variable `A1` définie en début de TP à la valeur 1 est par défaut de type `double`.

1.2.2 Opérations

Les opérateurs usuels sont disponibles pour les scalaires:

- `+` est l'addition. Ex: `A1+Bd` renvoie la somme de `A1` et `Bd`

- `-` est la soustraction. Ex: `A1-Bd`
- `*` est la multiplication. Ex: `A1*Bd`
- `/` est la division. Ex: `A1/Bd`. À noter que si `Bd` vaut 0 et `A1` est non nul, le résultat de cette opération est un scalaire particulier `Inf` (*infinite*, `-Inf` si `A1` est négatif). Si `A1` est également nul, alors le résultat sera un autre scalaire particulier `NaN` (*Not a Number*).

1.3 Vecteurs et matrices

Un vecteur peut être vu comme une matrice dont une dimension est 1. Nous ne parlerons donc que de matrices ici.

1.3.1 Déclaration - création d'une matrice

Une matrice est un tableau bi-dimensionnel de scalaires, tous de même type. Par conséquent, une matrice a également un type. Une matrice se définit en encadrant les valeurs par des crochets `[]`. Les lignes sont données successivement, et sont séparées par des `;`. Au sein de chaque ligne, un espace ou une virgule sépare les valeurs. Exemple :

```
>> M=[1 2 3 4;5 6 7 8;9 10 11 12]
M =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

`M` est une matrice de 3 lignes et 4 colonnes.

Un vecteur ligne sera donc créé par la commande :

```
>> vL=[1 2 3 4 5]
vL =
     1     2     3     4     5
```

et un vecteur colonne sera créé par la commande :

```
>> vC=[1;2;3;4;5]
vC =
     1
     2
     3
     4
     5
```

On peut connaître la taille d'une matrice par la commande `size`

```
>> size(M)
ans =
     3     4
>> size(vL)
ans =
     1     5
>> size(vC)
ans =
     5     1
```

Où on voit bien qu'un vecteur est en fait une matrice dont une dimension vaut 1.

On peut extraire la taille d'une matrice selon une seule dimension grâce à un argument supplémentaire de `size`. Ainsi, nous avons:

```
>> size(M,1)
ans =
     3
>> size(M,2)
ans =
     4
```

Pour un vecteur, on peut utiliser également la commande `length`

```
>> length(vL)
ans =
     5
>> length(vC)
ans =
     5
```

En fait cette commande calcule la dimension maximale d'une matrice. Ainsi :

```
>> length(M)
ans =
     4
```

Les commandes `ones`, `zeros` et `eye` peuvent aussi servir à initialiser une matrice, avec respectivement uniquement des 1, uniquement des 0, et uniquement des 0 sauf pour les éléments diagonaux mis à 1

```
>> ones(1,3)
ans =
     1     1     1
>> zeros(2)
ans =
     0     0
     0     0
>> eye(2,3)
ans =
     1     0     0
     0     1     0
```

Enfin, on peut définir un vecteur comme une suite arithmétique par la syntaxe suivante: `ElemMin:pas:ElemMax`. Exemples:

```
>> L=1:5
L =
     1     2     3     4     5
>> L2=0:0.3:2
L2 =
  Columns 1 through 6
                  0   0.300000000000000   0.600000000000000   0.900000000000000   1.200000000000000
  Column 7
   1.800000000000000
```

On peut remarquer que les bornes sont inclues :

```
>> L3=0:0.2:2
L3 =
  Columns 1 through 6
                  0   0.200000000000000   0.400000000000000   0.600000000000000   0.800000000000000
  Columns 7 through 11
   1.200000000000000   1.400000000000000   1.600000000000000   1.800000000000000   2.000000000000000
```

À noter que si vous voulez un format d'affichage avec moins de zéros (mais aussi moins de précision), vous pouvez aller modifier le 'Numeric format' dans 'Environment'→'Preferences'→'Command window'. Ceci ne change que l'affichage et en rien la précision des calculs.

1.3.2 Opérations sur les matrices

L'élément de la ligne `i` et la colonne `j` se note avec des parenthèses :

```
>> M(2,3)
ans =
     7
```

Pour le cas spécial du vecteur, on peut n'utiliser qu'un seul indice:

```
>> vL(2)
ans =
     2
>> vC(3)
ans =
     3
```

Vous noterez que les indices commencent à 1. Le premier élément d'un vecteur est donc `v(1)` et le dernier est `v(length(v))`.

On peut également extraire une ligne d'une matrice :

```
>> M(2,:)
ans =
     5     6     7     8
```

De même une colonne :

```
>> M(:,3)
ans =
     3
     7
    11
```

Ou seulement une sous-matrice :

```
>> M(1:2,2:4)
ans =
     2     3     4
     6     7     8
```

Ces expressions permettent aussi de modifier une sous-partie d'une matrice :

```
>> M(2,:)=11:14
M =
     1     2     3     4
    11    12    13    14
     9    10    11    12
>> M(3,:)=3
M =
     1     2     3     4
    11    12    13    14
     3     3     3     3
```

Vous remarquerez que dans le dernier cas, on peut également réinitialiser tous les éléments d'une matrice ou d'un vecteur à une constante.

Enfin, on peut afficher une matrice sous forme d'un vecteur colonne par la commande :

```
>> M(:)
ans =
     1
    11
     3
     2
    12
     3
     3
    13
     3
     4
    14
     3
```

Vous pouvez remarquer que ce sont les colonnes qui sont concaténées. Pour des opérations plus compliquées, vous pouvez vous référer à la commande `reshape`.

La transposition d'une matrice se note par une apostrophe :

```
>> M'
ans =
     1    11     3
     2    12     3
     3    13     3
     4    14     3

>> vC'
ans =
     1     2     3     4     5
```

L'addition et la soustraction, ainsi que la multiplication par un scalaire ont le sens usuel. Il faut simplement prêter attention aux dimensions des opérandes :

```
>> 2*M
ans =
     2     4     6     8
    22    24    26    28
     6     6     6     6
>> vL-vC'
ans =
     0     0     0     0     0
>> vL+1
ans =
     2     3     4     5     6
>> vL-vC
ans =
     0     1     2     3     4
    -1     0     1     2     3
    -2    -1     0     1     2
    -3    -2    -1     0     1
    -4    -3    -2    -1     0
>> vC-vL
ans =
     0    -1    -2    -3    -4
     1     0    -1    -2    -3
     2     1     0    -1    -2
     3     2     1     0    -1
     4     3     2     1     0
```

Les deux derniers cas étaient gérés par une erreur dans les anciennes versions de MATLAB. Le choix des dernières versions a été de gérer "intelligemment" ces cas pour éviter des erreurs[2] : on voit que l'opération est effectuée terme à terme. Ce comportement peut cacher des erreurs si vous n'êtes pas attentifs aux dimensions, mais d'un autre côté, le déboggage par erreurs est à proscrire !

Dans tous les cas, prenez le temps de lire les messages d'erreur remontés par MATLAB : ils sont souvent de bons conseils.

La multiplication est au sens matriciel, et il faut encore faire attention aux dimensions des opérandes :

```
>> ones(3,5)*vC
ans =
    15
    15
    15
>> vL*ones(5,3)
ans =
    15    15    15
>> vL*vC
ans =
    55
>> vC*vL
ans =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25
>> M*vC
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in the
first matrix matches the number of rows in the second matrix. To perform elementwise
multiplication, use '.*'.
```

[2]Ce que MATLAB appelle "expansion implicite" et qu'on connaît en général sous le terme "broadcasting" (notamment en Python avec numpy).

La division prend un sens particulier: `A/B` équivaut à `A*INV(B)`. Voir l'aide de la fonction `mrdivide`.

Il est également possible de faire une multiplication, une division, une mise à la puissance... membre à membre en faisant précéder l'opérateur par un `.` :

```
>> vL.*vC'
ans =
     1     4     9    16    25
>> M.^2
ans =
     1     4     9    16
   121   144   169   196
     9     9     9
```

Les commandes `sum` et `mean` permettent de respectivement renvoyer la somme et la moyenne des éléments selon la première dimension différente de 1. Vous pouvez spécifier la dimension selon laquelle vous voulez opérer grâce à un second argument :

```
>> sum(M)
ans =
    15    17    19    21
>> sum(M,2)
ans =
    10
    50
    12
```

Si vous voulez faire une somme de tous les termes d'une matrice (non plus sur une seule dimension mais bien sur les deux), vous pouvez utiliser:

```
>> sum(M(:))
ans =
   72
```

Les commandes `max` et `min` permettent de respectivement renvoyer la valeur maximale et minimale selon la première dimension différente de 1. Lisez l'aide de ces fonctions pour savoir comment spécifier la dimension de travail, prendre le maximum de deux matrices, ou récupérer le tableau des indices des éléments maximaux.

La commande `find` permet de renvoyer les indices des éléments vérifiant la condition placée en paramètre (ici tous les éléments de `M` plus grand que 3) :

```
>> find(M>3)
ans =
     2
     5
     8
    10
    11
```

Vous remarquerez que les indices renvoyés sont linéaires, c'est-à-dire qu'on va de 1 à 3 pour la première colonne, puis de 4 à 6 pour la deuxième, etc. En d'autres termes, ce sont les indices par rapport au vecteur `M(:)`.

On peut simplifier cette fonctionnalité quand il s'agit de sélectionner certains éléments pour modification:

```
>> M(M>3) = 0
M =
     1     2     3     0
     0     0     0     0
     3     3     3     3
```

Enfin, on peut concaténer deux matrices dans le sens horizontal en les séparant par un espace ou dans le sens vertical en les séparant par un `;` :

```
>> [M M]
ans =
     1     2     3     0     1     2     3     0
     0     0     0     0     0     0     0     0
```

```
     3     3     3     3     3     3     3     3
>> [M;M]
ans =
     1     2     3     0
     0     0     0     0
     3     3     3     3
     1     2     0     0
     0     0     0     0
     3     3     3     3
```

On peut également effacer tout une ligne ou une colonne d'une matrice de la manière suivante :

```
>> M(:,3) = []
M =
     1     2     0
     0     0     0
     3     3     3
>> M(2,:)=[]
M =
     1     2     0
     3     3     3
```

1.4 Lecture/écriture

Pour cet exercice, vous aurez besoin du fichier `pts.txt` qui se trouve sur `https://members.loria.fr/EKerrien/files/data/pts.txt`.

Placez-vous dans le répertoire qui contient ce fichier : soit par la commande 'cd', soit par la barre de répertoire courant, soit en naviguant dans la partie 'Current Folder'.

La commande `load` permet de lire un fichier structuré de données au format ASCII (par exemple un fichier de points) et stocker le résultat dans une matrice:

```
>> P=load('pts.txt')
P =
   100   100
   200   300
   300   200
   400   400
```

La commande `save` permet de sauvegarder le contenu d'une variable matrice M au format ASCII:

```
>> save('matriceM.txt','M','-ascii')
```

MATLAB offre des fonctionnalités de lecture/écriture plus complexes. Pour cela, voyez l'aide des fonctions `fprintf` et `fscanf` qui ne poseront aucun problème à ceux d'entre vous qui connaissent déjà le langage C.

Il existe des fonctions spécifiques de lecture/écriture d'image. Nous les verrons en fin de TP.

1.5 Graphiques

La commande `plot` permet d'afficher dans un graphique des données définies par des couples de points (x_i, y_i). Exemple:

```
>> tabx=[100 200 300 400]
tabx =
   100   200   300   400
>> taby=[100 300 200 400]
taby =
   100   300   200   400
>> plot(tabx,taby)
```

La figure 2 vous montre le résultat de cette commande. Vous remarquerez que par défaut `plot` relie les points par des segments bleus, formant une ligne brisée. Vous obtiendrez le même résultat en utilisant les points `P` chargés précédemment, grâce à la commande :

```
>> plot(P(:,1),P(:,2))
```

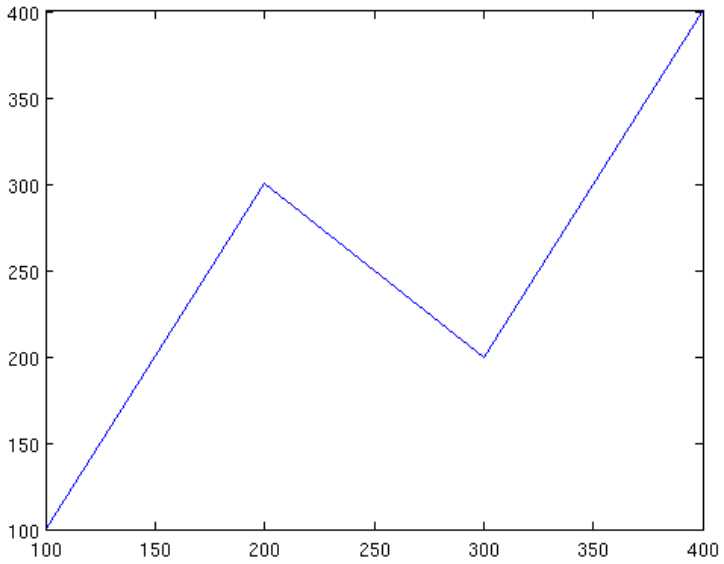

Figure 2: Résultat d'une commande `plot`

Notez que vous pouvez intégrer la fenêtre libre ouverte par `plot` dans la fenêtre MATLAB en cliquant sur la petite flèche noire en haut, tout à droite du bandeau des menus de la fenêtre `plot`.

Il est possible de varier les affichages. Pour cela, consultez l'aide en ligne pour la commande `plot`. Par exemple, la commande suivante :

```
>> figure, plot(P(:,1),P(:,2),'r+')
```

dessine les points de manière individuelle sous forme de croix rouges. La mention de `figure,` avant la commande plot demande à MATLAB d'ouvrir une nouvelle fenêtre pour faire ce nouveau dessin. Sinon, par défaut, le nouveau dessin utilise la dernière fenêtre utilisée et écrase par conséquent l'ancien dessin. Voir l'aide en ligne sur la commande `figure` pour savoir comment sélectionner la fenêtre ciblée pour le dessin. Voir également les commandes `hold` et `subplot`.

MATLAB n'est pas un logiciel de mathématiques formelles. Il ne sait donc pas dessiner le graphe d'une fonction. Pour ce faire, il faut créer un couple de tableaux: un tableau d'abscisses pour les valeurs de la variables `x` et un tableau d'ordonnées pour les valeurs correspondantes de la fonction. Ainsi pour dessiner le graphe de la fonction $f(x) = x^2$ pour $x \in [-4, -3.9, \ldots 3.9, 4]$, vous pouvez faire (remarquez ici l'intérêt du `;` final...) :

```
>> x=-4:0.1:4;
>> plot(x,x.^2)
```

2 Fonctions

Chaque commande MATLAB correspond en fait à une fonction MATLAB. Une fonction est identifiée par un nom, comme une variable, et prend en entrée un certain nombre de paramètres pour renvoyer en sortie potentiellement plusieurs valeurs.

2.1 Ecrire une fonction

MATLAB dispose d'un éditeur de texte spécifiquement dédié à la programmation et en particulier l'écriture de fonctions. Pour le lancer, aller dans 'File'→'New'→'Function' ('Open' ouvre un fichier existant). C'est dans cet éditeur que vous aller pouvoir écrire votre fonction. Lors de la sauvegarde, MATLAB reconnaîtra le nom que vous avez donné à votre fonction et vous proposera un nom de fichier adéquat pour qu'elle puisse être ensuite appelée dans la console MATLAB.

Une fonction de nom `funname` prenant en paramètre `in1`, `in2`, ... et retournant les paramètres `out1`, `out2`, ... se définit ainsi

```
function [out1, out2, ...] = funname(in1, in2, ...)
<CORPS DE LA FONCTION>
end
```

Cette structure de base d'une fonction vous est d'emblée proposée par l'éditeur. Le `end` final n'est pas obligatoire mais est fortement conseillé pour éviter des problèmes dans certains cas un peu tordus.

Exemple (dans l'éditeur):

```
function [mean, stdev] = stat(x)
  n=length(x);
  mean=sum(x)/n;
  stdev=sqrt(sum((x-mean).^2)/n);
end
```

Appel de la fonction (dans la console):

```
>> v=1:10;
>> [m,s]=stat(v)
m =
   5.500000000000000
s =
   2.872281323269014
```

Vous remarquerez encore ici l'intérêt des `;` en fin de commandes pour éviter que MATLAB vous envoie de nombreux messages inutiles lors de l'appel de la fonction. À noter aussi que pour renvoyer les variables notées en sortie, il suffit de modifier leur valeur dans le corps de la fonction. Il n'y a pas d'instruction `return` comme c'est le cas dans d'autres langages. Enfin, vous remarquerez qu'il est tout à fait possible de renvoyer plusieurs valeurs dans une fonction.

2.2 Structures de contrôle

Les structures de contrôle permettent de contrôler le flux de traitements dans un programme. Les deux genres de flux les plus utilisés sont les boucles qui permettent d'effectuer le même traitement sur un ensemble de valeurs et les expressions conditionnelles qui permettent de moduler le traitement à effectuer en fonction d'un test.

Boucle `for`

Syntaxe:

```
for variable = expression
    statements
end
```

Exemple (boucles imbriquées):

```
>> k=3;
>> for m=1:k
for n=1:k
a(m,n)=m+n;
end
end
>> a
a =
     2     3     4
     3     4     5
     4     5     6
```

Vous remarquerez dans cet exemple que d'une part la boucle `for` travaille sur un vecteur de valeurs (`1:k` définit le vecteur `[1 ... k]`, et d'autre part il n'est pas nécessaire d'initialiser la matrice `a` avant de pouvoir accéder à ses éléments : elle est créée au vol.

Conditionnelle `if`

Syntaxe:

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

Exemple:

```
>> seuil=4;
>> v=3;
>> if v> seuil
value=2;
elseif v<seuil
value=1;
else value=0;
end
>> value
value =
     1
```

Il est ici intéressant de s'intéresser aux tests :

- `a > b` (resp. `a < b` renvoie `vrai` si `a` est plus grand (resp. plus petit) que `b`, `faux` sinon. Si `a` et `b` sont des matrices de même taille, le test est fait élément par élément et renvoie donc une matrice booléenne de même taille, où `vrai` est codé par un 1 et `faux` est codé par un 0. Ceci est vrai pour tous les tests suivants.
- l'inégalité au sens large est notée `a >= b` et `a <= b`
- l'égalité stricte est notée (voir ci-dessus) `a == b` (si vous ne mettez qu'un =, alors c'est une affectation !)
- la différence est notée `a ~= b`
- il existe aussi des tests de type comme par exemple `isscalar(a)` pour vérifier si la variable `a` est de type scalaire.
- enfin, comme noté plus haut, les tests, quand il s'agit de matrices, renvoient une matrice. Or la commande `if` demande un seul booléen. Pour tester l'égalité de deux matrices et renvoyer un seul booléen, vous devez utiliser la commande `isequal`.

Vous trouverez toute une batterie de tests dans le navigateur d'aide en cherchant `is*`.

2.3 Le *path*

Quand une fonction, par exemple `funname`, est appelée dans une commande, MATLAB recherche un fichier nommé `funname.m` (pour la fonction `stat` ci-dessous, ce sera le fichier `stat.m`). Ce genre de fichier s'appelle un *m-file*. MATLAB effectue la recherche d'un tel fichier dans une liste de répertoires stockée dans le chemin de recherche appelé *path*. Le *path* est visible, et modifiable, par le menu de MATLAB 'Environment'→'Set Path'.

Un conseil : lors de chaque TP, créez un répertoire de travail et ajoutez-le au *path* MATLAB en utilisant l'interface 'Set Path'. Ainsi, toutes les fonctions (*m-files*) que vous créerez seront accessibles depuis l'interface MATLAB.

Et si MATLAB ne trouve pas une fonction que vous venez d'écrire, vérifiez bien votre *path*.

Par défaut, MATLAB cherche également le répertoire courant. Pour connaître le répertore courant, vous avez la commande `pwd`, mais il est également indiqué dans le bandeau entre les menus et les fenêtres. Vous pouvez changer de répertoire courant de manière intuitive dans l'interface.

3 Manipulation d'images

Nous allons finir ce TP par quelques manipulations d'images. Pour ce faire, il vous faut avoir récupéré le fichier `https://members.loria.fr/EKerrien/files/data/images.zip` et l'avoir décomprimé dans votre répertoire de TP. Les images sont dans un sous-répertoire appélé `images`. Les commandes spécifiques à connaître ici sont :

- `imread` : qui prend en paramètre un nom de fichier, encadré par des apostrophes (') et renvoie l'image sous forme de matrice. Attention, bien souvent la matrice renvoyée n'est pas de type `double`, ce qui peut poser des problèmes pour les calculs suivants. Nous allons bientôt voir cela.
- `imwrite` : qui prend en paramètre une matrice (l'image à sauvegarder), un nom de fichier (là où l'image doit être écrite) et une chaîne de caractères indiquant le format sous lequel l'image est sauvegardée.
- `imshow` : qui prend en paramètre une matrice et l'affiche sous forme d'image. Les valeurs contenues dans l'image sont normalisées entre une valeur minimale et une valeur maximale qui dépendent du type de la matrice. Attention donc car ces valeurs sont 0 et 1 pour une matrice de type `double`. Des paramètres supplémentaires permettent de changer ces valeurs minimale et maximale, ou bien il faut passer par un *cast* de type.

1. Avant de continuer le TP, placez-vous dans le répertoire `images`, soit en changeant le *Current directory*, soit par la commande:

   ```
   >> cd images
   ```

2. Chargez l'image `Lena1024.jpg` dans la variable `I` (ne pas oublier le ; final, sinon, vous allez vite comprendre son intérêt...) :

```
>> I=imread('Lena1024.jpg');
```

3. Visualisez l'image :

```
>> imshow(I);
Warning: Image is too big to fit on screen; displaying at 67%
> In images.internal.initSize (line 71)
  In imshow (line 336)
```

L'avertissement est simplement pour dire qu'un facteur de zoom (de 67%) a été effectué afin de pouvoir afficher l'intégralité de l'image dans la fenêtre. Ne vous fiez donc pas à la taille apparente de l'image.

4. C'est une image en niveaux de gris: regardez comment elle est stockée en mémoire :

```
>> whos I
  Name         Size                Bytes  Class    Attributes

  I         1024x1024            1048576  uint8
```

Remarquez la valeur de `Class` qui vaut `uint8` : la classe de l'image est un entier codé sur un octet (valeur entre 0 et 255). Vous retrouvez ces informations dans le Workspace.

Vous pouvez églement avoir plus de détails sur le codage dans le fichier lui-même par la commande

```
>> imfinfo('Lena1024.jpg')
ans =
           Filename: 'Lena1024.jpg'
        FileModDate: '08-févr.-2008 11:35:07'
           FileSize: 101318
             Format: 'jpg'
      FormatVersion: ''
              Width: 1024
             Height: 1024
           BitDepth: 8
          ColorType: 'grayscale'
    FormatSignature: ''
    NumberOfSamples: 1
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {}
```

5. Chargez l'image `jwm31.jpg` dans la variable `J` et visualisez-là

```
>> J=imread('jwm31.jpg');
>> imshow(J);
```

6. C'est une image couleur. Regardez comment elle est stockée en mémoire

```
>> whos J
  Name        Size                Bytes  Class    Attributes

  J         590x736x3            1302720  uint8
```

Vous remarquez que `J` est composée de trois matrices (la première pour le rouge, la deuxième pour le vert et la troisième pour le bleu) de taille 590×736, c'est-à-dire 590 lignes et 736 colonnes. L'indice de 'couleur' est le dernier.

7. Transformez cette image en niveaux de gris et stockez le résultat dans la variable `K`, visualisez le résultat, puis vérifiez la classe de `K`. Vous utiliserez la formule: $G = 0.3R + 0.59V + 0.11B$.

```
>> K=0.3*J(:,:,1)+0.59*J(:,:,2)+0.11*J(:,:,3);
>> imshow(K)
>> whos K
  Name        Size                Bytes  Class    Attributes

  K         590x736              434240  uint8
```

Vous remarquez que l'image affichée est correcte, mais vous remarquez aussi que la classe de `K` est `uint8`, ce qui paraît bizarre étant donné les coefficients utilisés pour la transcription. En fait l'image contient de toutes petits erreurs. Nous allons essayer de mettre en évidence le problème.

8. Faites la différences entre les deux premiers canaux de l'image `J` et stockez le résultat dans l'image `L`, puis visualez-le

```
>> L=J(:,:,1)-J(:,:,2);
>> imshow(L)
```

Vous remarquez que l'image est très noire : toutes les valeurs négatives ont été ramenées à 0. On a donc perdu toute information quand la deuxième composante est plus forte que la première. Vous pouvez vérifier la perte effective d'information en cherchant la valeur minimale de `L`

```
>> min(min(L))
ans =
  uint8
   0
```

qui est nulle alors qu'elle devrait être négative.

Pour remédier à ce problème on passe l'image originale en double, puis on refait la manipulation

```
>> J=double(J);
>> L=J(:,:,1)-J(:,:,2);
>> imshow(L, []);
```

Remarquez que nous sommes obligés de passer des paramètres supplémentaires à `imshow`. Ces paramètres définissent la plage d'amplitude à afficher, les valeurs à l'extérieur de cet intervalle étant saturée (noir ou blanc selon le cas, vous pouvez essayer...). Par défaut, pour un type double, cet intervalle est [0,1], ce qui est insuffisant dans notre cas. On peut aussi indiquer un intervalle vide `[]`, et dans ce cas, la fonction `imshow` utilisera automatiquement les valeurs min et max de l'image. La commande précédente est donc équivalente à

```
>> imshow(L, [min(min(L)) max(max(L))]);
```

Ce 'truc' ne marche pas dans le cas précédent car `L` était véritablement de classe `uint8` et ce n'était pas qu'un problème d'affichage car l'information était véritablement absente de `L` (aucune valeur négative codable dans un `uint8`).

Refaites alors l'image `K`

```
>> K=0.3*J(:,:,1)+0.59*J(:,:,2)+0.11*J(:,:,3);
>> imshow(K, [0 255])
>> whos K
  Name        Size              Bytes  Class    Attributes

  K         590x736            3473920 double
```

9. Ecrivez l'image résultante dans le fichier `jwm31_gs.jpg`

```
>> imwrite(uint8(K), 'jwm31_gs.jpg', 'jpg');
```

Remarquez que nous devons caster l'image `K` en `uint8` car le format JPEG ne supporte par des pixels de valeurs réelles. Attention : ce cast ne change pas le type de `K`, il crée juste une variable temporaire qui va contenir le résultat du cast, juste le temps d'effectuer la sauvegarde, mais `K` reste de type `double`.

10. visualisez à présent les trois images I, J, K dans la même fenêtre

```
>> subplot(2,2,1), imshow(I);
>> subplot(2,2,2), imshow(uint8(J));
>> subplot(2,2,3), imshow(K, [0 255]);
```

La commande subplot définit une matrice d'affichage dans les deux premiers paramètres, et le troisième paramètre indique où placer l'image (indice linéaire, en passant les lignes en revue). Notez qu'on ne peut pas utiliser le paramétrage de `imshow` permettant de modifier la valeur minimale et maximale pour l'image `J` car celle-ci est une image couleur, et ce paramétrage n'est permis que pour les images en niveaux de gris, comme l'est `K`. Ici à nouveau, je passe donc par un cast.

11. Enfin, dans un `m-file`, écrivez une fonction `rgb2gs` qui transforme une image couleur en image à niveaux de gris[3]

```
function G=rgb2gs(C)
   G=0.3*double(C(:,:,1)) + 0.59*double(C(:,:,2)) + 0.11*double(C(:,:,3));
end
```

Et testez-la

```
>> L=rgb2gs(J);
>> subplot(2,2,4), imshow(uint8(L));
```

[3]Cette fonction existe dans le toolkit image processing sous le nom `rgb2gray`