

Présentation projet MATLAB®

Ali ZAINOUL

for Styrel - Ametra
April 11, 2024



Table des matières

- 1 Introduction à MATLAB®**
 - Historique et présentation du logiciel
 - Historique de MATLAB®
 - Présentation de MATLAB®
 - Interface utilisateur
 - Notion d'espace de travail (Workspace)
 - Interface Utilisateur
 - Les variables, leurs types et leurs portées
 - Les variables dans MATLAB
 - Les types de variables MATLAB
 - Portées des variables
 - Les commandes de base à connaître

Table des matières

- 2** Les bases de la programmation MATLAB®
 - Notions de scripts et de fonctions sur MATLAB®
 - Notion de script MATLAB®
 - Notion de fonctions MATLAB®
 - Matrices, vecteurs, cellules et structures
 - Cellules
 - Structures
 - Vecteurs
 - Matrices
 - Quelques commandes avancées
 - Exécution des codes, RUN

Table des matières

3 Import et export des données sur MATLAB®

- Importer des données
- Exporter des données
- Exploiter des données
- Images et graphes

Table des matières

- 4** Programmation améliorée
 - À faire et à ne pas faire
 - Astuces et optimisation
 - Débogage

Table des matières

- 5** Les graphes dans MATLAB®
 - Affichage des variables
 - Les différents types de graphes
 - Menus et édition
 - L'onglet PLOTS

Table des matières

- 6** Développer une interface graphique
 - Prise en main de GUIDE
 - Les éléments de contrôle
 - Callbacks et Handles
 - Graphes

**** Introduction ****

Historique de MATLAB®

- MATLAB® a été développé par MathWorks, une entreprise basée à Natick, Massachusetts, États-Unis.
- Il a été initialement publié en 1984 par Cleve Moler, alors professeur de mathématiques à l'Université du Nouveau-Mexique.
- MATLAB® est devenu un outil de calcul numérique très populaire, utilisé dans divers domaines tels que les mathématiques, l'ingénierie, la finance, la recherche, et bien d'autres encore.

Évolution de MATLAB®

- Au fil des ans, MATLAB® a continué à évoluer, introduisant de nouvelles fonctionnalités et capacités.
- Les versions successives de MATLAB® ont ajouté des outils de visualisation avancés, des bibliothèques de fonctions spécialisées, une intégration avec d'autres langages de programmation, et bien plus encore.
- Aujourd'hui, MATLAB® est largement utilisé dans l'enseignement, la recherche, le développement de produits et d'applications, ainsi que dans de nombreuses autres applications industrielles et commerciales.

Introduction à MATLAB®



Figure: MATLAB® logo

■ MATLAB® en bref:

- MATrix LABoratory, plateforme orientée calcul numérique et matriciel
- Environnement propre de développement;
- Code source protégé (contrairement à Maple ou Mathematica qui sont open-source);

Avantages et cas d'utilisation MATLAB®

■ Avantages de MATLAB®:

- Convient aux débutants;
- Syntaxe intuitive et simple d'utilisation;
- Langage interprété (chaque expression est traduite en code machine au moment de son exécution).

■ Cas courants d'utilisation

- Calcul numérique et matriciel;
- Intelligence Artificielle & Machine Learning;
- Les systèmes de contrôle;
- Le traitement du signal et les séries temporelles etc.

Espace de travail (Workspace) en MATLAB

- L'espace de travail en MATLAB est l'ensemble des variables que vous créez et manipulez lors d'une session MATLAB.
- Il comprend toutes les variables que vous avez définies dans la session en cours, ainsi que leurs valeurs.
- L'espace de travail est l'endroit où MATLAB stocke les données et effectue les calculs.
- Il existe deux principaux types d'espaces de travail en MATLAB : l'espace de travail de base (base workspace) et les espaces de travail des fonctions (function workspaces).

Espace de travail de base (Base Workspace)

- L'espace de travail de base est l'espace de travail principal où vous interagissez directement avec MATLAB.
- Lorsque vous définissez des variables ou exécutez des scripts dans la fenêtre de commande, ils sont stockés dans l'espace de travail de base.
- Vous pouvez afficher et manipuler les variables dans l'espace de travail de base à l'aide du navigateur d'espace de travail (Workspace browser) ou de la commande `whos`.

Espaces de travail des fonctions (Function Workspaces)

- Les espaces de travail des fonctions sont créés lorsque vous appelez une fonction dans MATLAB.
- Chaque appel de fonction crée son propre espace de travail, qui est distinct de l'espace de travail de base.
- Les variables créées à l'intérieur d'une fonction sont locales à cette fonction et ne sont pas accessibles en dehors de celle-ci.
- Lorsqu'une fonction termine son exécution, son espace de travail est supprimé de la mémoire et toutes les variables créées à l'intérieur sont supprimées.

Gestion des espaces de travail

- Vous pouvez supprimer des variables de l'espace de travail à l'aide de la commande `clear`.
- Pour supprimer toutes les variables de l'espace de travail de base, vous pouvez utiliser `clearvars` ou `clear all`.
- Vous pouvez également enregistrer l'espace de travail actuel dans un fichier en utilisant la commande `save` et charger des variables à partir d'un fichier en utilisant la commande `load`.

Présentation de l'interface utilisateur

Voici une capture d'écran illustrant l'interface utilisateur de Matlab, ainsi qu'un premier programme à exécuter, le fameux helloWorld :

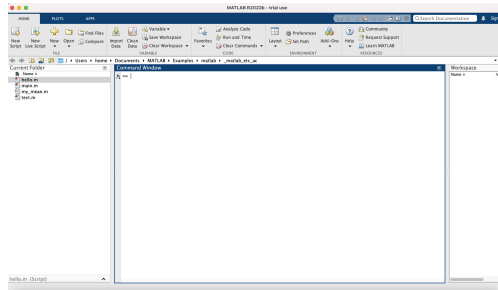


Figure: MATLAB® Interface

Compilation d'un programme *via* l'Interface Utilisateur

La compilation du fameux hello.m se fait en cliquant sur le bouton `run`.
Voici une capture d'écran illustrant la marche à suivre :

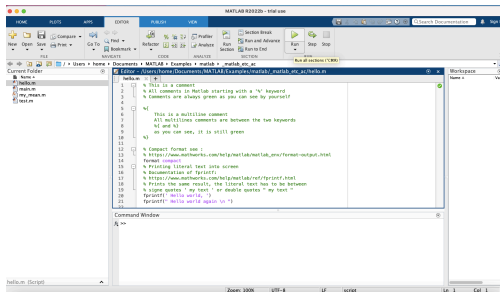


Figure: MATLAB® Interface

Résultat après compilation

Après compilation du hello.m, on obtient le résultat dans le Command Window:

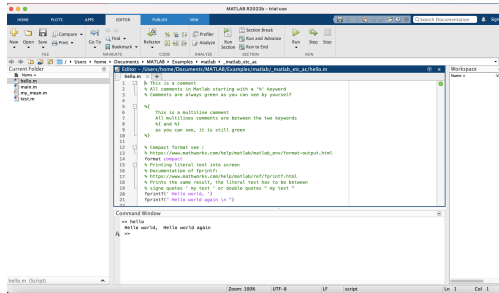


Figure: MATLAB® Interface

Les variables

- Une variable est caractérisée par:
 - Un **nom** formé d'une combinaison de lettres et de chiffres. Le premier caractère commence toujours par une lettre (e.g myArray = [1 2 3]);
 - Une **affectation**, **assignation** ou implicitement une **instanciation de classe**.
 - **Remarque:** Les variables x et X ne sont pas les mêmes ($x \neq X$).(Sensibilité à la casse)
- Toute variable dans MATLAB est considérée comme étant un tableau.

La représentation des variables en MATLAB

- **Scalars:** tableaux à une seule ligne **et** une seule colonne
 - scalaire 1×1 ;
- **Vecteurs:** tableaux à une seule ligne **ou** à une seule colonne:
 - vecteur **ligne** à une ligne et n colonnes $1 \times n$;
 - vecteur **colonne** à m lignes et 1 colonne $m \times 1$;
- **Matrices:** tableaux à plusieurs lignes **et** à plusieurs colonnes
 - matrice à m lignes et n colonnes $m \times n$;

Remarques:

- Un scalaire est donc une matrice avec $m = n = 1$ (1×1) ;
- Un vecteur colonne est donc une matrice avec $n = 1$ ($m \times 1$) ;
- Un vecteur ligne est donc une matrice avec $m = 1$ ($1 \times n$) ;

Les types de base de variables MATLAB

- On reconnaît les types habituels de variables en MATLAB:
 - Les **booléens** `logical`, soit **1** (true) soit **0** (false).
 - Les **entiers** `intnumBits` et `uintnumBits`, où $numBits \in \{16, 32, 64\}$;
 - Les **réels** `double` ;
 - Les **caractères** `char` (e.g: `myChar = 'A'`)
 - Les **chaînes de caractères** `string` (e.g: `myString = " A string. "`);
 - Les **complexes**
 - etc.
- D'autres types de données existent... Voir les différents types de données.

Exemple de types de base de variables MATLAB

Command Window

```
» a = 2; b = -3; c = 5.4 ; myBoolean = (a==b); z = 2 + 2i; ... aChar = 'c'; alongChar = 'Hello k'; aString = "oui"; whos
```

<i>Name</i>	<i>Size</i>	<i>Bytes</i>	<i>Class</i>	<i>Attributes</i>
<i>a</i>	1x1	8	<i>double</i>	
<i>aChar</i>	1x1	2	<i>char</i>	
<i>aString</i>	1x1	148	<i>string</i>	
<i>alongChar</i>	1x7	14	<i>char</i>	
<i>b</i>	1x1	8	<i>double</i>	
<i>c</i>	1x1	8	<i>double</i>	
<i>myBoolean</i>	1x1	1	<i>logical</i>	
<i>z</i>	1x1	16	<i>double</i>	<i>complex</i>

Création des variables numériques

Type	Description
double	Tableaux en double précision sur 64 bits
single	Tableaux en simple précision sur 32 bits
int8	Tableaux d'entiers signés sur 8 bits
int16	Tableaux d'entiers signés sur 16 bits
int32	Tableaux d'entiers signés sur 32 bits
int64	Tableaux d'entiers signés sur 64 bits
uint8	Tableaux d'entiers non signés sur 8 bits
uint16	Tableaux d'entiers non signés sur 16 bits
uint32	Tableaux d'entiers non signés sur 32 bits
uint64	Tableaux d'entiers non signés sur 64 bits

Exemple de création de variables numériques

```
1 % Example of creating numeric variables in MATLAB
2 x = 10; % double
3 y = single(20); % single
4 z = int8(5); % int8
5 w = uint64([1 2 3 4 5]); % Array of 1x5 of uint64
```

Conversion entre les types numériques

Fonction	Description
cast	Convertit une variable vers un type de données différent
typecast	Convertit un type de données sans changer les données sous-jacentes

Exemple de conversion entre les types numériques

```
1 % Example of conversion between numeric types in MATLAB
2 x = 10; % double
3 y = cast(x, 'single'); % conversion to single
4 z = typecast(single(20), 'double'); % conversion without changing
   the data
```

Interrogation sur le type et la valeur

Fonction	Description
allfinite	Détermine si tous les éléments du tableau sont finis
anynan	Détermine si un élément du tableau est NaN
isinteger	Détermine si l'entrée est un tableau d'entiers
isfloat	Détermine si l'entrée est un tableau de nombres flottants
isnumeric	Détermine si l'entrée est un tableau numérique
isreal	Détermine si le tableau utilise un stockage complexe
isfinite	Détermine quels éléments du tableau sont finis
isinf	Détermine quels éléments du tableau sont infinis
isnan	Détermine quels éléments du tableau sont NaN

Exemple sur l'interrogation sur le type et la valeur

```
1 % Example of querying type and value in MATLAB
2 A = magic(3); % Creates a nxn magic matrix with elements ranging
   from 1 to n (n=3)
3 is_int = isinteger(A); % Checks if A is an array of integers
4 is_real = isreal(A); % Checks if A uses complex storage
5 is_numeric = isnumeric(A); % Checks if A is a numeric array
6 is_float = isfloat(A); % Checks if A is an array of floating-point
   numbers
7 is_finite = isfinite(A); % Determines which elements of A are
   finite
8 is_inf = isinf(A); % Determines which elements of A are infinite
9 is_nan = isnan(A); % Determines which elements of A are NaN
```

Les limites de valeurs numériques

Fonction	Description
eps	Précision relative des nombres flottants
flintmax	Plus grand entier consécutif au format à virgule flottante
Inf	Crée un tableau de toutes les valeurs Inf
intmax	Plus grande valeur d'un type d'entier spécifique
intmin	Plus petite valeur d'un type d'entier spécifique
NaN	Crée un tableau de toutes les valeurs NaN
realmax	Plus grand nombre flottant positif
realmin	Plus petite nombre flottant normalisé

Exemple sur les limites de valeurs numériques

```
1 % Example of numeric value limits in MATLAB
2 eps_value = eps; % Relative accuracy of floating-point numbers
3 flintmax_value = flintmax; % Largest consecutive integer in
   floating-point format
4 inf_array = Inf(3); % Creates an array of three Inf values
5 intmax_value = intmax('int64'); % Largest value of a specific
   integer type
6 intmin_value = intmin('int16'); % Smallest value of a specific
   integer type
7 nan_array = NaN(2,2); % Creates an array of NaN values
8 realmax_value = realmax; % Largest positive floating-point number
9 realmin_value = realmin; % Smallest normalized floating-point
   number
```

Portée des variables en MATLAB

- MATLAB prend en charge deux types de portée de variables : **locale** et **globale**.
 - Les variables **locales** sont déclarées à l'intérieur d'une fonction et ne sont accessibles qu'à l'intérieur de cette fonction.
 - Les variables **globales** sont déclarées en dehors de toute fonction et sont accessibles de manière globale dans tout le script MATLAB grâce au mot-clé : `global`.
- Pour plus d'informations sur la fonction `global`, vous pouvez consulter la documentation officielle de MATLAB :
Documentation de la fonction global

Exemple de partage d'une variable globale entre plusieurs fonctions

```
1 % Sharing a global variable between multiple functions
2 function setGlobalx(value)
3     global x
4     x = value;
5 end
6
7 function result = getGlobalx
8     global x
9     result = x;
10 end
11
12 % Set the value of the global variable x and retrieve it from another workspace.
13 setGlobalx(1138);
14 result = getGlobalx;
15 disp(result); % Displays 1138
```

Exemple de partage d'une variable globale entre une fonction et une ligne de commande

```
1 % Sharing a global variable between a function and a command line
2 clear all;
3 setGlobalx(42);
4 x; % Error: x is undefined from the command line
5 global x
6 disp(x); % Displays 42
7
8 % Modify the value of x and use the defined function to return the
   global value from another workspace.
9 x = 1701;
10 result = getGlobalx;
11 disp(result); % Displays 1701
```

Les commandes de base à connaître

- Le lien présent regroupe un ensemble de commandes de base sur MATLAB, pour un gain de temps optimal, l'Université de Bourgogne a résumé l'ensemble des commandes couramment utilisées dans le cadre de la programmation en MATLAB Commandes de base¹.

¹Merci à l' Université de Bourgogne

**** Basics ****

Notions de scripts et de fonctions sur MATLAB®

- Découverte de l'utilisation des **scripts**.
- Découverte de l'utilisation des **fonctions**.
- Application du principe **DRY** (Don't Repeat Yourself).
- Écrire des fonctions / scripts **réutilisables**.

Notion de script MATLAB®

- Un programme MATLAB® élémentaire s'appelle un **script**.
- Un **script** est un fichier contenant une séquence de plusieurs lignes de commandes et d'appels de fonction MATLAB®.
- Vous pouvez exécuter un **script** en saisissant son nom dans la ligne de commande. Ci-dessous la démarche à suivre pour un simple programme helloWorld

Notion de script MATLAB® - Suite

- Il suffit de taper le nom du script: `name_Script` dans la ligne de commande si le fichier s'appelle `name_Script.m`

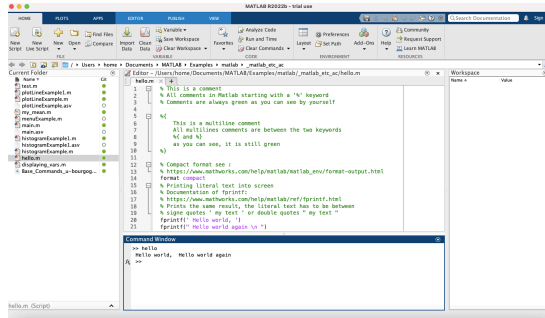


Figure: hello

Définition d'une fonction

- Les **fonctions** sont des blocs autonomes de code qui favorisent la modularité et la réutilisabilité. Elles encapsulent l'implémentation, acceptent des paramètres et retournent des valeurs, ce qui réduit la complexité en divisant les problèmes en sous-tâches.
- Une **fonction** est simplement une suite d'instructions qui répond à un problème donné:
 - Calculer la moyenne d'une liste ;
 - Inverser une matrice ;
 - Trouver la solution d'un système linéaire ;
 - Ou encore, Trouver le plus court chemin dans un graphe ;
 - etc.

Notion de fonctions MATLAB®

- Les **fonctions** sont des fichiers scripts (extension .m) qui acceptent des arguments en entrée et qui retournent des arguments en sortie.
- Les noms du fichier et de la fonction doivent être **identiques**. (e.g.: **myFunction** pour le nom de la fonction et **myFunction.m** pour le nom du script)
- Les fonctions opèrent sur les variables au sein de leur propre espace de travail, qui est distinct de l'espace de travail auquel vous accédez au niveau de l'invite de commande MATLAB®.

Notion de fonctions MATLAB® - Suite

- Le **nom du fichier** contenant la fonction doit **correspondre exactement au nom de la fonction**.
- Il est possible de **regrouper plusieurs fonctions** dans le même M-file (script .m), mais **seule la fonction ayant le même nom que le fichier** peut être **utilisée** ou **appelée** depuis la fenêtre de commandes, une autre fonction ou un script.
- Les autres fonctions éventuellement stockées dans le fichier peuvent s'appeler entre elles, mais **elles ne sont pas visibles de l'extérieur**.

Signature d'une fonction MATLAB®

- La signature d'une fonction MATLAB® prend cette forme:

Listing 1: functionSignature.m

```
1 function [output1, ..., outputn] = myFunction(input1, ..., inputn)
2     % instructions
3     % instructions
4 end
```

Voici quelques exemples élémentaires:

Exemple 1

■ La fonction hello_name:

Listing 2: hello_name.m

```
1 function [greetings] = hello_name ( name )
2     if ~isempty(name)
3         myHello = "Hello ";
4         % strcat
5         % input: s_1, ..., s_n of type: char | cell | string
6         % output: outputString of type: string
7         % See: https://fr.mathworks.com/help/matlab/ref/strcat.html
8         greetings = strcat(myHello, name, "!");
9     end
10 end
```

Exemple 1 - Suite

- La fonction hello_name.m ci-dessus peut être appelée dans le script personalGreetings.m, il suffit que la fonction ainsi que le script soient présents dans le même *WorkSpace*. (e.g.: exemple:
`/Users/home/Documents/MATLAB/Examples/MATLAB/_MATLABStyrel`)

Exemple 1 - Suite

Listing 3: personalGreetings.m

```
1 % Compact format see :
2 % https://www.mathworks.com/help/matlab/matlab\_env/format-output.html
3 format compact
4
5 % Creating a variable named " name ", and assigning to it
6 % an input value of type string i.e: 's'
7 % input(...) is a keyword! More examples will be shown
8 % See: https://www.mathworks.com/help/matlab/ref/input.html
9 name = input('What 's your name?', 's');
10
11 % Checking if there is an entry, then printing the name
12 % otherwise do nothing
13 myGreetings = hello_name(name);
14 if ~isempty(myGreetings)
15     disp(myGreetings)
16 end
17 % A statement of type "if" must have an "end" keyword
18 % i.e: if condition
19 %     do stuff
20 %     end
```

Exemple 2

- L'exemple suivant du calcul de la moyenne d'un vecteur ou d'un tableau illustre la procédure à suivre afin d'appeler la fonction calculateAverage.m dans un script quelconque. Considérons la fonction suivante:

Listing 4: calculateAverage.m

```
1 function ave = calculateAverage(x)
2     ave = sum(x(:))/numel(x);
3 end
```

Exemple 2 - Suite

- L'appel à cette fonction calculateAverage.m se fait dans le test_mean.m ci-dessous:

Listing 5: test_mean.m

```
1 % Writing External functions, they have to be in the same directory
2 % e.g /Users/home/Documents/MATLAB/Examples/matlab/_matlab_etc_ac
3 %     calculateAverage.m | my_mean.m | test_mean.m
4
5 myVector1 = [1 2 3 4]           % average = 10 / 4 = 2.5
6 myVector2 = [2.0 5.2 4.8]       % average = 12.0 / 3 = 4.0
7 myVector3 = [-2.2 -1.2 -9.6 +10.5] % average = -2.5 / 4 = -0.6250
8 myVectors = {myVector1, myVector2, myVector3} % an array of vectors
9
10 mylengthVectors = numel(myVectors) % one can call numel(x) OR length(x)
```


Exemple 2 - Suite

■ Suite: test_mean.m

```
1 % myotherlengthVectors = length(myVectors)
2 for i = 1 : mylengthVectors
3     fprintf("-----")
4     fprintf("\n Test Case %d \n", i)
5     m_ = mean(myVectors{i})
6     my_m = my_mean(myVectors{i})
7     my_average = calculateAverage(myVectors{i})
8     test_my_mean = (m_ == my_m)
9     test_my_average = (m_ == my_average)
10 end
```

Fonctions à plusieurs variables de sortie

■ Fonctions à plusieurs variables de sortie:

- Considérons l'exemple précédent, où l'on a envie cette fois-ci de récupérer une liste / un tableau / un vecteur de plusieurs statistiques, on se place dans l'exemple où l'on a envie de récupérer à la fois la taille, la somme, la moyenne ainsi que l'écart-type d'une variable discrète; la fonction stats.m ci-dessous à plusieurs **outputs** nous permet de faire cela:

Exemple 3

■ La fonction à plusieurs outputs stats.m:

Listing 6: stats.m

```
1 function [myLength, mySum, myAverage, myStdeviation] = stats(x)
2     myLength = length(x);
3     mySum = sum(x);
4     myAverage = mySum/myLength;
5     myStdeviation = sqrt( sum( (x-myAverage).^ 2 /myLength) );
6     % Component wise squaring
7     % the function above may be rewritten as follow:
8     % myAverageVector = x-myAverage
9     % myVectorDeviation = myAverageVector.^ 2
10    % the last line is equivalent to:
11    % sqr = @(x) x^2;
```

Exemple 3 - Suite

■ Suite: La fonction à plusieurs outputs stats.m:

```
1      % myVectorDeviation = []
2      % for i = 1 : length(myAverageVector)
3      %     myVectorDeviation(i) = sqr(myAverageVector(i))
4      % end
5      % Dividing a vector by a scalar
6      % myVectorDeviationMean = myVectorDeviation / myLength
7      % Summing elements of a vector
8      % averageDeviation = sum(myVectorDeviationMean)
9      % Calculating the squirt root of a scalar
10     % myStdeviation = sqrt(averageDeviation)
11 end
```

Exemple 3 - Suite

- L'on pourrait tester la fonction à plusieurs outputs stats.m en exécutant le script test_stats.m :

Listing 7: test_stats.m

```
1 % Writing External functions, they have to be in the same directory
2 % e.g /Users/home/Documents/MATLAB/Examples/matlab/_matlab_etc_ac
3 % stats.m | test_stats.m
4
5 myVector1 = [1 2 3 4] % average = 10 / 4 = 2.5
6 myVector2 = [2.0 5.2 4.8] % average = 12.0 / 3 = 4.0
7 myVector3 = [-2.2 -1.2 -9.6 +10.5] % average = -2.5 / 4 = -0.6250
8 myVectors = {myVector1, myVector2, myVector3} % an array of vectors
9
10 mylengthVectors = numel(myVectors) % one can call numel(x) OR length(x)
```

Exemple 3 - Suite

■ Suite: test_stats.m:

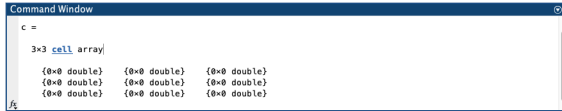
```
1 % myotherlengthVectors = length(myVectors)
2 for i = 1 : mylengthVectors
3     fprintf("-----")
4     fprintf("\n Test Case %d \n", i)
5     [length_,sum_,average_, stD_] = stats(myVectors{i})
6 end
```

Les cellules dans MATLAB®

- Un tableau de Cellules dans MATLAB® est un type de données avec des conteneurs de données indexées appelés **cellules**, où chaque cellule peut contenir n'importe quel type de données.
- Les tableaux de cellules contiennent généralement:
 - des listes de texte
 - des combinaisons de texte et de nombres
 - des tableaux numériques de différentes tailles

Création de cellules dans MATLAB®

- Il existe quatre manières différentes afin de créer une cellule, on illustrera ici les deux premiers cas d'utilisation:
 - $c = \text{cell}(n)$ retourne un tableau de cellules de matrices vides de taille $n \times n$.
Exemple: $c = \text{cell}(3)$ crée un tableau de cellules vides de taille 3×3 , le rendu est comme suit:



```
Command Window
c =
3x3 cell array
{0x0 double} {0x0 double} {0x0 double}
{0x0 double} {0x0 double} {0x0 double}
{0x0 double} {0x0 double} {0x0 double}
```

Figure: 3x3CellArrayExample

Création de cellules dans MATLAB® - Suite

■ Suite:

- Ainsi, si l'on écrit: $c1 = \text{ones}(2)$, le premier élément du tableau de cellules c va représenter une matrice 2x2 avec des 1 dans ses éléments. L'on peut voir cela dans la figure suivante:



```
Command Window
>> c{1}

ans =

     1     1
     1     1

fx >> |
```

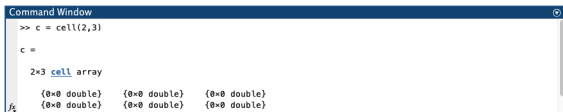
Figure: cell1matrix

- Vous l'auriez compris, on accède aux éléments d'un tableau de cellules grâce à des accolades {}.

Création de cellules dans MATLAB® - Suite

■ Suite:

- De la même manière, on crée un tableau de cellules de taille $n \times m$ en écrivant: $c = \text{cell}(n, m)$, comme le montre l'exemple suivant:



```
Command Window
>> c = cell(2,3)

c =

    2x3 cell array

    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}
```

Figure: 2x3CellArrayExample

- On peut généraliser le concept ci-dessus, ainsi pour créer un tableau de cellules de taille $s_1 \times s_2 \times \dots \times s_n$, on procède comme suit:
 $c = \text{cell}(s_1, s_2, \dots, s_n)$, ainsi $c = \text{cell}(2, 3, 4)$ crée un tableau de cellules de dimension $2 \times 3 \times 4$.
- Voici un exemple complet.

Les structures dans MATLAB®

- Les structures en programmation informatique est un cas d'usage courant, car ces dernières permettent de regrouper et d'organiser l'information d'une manière optimale.
- Un tableau de structures en MATLAB® est un type de données qui regroupe des données associées à l'aide de **conteneurs** (**containers**) de données appelés **champs**. (**fields**).

Les structures dans MATLAB® - Suite

- Chaque champ (field) peut contenir n'importe quel type de donnée.
- L'accès à une donnée se fait en utilisant la notation " . "
Exemple: `myStruct.myField1 = 33` et `myStruct.myField2 = "Hello"` ainsi on a créé une structure qui porte le nom `myStruct`, avec deux champs `myField1` et `myField2`.

Création de structures dans MATLAB®

- On distingue cinq manières de créer une structure dans MATLAB®:
 - `s = struct`, ce qui permet de créer une structure 1x1 scalaire sans champs.



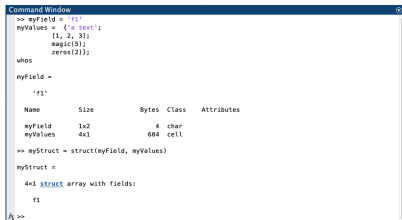
```
Command Window
>> emptyStruct = struct
emptyStruct =
    struct with no fields.
>> whos
      Name      Size      Bytes  Class  Attributes
emptyStruct    1x1           0   struct
```

Figure: emptyStruct

- `s = struct(field, value)`, ce qui permet de créer une structure avec un champ et une valeur. L'exemple ci-dessous (tiré de la documentation MATLAB®) montre que la valeur peut être de n'importe quel type, autrement dit, un tableau de cellules, où chaque cellule contient des données différentes!

Création de structures dans MATLAB® - Suite

■ Suite:



```
Command Window
>> myField = 'f1';
myValues = {'a text';
            [1, 2, 3];
            magic(5);
            zeros(2)};

whos

myField =
    'f1'

Name      Size      Bytes  Class  Attributes
myField    1x2         4    char
myValues   4x1        684    cell

>> myStruct = struct(myField, myValues)

myStruct =
    4x1 struct array with fields:
        f1
```

Figure: myStructExample

- `s = struct(field1, value1, ..., fieldn, valuen)`, ce qui permet de créer une structure avec plusieurs champs et plusieurs valeurs. J'invite les lecteurs à regarder la [documentation](#) pour un exemple détaillé.

Création de structures dans MATLAB® - Suite

■ Suite:

- `s = struct([])`, crée une structure vide 0x0 sans champs. Afin d'insérer des éléments à la structure `s`, on procède ainsi: `s(1).myField = myValue`.
Exemple: `s(1).a = 1`, ainsi `s` est une structure avec le champ "a" et la valeur de `a` est de 1.
- `s = struct(obj)`, j'invite les lecteurs à regarder la [documentation](#) concernant cette déclaration.

Rappels sur les vecteurs

- Un Vecteur en mathématiques, et plus précisément en algèbre linéaire, est un objet qui généralise plusieurs notions, notamment celles de la géométrie (points, translations, etc.), de l'algèbre (solution d'un système linéaire), et de la physique (notion de force, notion vitesse et d'accélération, etc.).
- En notation mathématique, et en considérant la base $e_1 \dots e_n$ on écrit un vecteur ainsi: $\vec{u} = u_1 \vec{e}_1 + u_2 \vec{e}_2 + \dots + u_n \vec{e}_n$, ce qui pourrait être représenté aussi comme suit: $\vec{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}$

Les vecteurs dans MATLAB®

- Un vecteur dans MATLAB® est simplement un tableau ou une liste (dans Python par exemple) et il peut être représenté de diverses manières, elles seront toutes listées dans le cadre de ce cours:
 - `myVector = [element1 element2 ... elementn]`, permet de créer un vecteur de n éléments. Les éléments ne sont pas obligés d'avoir le même type, néanmoins certaines règles s'appliquent.
 - `myVector = [element1, element2, ..., elementn]`, On peut faire exactement la même chose que précédemment, avec des virgules " , " entre les éléments.
 - Voici un exemple complet.

L'opérateur colon

- Un concept important quand on utilise les vecteurs est l'opérateur colon. Il permet lui aussi la création d'un vecteur, et mais surtout, pouvoir manipuler un vecteur avec le moins de lignes de code possible.
 - On peut créer un vecteur en appelant `myVector = beginning:end`, à titre d'exemple: `myVector = 1 : 10` crée un vecteur de 10 composantes de taille 1x10 dont les composantes sont les nombres de 1 à 10.
 - On peut créer un vecteur en appelant `myVector = beginning:condition:end`, à titre d'exemple: `myVector = 2 : 2 : 10` crée un vecteur de 5 éléments de taille 1x5 composés des nombres paires de 2 à 10.

Transposée d'un vecteur

- Dernier élément important sur les vecteurs, c'est ce qu'on appelle la **transposée d'un vecteur**. La transposée (en algèbre linéaire) d'un vecteur ligne \vec{u} est le même vecteur mais représenté en colonne et réciproquement. La transposée est représentée par une apostrophe " ' " Ainsi, si \vec{u} est un vecteur, sa transposée est: \vec{u}' .

$$\text{i.e. : } \vec{u} = (u_1, \dots, u_n) \Leftrightarrow \vec{u}' = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}$$

- Dernière propriété à connaître: $\vec{u}'' = \vec{u}$, autrement dit la transposée de la transposée d'un vecteur est le vecteur lui-même.

Rappels sur les matrices

- En algèbre linéaire, les matrices sont simplement des tableaux d'éléments. On dit que la matrice A a $m \times n$ éléments et on écrit: $A = (a_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$ ou encore:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

Un exemple simple est donné comme suit:

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \text{ où: } a_{1,1} = a_{2,1} = a_{2,2} = 1 \text{ et } a_{1,2} = -1.$$

Les matrices dans MATLAB®

- Dans MATLAB®, on déclare et initialise une matrice A de taille $m \times n$ ainsi:
$$A = [a_{1,1} \ a_{1,2} \dots a_{1,n}; \dots; a_{m,1} \ a_{m,2} \dots a_{m,n}]$$
- Exemple: afin de déclarer la matrice $\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ dans MATLAB®, il suffit d'écrire: $A = [1 \ -1; 1 \ 1]$.
- **Remarque importante:** On a l'**obligation** de séparer les lignes d'une matrice dans MATLAB® avec un point-virgule ;.
- Voici un exemple complet.

Quelques commandes avancées

- Le lien suivant explicite certaines commande avancées à connaître, on utilisera quelques unes dans le cadre de ce cours.
Commandes avancées.

Exécution des codes, RUN

- On a vu qu'il existe deux manières différentes d'exécuter un code MATLAB®:
 - En cliquant sur le bouton **Run** en haut à droite, comme le montre la figure suivante:

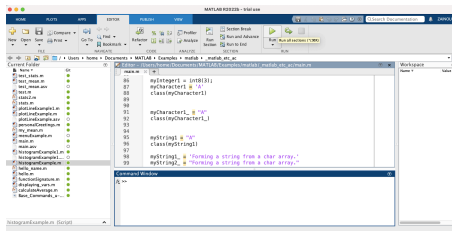


Figure: runButton

Exécution des codes, RUN

■ Suite:

- En appelant directement le nom du script dans le **Command Window**, comme le montre la figure suivante:

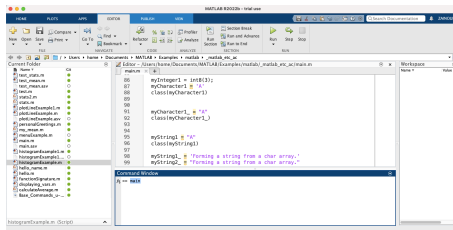


Figure: runCommandWindow

**** ImportExport ****

Import et export des données sur MATLAB®

- Une des pratiques courantes dans l'utilisation d'un langage de programmation est la manipulation des fichiers: textes, csv, tableaux, images, graphes, graphiques etc. La question de leur import et export se pose naturellement:
 - Comment importer un fichier dans MATLAB ?!
 - Comment exporter un fichier dans MATLAB ?!
 - Comment exploiter un fichier dans MATLAB ?!
- Ce [lien](#) présente les formats de fichiers supportés pour de l'import/export dans MATLAB. On traitera ici quelques exemples, notamment l'importation et l'exportation d'un graphe ainsi que d'un fichier CSV.

Import des données sur MATLAB®

- On s'intéressera ici à l'import d'une image, d'un fichier texte et d'un fichier CSV.
 - **L'import d'un graphique:** ceci se fait grâce à la fonction imread(...), un exemple complet est donné dans le fichier imreadMyimg.m.
 - **L'import d'un fichier texte .txt ou .dat, d'une feuille de calcul type csv, Excel® etc.,** deux exemples sont donnés ici, l'ouverture d'un fichier .txt et l'ouverture d'un fichier .csv:
 - ▶ **D'un fichier .txt:** un exemple illustrant les propos est donné par: readtableMytxt.m
 - ▶ **D'un fichier .csv:** un exemple illustrant les propos est donné par: readtableMycsv.m

Export des données sur MATLAB®

- On s'intéressera ici à l'export d'une image, d'un fichier texte et d'un fichier CSV.
 - **L'export d'un graphique:** ceci se fait grâce à la fonction imwrite(...), un exemple complet est donné dans le fichier imwriteMyimg.m.
 - **L'export d'un fichier texte .txt ou .dat, d'une feuille de calcul type csv, Excel® etc.,** deux exemples sont donnés ici, l'écriture d'un fichier .txt et l'écriture d'un fichier .csv:
 - ▶ **D'un fichier .txt:** un exemple illustrant les propos est donné par: writetableMytxt.m
 - ▶ **D'un fichier .csv:** un exemple illustrant les propos est donné par: writetableMycsv.m

Exploitation des données sur MATLAB®

- Dans MATLAB, parfois on veut avoir accès à la donnée afin de pouvoir l'exploiter grâce à un logiciel externe, à ce moment là, on peut enregistrer toute variable dans MATLAB *via* trois manières différentes:
 - Les fichiers .mat de MATLAB, leur utilisation est relativement simple, ils permettent de sauvegarder les données MATLAB dans le workspace afin de pouvoir les charger plus tard. L'utilisation des fonctions save et load est illustrée dans ce lien: [exploitation des données](#).
 - Les fichiers .csv comme vu dans la section précédente.
 - Les fichiers .txt ou .dat comme vu dans la section précédente.

Exploitation des images et graphes sur MATLAB®

- La documentation officielle de MATLAB offre une variété d'exemples d'utilisation et d'exploitation d'une image ou d'un graphe MATLAB.

**** ImprovedProgramming ****

À faire et à ne pas faire

- Voici une liste non exhaustive de certains principes fondamentaux en programmation informatique et développement logiciel d'une manière large:
 - Tests unitaires
 - DRY (Don't Repeat Yourself)
 - KISS (Keep It Simple, Stupid!)
 - SOLID
 - Développement Agile et SCRUM
 - Programmation Orientée Objet (pas concerné dans le cadre de ce cours)
 - Design Patterns (pas concerné dans le cadre de ce cours)

À faire et à ne pas faire

■ Suite:

- Les tests unitaires permettent de vérifier si une fonction ou une méthode ou alors une classe donnent bien le résultat attendu, c'est une pratique courante en programmation informatique. Dans MATLAB, deux procédés possibles, soit la solution native **assert(condition)** (comme en C/C++) ou *via* XUnit (pas concerné par ce cours).
- DRY (Don't Repeat Yourself), ici le principe est simple, c'est une philosophie de code qui stipule qu'il ne faut pas se répéter. Après avoir codé une fonction ou une méthode, il est utile de savoir reconnaître si elle est dupliquée ailleurs dans le code, si tel est le cas, c'est que l'on a pas respecté le principe DRY.

À faire et à ne pas faire

■ Suite:

- KISS (Keep It Simple, Stupid!), c'est un principe qui stipule qu'il faut simplifier son code autant que possible. Chaque fonction doit avoir une seule et unique tâche.
- SOLID, c'est un acronyme mnémonique qui regroupe cinq principes primordiaux en Programmation Orientée Objet. Il permet d'avoir une architecture logicielle bien établie, solide et facilement modifiable.

À faire et à ne pas faire

■ Suite:

- Développement Agile et SCRUM, dans le cadre de la gestion de projets, il est courant qu'une équipe de deux personnes ou plus travaille sur un repository Github en parallèle, l'agilité dans le développement informatique permet d'avoir une solution fiable, rapide et mais surtout de pouvoir décomposer un projet complexe en étapes simples à réaliser (ce qu'on appelle des User Stories).

À faire et à ne pas faire

■ Suite:

- Programmation Orientée Objet (POO) (pas concerné dans le cadre de ce cours), c'est un des principes fondamentaux en programmation informatique, ses origines remontent aux années 1970 avec les langages Simula et Smalltalk, mais le principe a rapidement pris son envol grâce à la création du langage C++ qui est l'extension du langage C, avec en effet, cette quête de rendre les logiciels plus robustes. Une mnémotechnique utile regroupant les six concepts fondamentaux de la POO est: **ACOPIE**
 - ▶ Abstraction
 - ▶ Class
 - ▶ Object
 - ▶ Inheritance
 - ▶ Polymorphism
 - ▶ Encapsulation

À faire et à ne pas faire

■ Suite:

- Design Patterns (pas concerné dans le cadre de ce cours) ce qu'on appelle les patrons de conception; un problème récurrent a forcément une solution optimale déjà traitée par d'autres, **utilisez-là!**.

Astuces et optimisation

- Voici quelques concepts à connaître sur la programmation informatique d'une manière générique, et sur MATLAB en particulier:
 - Séparer le processus en plusieurs petits éléments.
 - Chaque élément doit être codé dans une fonction qui a une seule et unique tâche.
 - **Une idée clé:** chaque fonction codée doit être aussi générique et indépendante du reste du code que possible. (l'encapsulation).

Débogage

- Il est courant qu'un développeur / programmeur veuille déboguer son code à un moment donné afin de savoir si telle fonction ou telle feature rajoutée n'impacte pas le reste de l'exécution du code. Dans MATLAB, trois manières différentes existent afin que l'on puisse débog un code:
 - En enlevant les points virgules des déclarations de variables, ainsi l'on a le résultat dans l'invite de commande.

Débogage

■ Suite:

- Exécuter un script ou une fonction ligne par ligne, et ce en cliquant sur le bouton **Step** puis le bouton **Step in** représentés par les symboles suivants:



Figure:
But-
ton:
Step



Figure:
But-
ton:
StepIn

- Rajouter des **breakpoints** (points d'arrêt) au script afin de forcer l'exécution à s'arrêter à des lignes spécifiques, ceci est représenté par l'exemple suivant dans la documentation Matlab: [breakpoints](#).

**** Graphs ****

Affichage des variables

- Il existe plusieurs façons afin d'afficher le contenu d'une variable dans MATLAB, on rappelle qu'une variable dans MATLAB ne nécessite ni déclaration de type ou de dimension. Le type ainsi que la dimension d'une variable sont déterminés de manière automatique à partir de l'expression mathématique ou de la valeur affectée à la variable.

Affichage des variables - Suite

- Le lien suivant [displaying_vars.m](#) explicite les diverses manières avec lesquelles l'on pourrait afficher une variable dans MATLAB. Tout dépendra du type de la variable, et du résultat attendu.
- Deux façons de faire: soit l'on déclare une variable (locale ou globale) dans un script (extension .m) pour ensuite pouvoir l'utiliser dans une fonction à titre d'exemple, ou alors la déclarer directement dans le Command Window, et dans ce cas sa portée réside dans le même Command Window, un exemple est donné ci-dessous:

Affichage des variables - Suite

Command Window

```
» a = 1,2; b = 3; c = -64.9; z = 3 - 7i; y = 8 + 7j; a == b; myChar = 'd'; mylongChar = 'Ceci est un char'; myString = "Ceci est un string aussi"; whos
```

<i>Name</i>	<i>Size</i>	<i>Bytes</i>	<i>Class</i>	<i>Attributes</i>
<i>a</i>	1x1	8	<i>double</i>	
<i>ans</i>	1x1	1	<i>logical</i>	
<i>b</i>	1x1	8	<i>double</i>	
<i>c</i>	1x1	8	<i>double</i>	
<i>myChar</i>	1x1	2	<i>char</i>	
<i>myString</i>	1x1	190	<i>string</i>	
<i>mylongChar</i>	1x16	32	<i>char</i>	
<i>y</i>	1x1	16	<i>double</i>	<i>complex</i>
<i>z</i>	1x1	16	<i>double</i>	<i>complex</i>

Affichage des variables - Suite

- Par ailleurs, les variables déclarées dans le slide précédent, sont visibles *via* le **workspace**, ce dernier donne des informations quant à leur type, dimension, attribut et classe. La figure ci-dessous illustre les propos qui viennent d'être énoncés:

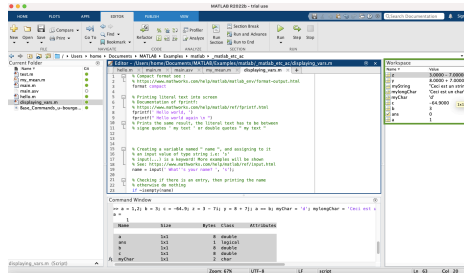


Figure: Workspace MATLAB

Les différents types de graphes

- Il existe une variété de types de graphes plots sur MATLAB, du 1D au 3D, des plots pour des probabilités (simuler une loi normale), des graphes pour de la géographie, des histogrammes etc. Ceci est une liste complète des différents graphes que l'on peut faire sur MATLAB:

- Line Plots
- Scatter and bubble charts
- Data distribution plots
- Discrete Data Plots
- Geographic Plots
- Polar Plots
- Contour plots
- Vector Fields
- Surface and Mesh Plots
- Volume Visualization
- Animation
- Images

Exemple d'un graphe

- Ci-dessous une visualisation d'un plot 2D de la fonction Sinus à titre d'exemple sur l'intervalle $[0, 2\pi]$:

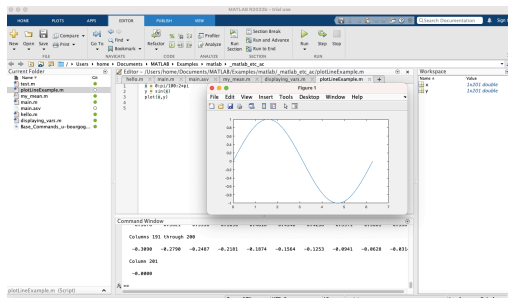


Figure: plot $\sin(x)$ on $[0, 2\pi]$

Les différents types de graphes

- À la fonction plot, on peut rajouter des options: changer de couleur du rendu, restreindre l'intervalle, rajouter des labels etc., ci-dessous un exemple complet de la même fonction sinus avec diverses options, ainsi que la fonction cosinus sur le même intervalle:

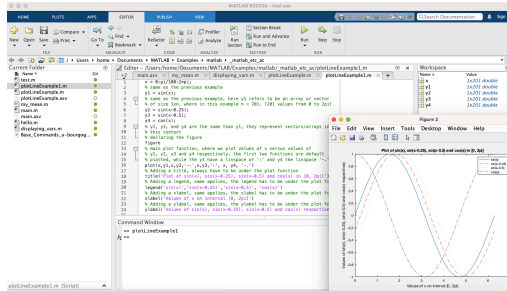


Figure: complete plot example

Menus et édition

- On suivra le tutoriel décrit par le lien suivant: [uimenu](#).

L'onglet PLOTS

- Comme déjà vu dans le slide 2, il existe différents types de graphes. On retrouve l'ensemble des possibilités de graphes *via* l'onglet **plots** et ce en sélectionnant une variable *X* au préalable. Tout dépendra du type de la variable *X*: array, vector, matrix etc. L'onglet **plots** se trouve comme montre la figure ci-dessous (en haut à gauche):

L'onglet PLOTS

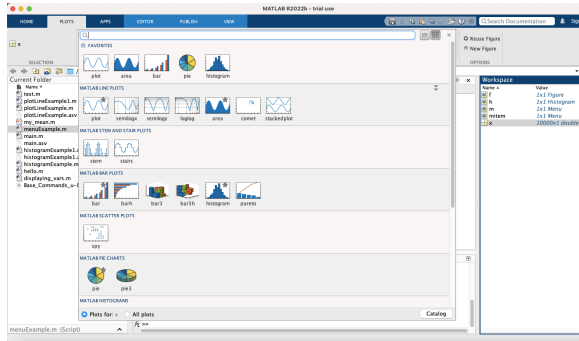


Figure: plots

**** GraphicalInterface ****

Introduction aux interfaces graphiques

Pour créer une **interface graphique (GUI)** en MATLAB, on peut utiliser l'environnement **GUIDE** (Graphical User Interface Development Environment). GUIDE permet de créer des interfaces utilisateur interactives de manière visuelle.

Prise en main de GUIDE

Pour ouvrir **GUIDE**, vous pouvez taper `guide` dans la fenêtre de commande MATLAB. Cela ouvrira l'environnement **GUIDE** où vous pouvez concevoir et éditer votre interface utilisateur.

Les éléments de contrôle

- Les éléments de contrôle sont les composants graphiques que vous pouvez ajouter à votre interface utilisateur. Voici quelques-uns des éléments de contrôle couramment utilisés :
 - Bouton (PushButton)
 - Case à cocher (CheckBox)
 - Liste déroulante (DropDown)
 - Zone de texte (EditText)
 - Graphique (Axes)
- Vous pouvez ajouter ces éléments en faisant glisser et en déposant depuis la palette d'outils de **GUIDE**.

Callbacks et Handles

- Les **callbacks** sont des fonctions MATLAB qui sont déclenchées en réponse à une action de l'utilisateur sur un élément de contrôle.
- Les **handles** sont des identifiants uniques pour chaque élément de contrôle de l'interface utilisateur.
- Par exemple, pour exécuter une fonction lorsque l'utilisateur clique sur un bouton, vous pouvez définir la fonction dans le callback `Callback` de ce bouton.

Graphes

Les graphes peuvent être affichés dans une interface graphique en utilisant l'élément de contrôle `Axes`. Vous pouvez tracer des graphiques en utilisant des fonctions telles que `plot`, `bar`, `surf`, etc. et en spécifiant l'axe sur lequel tracer le graphe.

Par exemple :

Listing 8: Example of custom exception

```
1 x = linspace(0, 2*pi, 100);  
2 y = sin(x);  
3 plot(handles.axes1, x, y);
```