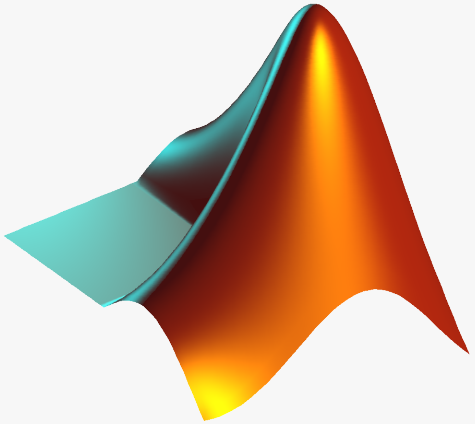
Université de Montréal



Séminaire MATLAB

MATLAB à vol d'oiseau

Damien Rioux Lavoie
Département de mathématiques et de statistique

30 juillet 2014

Déroulement de l'exposé

Université de Montréal

Bases

Images et animations

Structure de données

Outils et concepts avancées

Références

MATLAB, qu'est-ce que c'est au juste ?

Université de Montréal

MATLAB peut être décrit comme :

- un calculateur graphique très puissant ;
- un langage de programmation interprété et de haut niveau.

Pourquoi utiliser MATLAB ?

Université de Montréal

Idéal pour :

- les manipulations matricielles ;
- l'analyse numérique ;
- l'analyse de données ;
- la modélisation (simulink).

Pourquoi utiliser MATLAB ?

Université de Montréal

Idéal pour :

- les manipulations matricielles ;
- l'analyse numérique ;
- l'analyse de données ;
- la modélisation (simulink).

MATLAB offre, entre autres :

- une large banque de fonctions ;
- un manuel d'utilisation complet ;
- une communauté en ligne fournissant continuellement des nouvelles fonctions et *toolbox* ;
- un language simple à apprendre et à implémenter.

Environnement

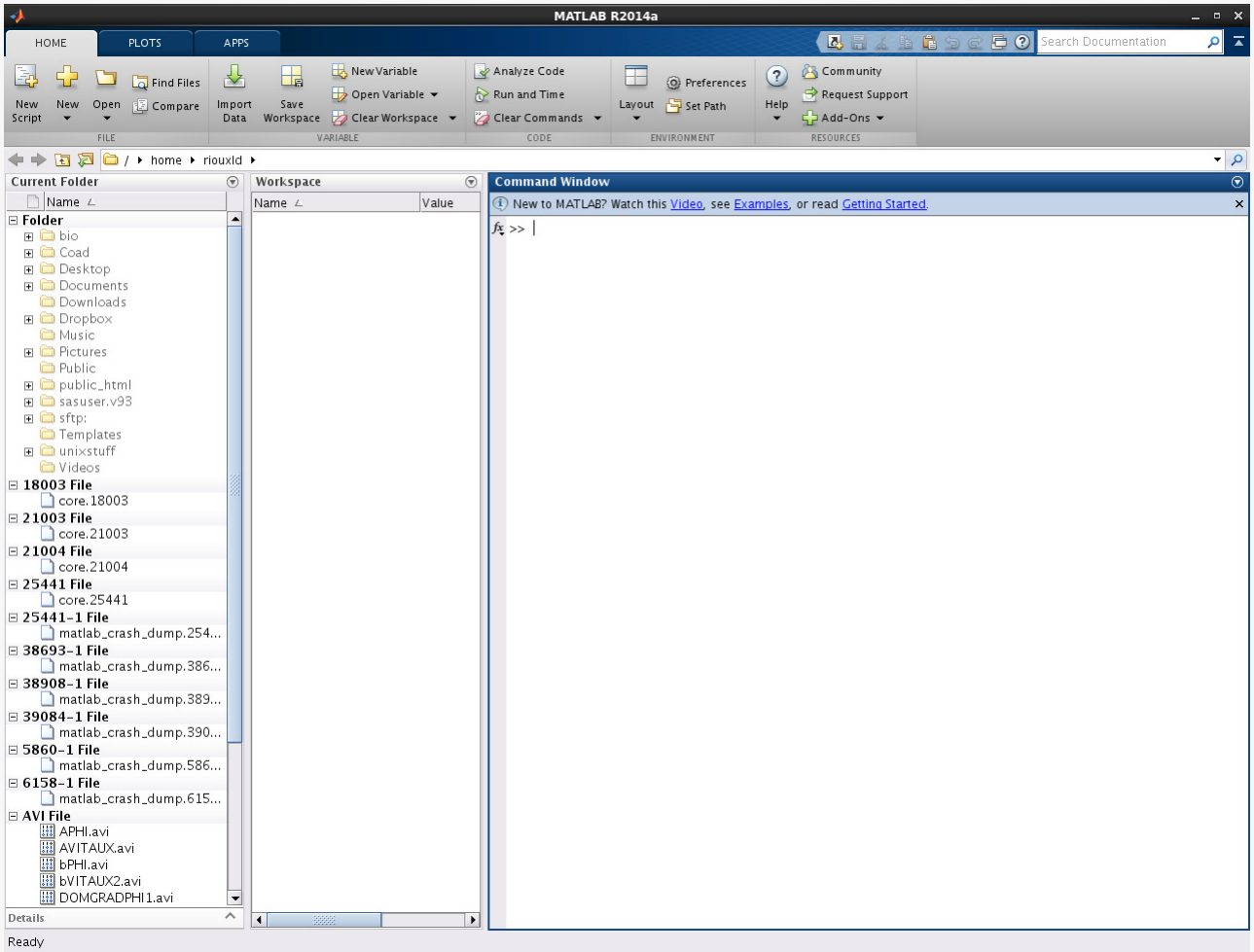


Figure : Environnement MATLAB.

Vous trouverez, sur le bureau :

- *Toolbar* : barre d'outils ;
- *Current folder* : contenu du dossier courant ;
- *Command window* : endroit où les commandes entrées sont affichées ;
- *Workspace* : endroit sont sauvegardées les variables.

Voici quelques commandes élémentaires utiles :

- `help` : donne des informations sur une commande ;
- `clear` : supprime des variables ;
- `clc` : efface le *Command window* ;
- `save`/`load` : sauvegarde/charge des variables dans un fichier *.mat* ;
- `size`/`length` : donne le format/longueur d'une matrice/vecteur ;
- `diary` : sauvegarde l'affichage dans un fichier ;
- `format` : change le format d'affichage ;
- `tic`/`toc` : calcule le temps écoulé ;
- `for`/`while`/`if` : structures de contrôle de programmation.

Plusieurs types de variables sont implémentés par MATLAB : `double`, `int`, `char`, `struct`, `cell` et plus encore.

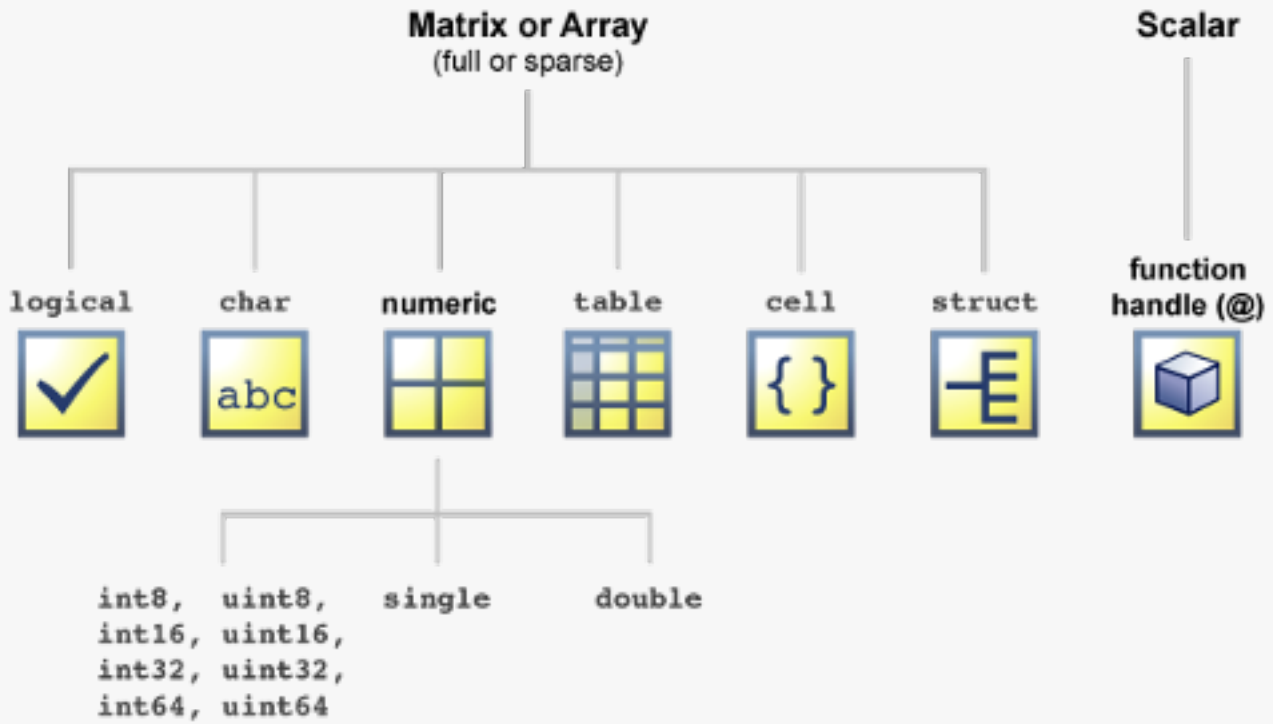


Figure : Types de variables fondamentales.

Un *array* est une structure rectangulaire possédant des valeurs de même types. Notons que, dans MATLAB, Toutes les variables sont des *array*, à l'exception des *function handle*.

Un *array* est une structure rectangulaire possédant des valeurs de même types. Notons que, dans MATLAB, Toutes les variables sont des *array*, à l'exception des *function handle*.

- *logical*
  - Booléen : *true* ou *false*.

Un *array* est une structure rectangulaire possédant des valeurs de même types. Notons que, dans MATLAB, Toutes les variables sont des *array*, à l'exception des *function handle*.

- *logical*
  - Booléen : *true* ou *false*.
- *char*
  - Chaînes de caractère entre guillemets.

Un *array* est une structure rectangulaire possédant des valeurs de même types. Notons que, dans MATLAB, Toutes les variables sont des *array*, à l'exception des *function handle*.

- *logical*
  - Booléen : *true* ou *false*.
- *char*
  - Chaînes de caractère entre guillemets.
- *numeric*
  - Nombre en virgule flottante.
  - *double* : Précision double (*64 bits*).
  - *single* : Précision simple (*32 bits*).

Variable

Université de Montréal

- Nul besoin d'initialiser les variables, il suffit d'assigner une valeur à un nom :

- Nul besoin d'initialiser les variables, il suffit d'assigner une valeur à un nom :

```
>> Prenom = 'Damien'
Prenom =
Damien
```

- Nul besoin d'initialiser les variables, il suffit d'assigner une valeur à un nom :

```
>> Prenom = 'Damien'
Prenom =
Damien
```

- Restrictions pour les noms des variables :
  - commence par une lettre ;
  - lettres, chiffres et _ sont acceptés ;
  - sensible à la casse ;
  - éviter `i`, `j`, `pi`, `ans`, `Inf`, `realmin`, `realmax` et `NaN` car ce sont des variables prédéfinies dans MATLAB.

Variable

Université de Montréal

- Nul besoin d'initialiser les variables, il suffit d'assigner une valeur à un nom :

```
>> Prenom = 'Damien'
Prenom =
Damien
```

- Restrictions pour les noms des variables :
  - commence par une lettre ;
  - lettres, chiffres et _ sont acceptés ;
  - sensible à la casse ;
  - éviter `i`, `j`, `pi`, `ans`, `Inf`, `realmin`, `realmax` et `NaN` car ce sont des variables prédéfinies dans MATLAB.
- Peuvent être définis par le résultat d'une fonction :

- Nul besoin d'initialiser les variables, il suffit d'assigner une valeur à un nom :

```
>> Prenom = 'Damien'
Prenom =
Damien
```

- Restrictions pour les noms des variables :
  - commence par une lettre ;
  - lettres, chiffres et _ sont acceptés ;
  - sensible à la casse ;
  - éviter `i`, `j`, `pi`, `ans`, `Inf`, `realmin`, `realmax` et `NaN` car ce sont des variables prédéfinies dans MATLAB.
- Peuvent être définis par le résultat d'une fonction :

```
>> Nom_Complet = strcat(Prenom, '_','Lavoie')
Nom_Complet =
Damien_Lavoie
```

Bien entendu, il est souhaitable de sauvegarder dans des fichiers les lignes de code que nous écrivons. Pour ce faire, nous pouvons écrire dans un fichier qui a comme extension *.m*, appellé *m-files*, notre code. Le code est écrit grâce à l'éditeur de MATLAB. Il y a deux façons de créer un script :

Bien entendu, il est souhaitable de sauvegarder dans des fichiers les lignes de code que nous écrivons. Pour ce faire, nous pouvons écrire dans un fichier qui a comme extension *.m*, appellé *m-files*, notre code. Le code est écrit grâce à l'éditeur de MATLAB. Il y a deux façons de créer un script :

- *inline* ;

Bien entendu, il est souhaitable de sauvegarder dans des fichiers les lignes de code que nous écrivons. Pour ce faire, nous pouvons écrire dans un fichier qui a comme extension *.m*, appellé *m-files*, notre code. Le code est écrit grâce à l'éditeur de MATLAB. Il y a deux façons de créer un script :

- *inline* ;

```
>> edit Base.m
```

Bien entendu, il est souhaitable de sauvegarder dans des fichiers les lignes de code que nous écrivons. Pour ce faire, nous pouvons écrire dans un fichier qui a comme extension *.m*, appellé *m-files*, notre code. Le code est écrit grâce à l'éditeur de MATLAB. Il y a deux façons de créer un script :

- *inline* ;

  ```
  >> edit Base.m
  ```

- dans la section *Toolbar* ou *Current Directory*.

Lorsque nous avons besoin d'utiliser plusieurs fois une même portion de code dans notre script, il est parfois judicieux d'utiliser une fonction. Nous évitons ainsi les erreurs de copie tout en rendant le code plus lisible. Ceci est fait en écrivant la fonction dans un *m-file* avec la syntaxe :

Lorsque nous avons besoin d'utiliser plusieurs fois une même portion de code dans notre script, il est parfois judicieux d'utiliser une fonction. Nous évitons ainsi les erreurs de copie tout en rendant le code plus lisible. Ceci est fait en écrivant la fonction dans un *m-file* avec la syntaxe :

```
function [z] = ExempleGraphique(x, y)
% EXAMPLEGRAPHIQUE Ce qui est affiché par help
z = exp(-sqrt(x.^ 2+y.^ 2)).*cos(x).*sin(y);
% ";" à la fin de la ligne sert à ne pas afficher
% la commande exécutée
end
```

Lorsque nous avons besoin d'utiliser plusieurs fois une même portion de code dans notre script, il est parfois judicieux d'utiliser une fonction. Nous évitons ainsi les erreurs de copie tout en rendant le code plus lisible. Ceci est fait en écrivant la fonction dans un *m-file* avec la syntaxe :

```
function [z] = ExempleGraphique(x, y)
% EXAMPLEGRAPHIQUE Ce qui est affiché par help
z = exp(-sqrt(x.^ 2+y.^ 2)).*cos(x).*sin(y);
% ";" à la fin de la ligne sert à ne pas afficher
% la commande exécutée
end
```

Attention, le nom donné au fichier doit être le même que le nom de la fonction !.

Lorsque nous avons besoin d'utiliser plusieurs fois une même portion de code dans notre script, il est parfois judicieux d'utiliser une fonction. Nous évitons ainsi les erreurs de copie tout en rendant le code plus lisible. Ceci est fait en écrivant la fonction dans un *m-file* avec la syntaxe :

```
function [z] = ExempleGraphique(x, y)
% EXAMPLEGRAPHIQUE Ce qui est affiché par help
z = exp(-sqrt(x.^ 2+y.^ 2)).*cos(x).*sin(y);
% ";" à la fin de la ligne sert à ne pas afficher
% la commande exécutée
end
```

Attention, le nom donné au fichier doit être le même que le nom de la fonction !.
Par exemple, il faudrait, ici, donner *ExempleGraphique.m* comme nom à notre fichier.

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

```
>> Vector_Ligne1 = [1, 2], Vector_Ligne2 = [1 2];
Vector_Ligne1 =
     1     2
```

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

```
>> Vector_Ligne1 = [1, 2], Vector_Ligne2 = [1 2];
Vector_Ligne1 =
     1     2
```

- Vecteur colonne : on sépare les valeurs par un point-virgule :

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

```
>> Vector_Ligne1 = [1, 2], Vector_Ligne2 = [1 2];
Vector_Ligne1 =
     1     2
```

- Vecteur colonne : on sépare les valeurs par un point-virgule :

```
>> Vector_Colonne = [1; 2]
Vector_Colonne =
     1
     2
```

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

```
>> Vector_Ligne1 = [1, 2], Vector_Ligne2 = [1 2];
Vector_Ligne1 =
     1     2
```

- Vecteur colonne : on sépare les valeurs par un point-virgule :

```
>> Vector_Colonne = [1; 2]
Vector_Colonne =
     1
     2
```

On accède à une composante d'un vecteur grâce à l'indice correspondant :

- Vecteur ligne : On sépare les valeurs par une espaces ou une virgules :

```
>> Vector_Ligne1 = [1, 2], Vector_Ligne2 = [1 2];
Vector_Ligne1 =
     1     2
```

- Vecteur colonne : on sépare les valeurs par un point-virgule :

```
>> Vector_Colonne = [1; 2]
Vector_Colonne =
     1
     2
```

On accède à une composante d'un vecteur grâce à l'indice correspondant :

```
>> Vector_Colonne(2)
ans =
     2
```

Matrice

Il s'agit de la généralisation du vecteur. Les virgules séparent les éléments d'une même ligne et les point-virgules les lignes elles-mêmes :

Matrice

Université de Montréal

Il s'agit de la généralisation du vecteur. Les virgules séparent les éléments d'une même ligne et les point-virgules les lignes elles-mêmes :

```
>> Matrix = [1, 2; 3, 4]
Matrix =
     1     2
     3     4
```

Matrice

Université de Montréal

Il s'agit de la généralisation du vecteur. Les virgules séparent les éléments d'une même ligne et les point-virgules les lignes elles-mêmes :

```
>> Matrix = [1, 2; 3, 4]
Matrix =
     1     2
     3     4
```

Pour accéder à un élément nous utilisons deux indices, le premier pour la ligne, le second pour la colonne :

Matrice

Il s'agit de la généralisation du vecteur. Les virgules séparent les éléments d'une même ligne et les point-virgules les lignes elles-mêmes :

```
>> Matrix = [1, 2; 3, 4]
Matrix =
     1     2
     3     4
```

Pour accéder à un élément nous utilisons deux indices, le premier pour la ligne, le second pour la colonne :

```
>> Matrix(1, :)
ans =
     1    2
```

Il s'agit de la généralisation du vecteur. Les virgules séparent les éléments d'une même ligne et les point-virgules les lignes elles-mêmes :

```
>> Matrix = [1, 2; 3, 4]
Matrix =
     1     2
     3     4
```

Pour accéder à un élément nous utilisons deux indices, le premier pour la ligne, le second pour la colonne :

```
>> Matrix(1, :)
ans =
     1    2
```

La fonction `zeros` est très utile pour initialiser des matrices de grande taille. En effet, souvent changer la taille d'une matrice peut prendre beaucoup de temps.

MATLAB offre une vaste quantité de fonctions permettant la visualisation des données. En voici quelques exemples :

- 2D : `plot`, `scatter`, `loglog`, `bar`, `contour`, `image`, `quiver`, `spy`, `polar` ;
- 3D : `line3`, `contour3`, `mesh`, `surf`, `quiver3`, `streamslice`, `scatter3`, `bar3`.

MATLAB offre une vaste quantité de fonctions permettant la visualisation des données. En voici quelques exemples :

- 2D : `plot`, `scatter`, `loglog`, `bar`, `contour`, `image`, `quiver`, `spy`, `polar` ;
- 3D : `line3`, `contour3`, `mesh`, `surf`, `quiver3`, `streamslice`, `scatter3`, `bar3`.

Toutes ces fonctions sont accompagnées d'une grande quantité d'options (voir *LineSpec* et `set`), permettant le changement de texture, de couleur, l'ajout d'un titre et ainsi de suite.

```
x = linspace(-4*pi, 4*pi, 200);
y1 = sin(x); y2 = cos(x);
Handle_Plot = figure(1);
plot(x, y1)
hold on
plot(x, y2, 'r')
hold off
xlim([-4*pi, 4*pi])
xlabel('Longueur dans la direction x')
ylim([-1, 1])
ylabel('Hauteur dans la direction y')
legend('Sinus','Cosinus', 'location', 'NorthEastOutside')
title('Exemple Plot')
saveas(Handle_Plot, 'Exemple_Plot', 'fig')
close(gcf)
```

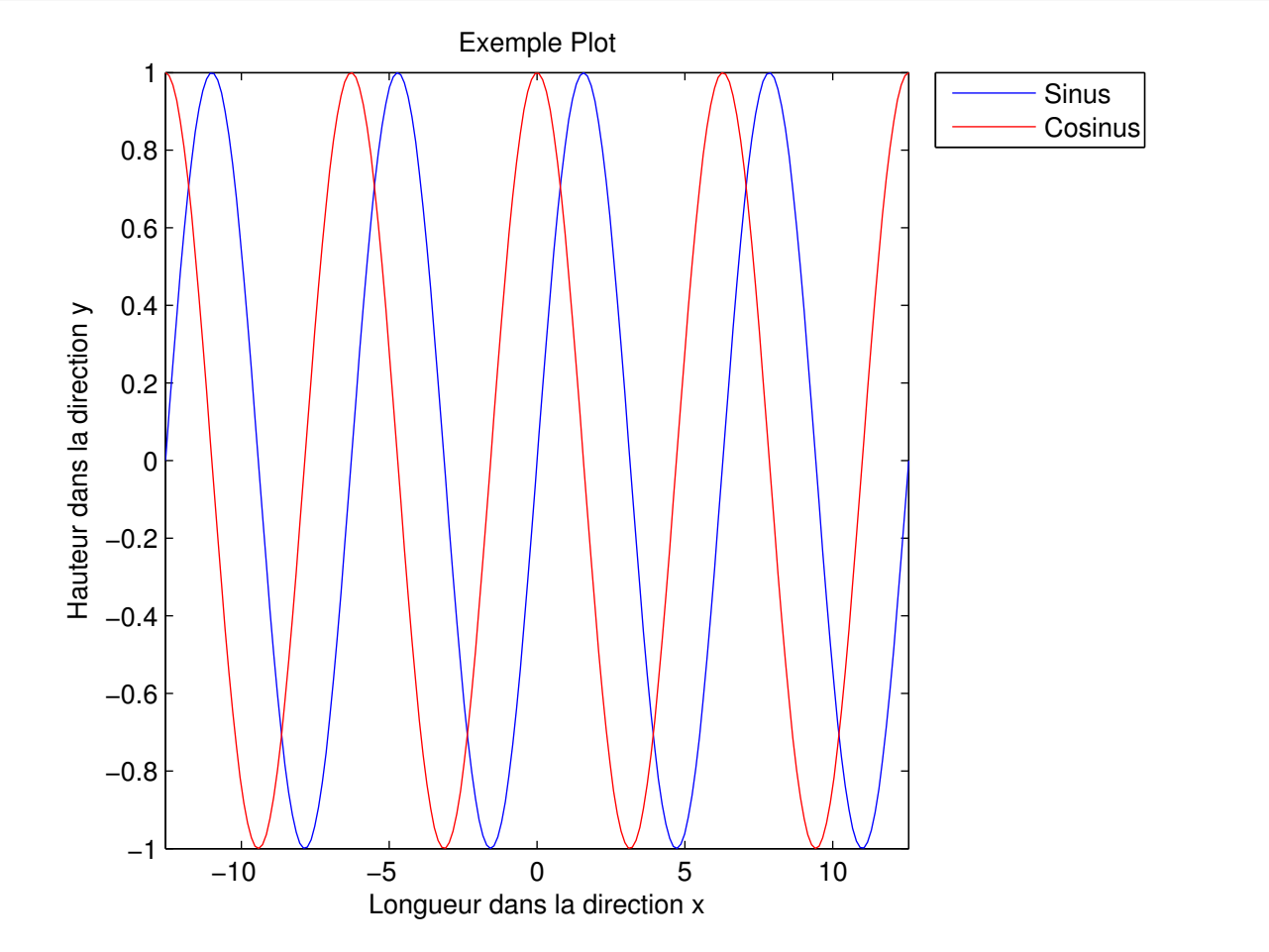


Figure : Exemple de graphique avec `plot`.

```
xgp = linspace(-pi, pi, 100);
ygp = linspace(-pi, pi, 100);
[X, Y] = meshgrid(xgp,ygp);
Z = ExempleGraphique(X, Y);
Color = gradient(Z);
Handle_Surf = figure(2);
surf(X,Y,Z,Color); colorbar
xlim([-pi, pi])
xlabel('Longueur dans la direction x')
ylim([-pi, pi])
ylabel('Longueur dans la direction y')
zlabel('Hauteur dans la direction z')
title('Exemple Surf')
saveas(Handle_Surf, 'Exemple_Surf', 'fig')
close(gcf)
```

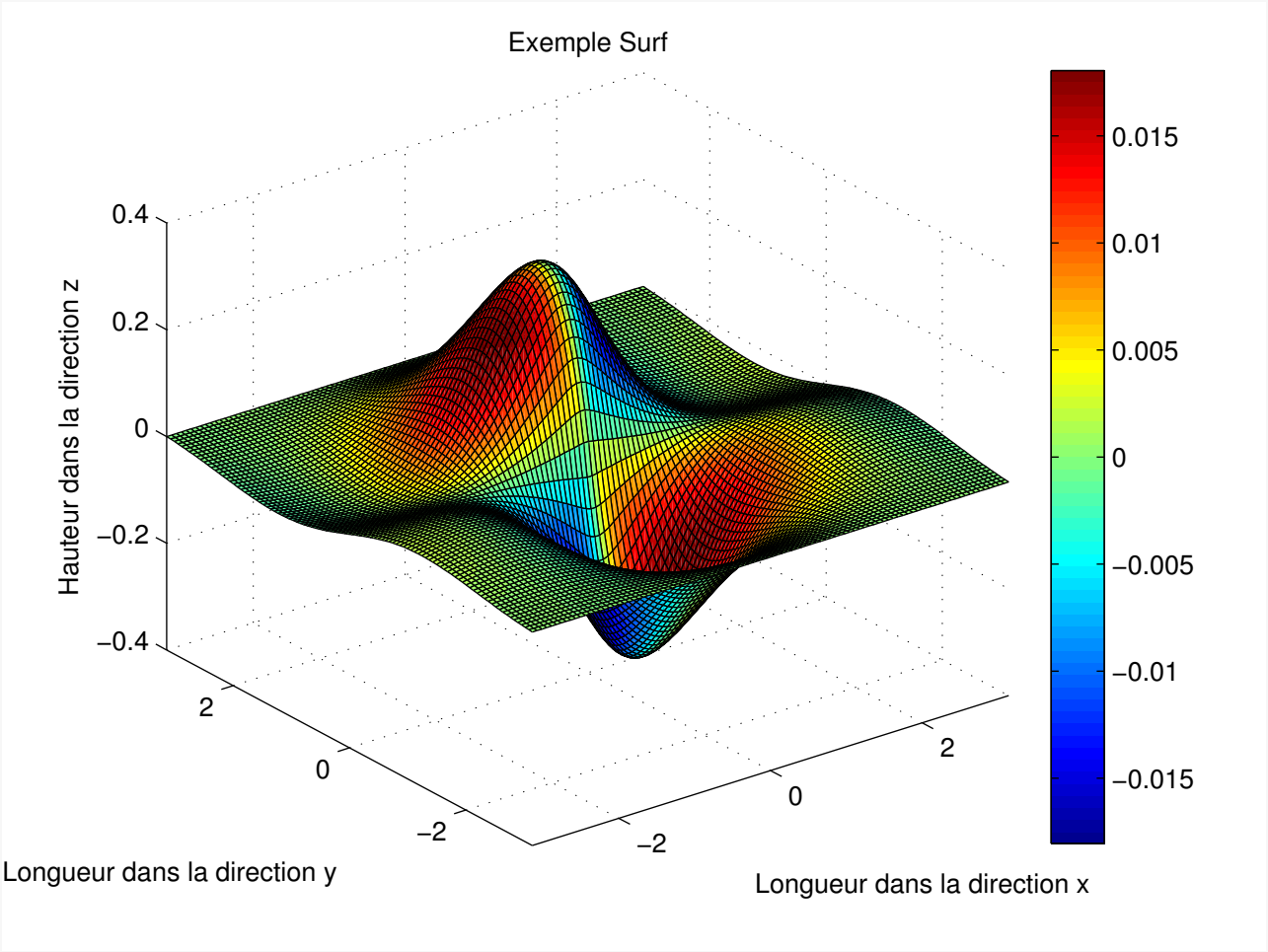


Figure : Exemple de graphique avec `surf`.

```
writerObj = VideoWriter('Exemple_Animation.avi');
open(writerObj);
Z = ExempleGraphique(X, Y);
surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
for k = 1:50
    surf(sin(2*pi*k/20)*Z,Z)
    frame = getframe;
    writeVideo(writerObj,frame);
end
close(writerObj);
```

Ceci nous donne comme résultat : Animation

Nous avons déja présenté la structure de données *array*. Ce type de structure a comme caractéristique :

Nous avons déja présenté la structure de données *array*. Ce type de structure a comme caractéristique :

- peut être généralisée à *n*-dimensions ;

Nous avons déja présenté la structure de données *array*. Ce type de structure a comme caractéristique :

- peut être généralisée à *n*-dimensions ;
- tous les éléments doivent être de même type ;

Nous avons déja présenté la structure de données *array*. Ce type de structure a comme caractéristique :

- peut être généralisée à *n*-dimensions ;
- tous les éléments doivent être de même type ;
- on accède aux éléments à l'aide d'indices.

- Permet de stocker des éléments de types différents.

Cell array

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

```
Cell_Array1 = cell(2,2);
Cell_Array1{1,1} = 'Damien'; Cell_Array1{1,2} = 25;
Cell_Array1{2,1} = [1, 2]; Cell_Array1{2,2} = true;
```

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

```
Cell_Array1 = cell(2,2);
Cell_Array1{1,1} = 'Damien'; Cell_Array1{1,2} = 25;
Cell_Array1{2,1} = [1, 2]; Cell_Array1{2,2} = true;
```

  - en listant les éléments.

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

```
Cell_Array1 = cell(2,2);
Cell_Array1{1,1} = 'Damien'; Cell_Array1{1,2} = 25;
Cell_Array1{2,1} = [1, 2]; Cell_Array1{2,2} = true;
```

  - en listant les éléments.

```
Cell_Array2 = {'Damien', 25; [1, 2], true};
```

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

```
Cell_Array1 = cell(2,2);
Cell_Array1{1,1} = 'Damien'; Cell_Array1{1,2} = 25;
Cell_Array1{2,1} = [1, 2]; Cell_Array1{2,2} = true;
```

  - en listant les éléments.

```
Cell_Array2 = {'Damien', 25; [1, 2], true};
```

- Pour accéder aux éléments, il faut spécifier les indices entre accolades $\{\}$:

- Permet de stocker des éléments de types différents.
- Deux méthodes pour les initialiser :
  - en utilisant la fonction `cell` ;

```
Cell_Array1 = cell(2,2);
Cell_Array1{1,1} = 'Damien'; Cell_Array1{1,2} = 25;
Cell_Array1{2,1} = [1, 2]; Cell_Array1{2,2} = true;
```

  - en listant les éléments.

```
Cell_Array2 = {'Damien', 25; [1, 2], true};
```

- Pour accéder aux éléments, il faut spécifier les indices entre accolades {} :

```
>> Cell_Array1{1,2}
ans =
    25
```

- Permet de stocker des éléments de types différents.

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

```
Struct1 = struct;
Struct1.Prenom = 'Damien'; Struct1.Age = 25;
Struct1.Position = [1,2];
```

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

```
Struct1 = struct;
Struct1.Prenom = 'Damien'; Struct1.Age = 25;
Struct1.Position = [1,2];
```

- Alternativement, on peut initialiser la *struct* avec des paires champ-valeur :

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

```
Struct1 = struct;
Struct1.Prenom = 'Damien'; Struct1.Age = 25;
Struct1.Position = [1,2];
```

- Alternativement, on peut initialiser la *struct* avec des paires champ-valeur :

```
Struct2 = struct('Prenom', 'Damien', 'Age', 25);
```

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

```
Struct1 = struct;
Struct1.Prenom = 'Damien'; Struct1.Age = 25;
Struct1.Position = [1,2];
```

- Alternativement, on peut initialiser la *struct* avec des paires champ-valeur :

```
Struct2 = struct('Prenom', 'Damien', 'Age', 25);
```

- Pour accéder à un champ, on utilise la syntaxe s.champs.

- Permet de stocker des éléments de types différents.
- Pour les initialiser, on utilise la fonction `struct` puis on ajoute des champs à l'aide de `.`, suivi du nom du champ :

```
Struct1 = struct;
Struct1.Prenom = 'Damien'; Struct1.Age = 25;
Struct1.Position = [1,2];
```

- Alternativement, on peut initialiser la *struct* avec des paires champ-valeur :

```
Struct2 = struct('Prenom', 'Damien', 'Age', 25);
```

- Pour accéder à un champ, on utilise la syntaxe s.champs.

```
>> Struct1.Age
ans =
    25
```

Function handle

Université de Montréal

- Il s'agit du seul type de variable qui n'est pas un *array*.

Function handle

Université de Montréal

- Il s'agit du seul type de variable qui n'est pas un *array*.
- Il s'agit d'un pointeur vers une fonction et permet de passer une fonction en argument à une autre fonction.

Function handle

Université de Montréal

- Il s'agit du seul type de variable qui n'est pas un *array*.
- Il s'agit d'un pointeur vers une fonction et permet de passer une fonction en argument à une autre fonction.
- Pour créer un *function handle* nous utilisons le symbole `@` :

Function handle

Université de Montréal

- Il s'agit du seul type de variable qui n'est pas un *array*.
- Il s'agit d'un pointeur vers une fonction et permet de passer une fonction en argument à une autre fonction.
- Pour créer un *function handle* nous utilisons le symbole `@` :

```
sqr = @(x) x.^2;
a = sqr(5), integral(sqr, 0, 1)
a =
    25
ans =
   0.333333333333333
```

- Il s'agit du seul type de variable qui n'est pas un *array*.
- Il s'agit d'un pointeur vers une fonction et permet de passer une fonction en argument à une autre fonction.
- Pour créer un *function handle* nous utilisons le symbole `@` :

```
sqr = @(x) x.^2;
a = sqr(5), integral(sqr, 0, 1)
a =
    25
ans =
   0.333333333333333
```

`@` est souvent couplé avec la fonction `feval`.

- Puisque MATLAB est un langage interprété, les boucles sont à éviter lorsque possible.

- Puisque MATLAB est un langage interprété, les boucles sont à éviter lorsque possible.
- Heureusement, MATLAB est optimisé pour le traitement des matrices et nous pouvons souvent remplacer des boucles par des opérations matricielles.

- Puisque MATLAB est un langage interprété, les boucles sont à éviter lorsque possible.
- Heureusement, MATLAB est optimisé pour le traitement des matrices et nous pouvons souvent remplacer des boucles par des opérations matricielles.
- La plupart des fonctions, ainsi que les opérateurs `.*`, `.^` et `./`, s'appliquent à tous les éléments d'une matrice (*entry-wise*).

Voici un exemple de code non-vectorisé :

Voici un exemple de code non-vectorisé :

```
tic, x = 1:10000; ylength = (length(x)-mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
    y(n/5) = sum(x(1:n));
end, TempsBoucle = toc
TempsBoucle =
     8.586812290388862e+00
```

Voici un exemple de code non-vectorisé :

```
tic, x = 1:10000; ylength = (length(x)-mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
    y(n/5) = sum(x(1:n));
end, TempsBoucle = toc
TempsBoucle =
     8.586812290388862e+00
```

Après vectorisation, il devient :

Voici un exemple de code non-vectorisé :

```
tic, x = 1:10000; ylength = (length(x)-mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
    y(n/5) = sum(x(1:n));
end, TempsBoucle = toc
TempsBoucle =
     8.586812290388862e+00
```

Après vectorisation, il devient :

```
tic, x = 1:100000; xsums = cumsum(x);
y = xsums(5:5:length(x)); TempsVecto = toc
TempsVecto =
     1.239333490709015e-03
```

Fichier MEX

Université de Montréal

Les fichiers MEX sont des fichiers écrits en C ou en CUDA qui, une fois compilés, peuvent être utilisés dans MATLAB de la même manière que des fonctions.

Les fichiers MEX sont des fichiers écrits en C ou en CUDA qui, une fois compilés, peuvent être utilisés dans MATLAB de la même manière que des fonctions.

- Permets l'utilisation de code déjà fait dans ces langages.

Fichier MEX

Université de Montréal

Les fichiers MEX sont des fichiers écrits en C ou en CUDA qui, une fois compilés, peuvent être utilisés dans MATLAB de la même manière que des fonctions.

- Permets l'utilisation de code déjà fait dans ces langages.
- Peut accélérer l'exécution du code.

Les fichiers MEX sont des fichiers écrits en C ou en CUDA qui, une fois compilés, peuvent être utilisés dans MATLAB de la même manière que des fonctions.

- Permets l'utilisation de code déjà fait dans ces langages.
- Peut accélérer l'exécution du code.
- Permet l'appel de fonctions des bibliothèques de ces languages.

Les fichiers MEX sont des fichiers écrits en C ou en CUDA qui, une fois compilés, peuvent être utilisés dans MATLAB de la même manière que des fonctions.

- Permets l'utilisation de code déjà fait dans ces langages.
- Peut accélérer l'exécution du code.
- Permet l'appel de fonctions des bibliothèques de ces languages.
- Prends toujours 4 arguments :
  - *nlhs* : nombre d'arguments de sortie ;
  - *plhs* : tableau de pointeurs contenant les sorties ;
  - *nrhs* : nombre d'arguments d'entrée ;
  - *prhs* : tableau de pointeurs contenant les entrées.

D'abord, un code dans un de ces langages doit être écrit.

Fichier MEX

Université de Montréal

D'abord, un code dans un de ces langages doit être écrit.

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[] ) {
mexPrintf("Hello World !\n"); }
```

Fichier MEX

Université de Montréal

D'abord, un code dans un de ces langages doit être écrit.

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[] ) {
mexPrintf("Hello World !\n"); }
```

Ensuite, nous pouvons le compiler à l'aide de la commande `mex`.

D'abord, un code dans un de ces langages doit être écrit.

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[] ) {
mexPrintf("Hello World !\n"); }
```

Ensuite, nous pouvons le compiler à l'aide de la commande `mex`.

```
>>mex HelloWorld.c
Building with 'gcc'.
MEX completed successfully.
```

D'abord, un code dans un de ces langages doit être écrit.

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[] ) {
mexPrintf("Hello World !\n"); }
```

Ensuite, nous pouvons le compiler à l'aide de la commande `mex`.

```
>>mex HelloWorld.c
Building with 'gcc'.
MEX completed successfully.
```

Nous pouvons maintenant l'appeler comme une fonction.

D'abord, un code dans un de ces langages doit être écrit.

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[] ) {
mexPrintf("Hello World !\n"); }
```

Ensuite, nous pouvons le compiler à l'aide de la commande `mex`.

```
>>mex HelloWorld.c
Building with 'gcc'.
MEX completed successfully.
```

Nous pouvons maintenant l'appeler comme une fonction.

```
>> HelloWorld
Hello World !
```

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

- *.txt* : importer des structures à l'aide de `importdata` ;

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

- *.txt* : importer des structures à l'aide de `importdata` ;
- *.dat* : importer des *cell array* à l'aide de `textscan`, `fopen` et `fclose` ;

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

- *.txt* : importer des structures à l'aide de `importdata` ;
- *.dat* : importer des *cell array* à l'aide de `textscan`, `fopen` et `fclose` ;
- *csv* : importer des matrices numériques à l'aide de `csvread`.

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

- *.txt* : importer des structures à l'aide de `importdata` ;
- *.dat* : importer des *cell array* à l'aide de `textscan`, `fopen` et `fclose` ;
- *csv* : importer des matrices numériques à l'aide de `csvread`.

Il est aussi possible d'exporter des variables dans des fichiers *.csv* à l'aide de `csvwrite`.

Dans MATLAB, il est possible d'importer des variables à partir de fichiers *.txt*, *.dat* et *.xls*. En effet, nous pouvons :

- *.txt* : importer des structures à l'aide de `importdata` ;
- *.dat* : importer des *cell array* à l'aide de `textscan`, `fopen` et `fclose` ;
- *csv* : importer des matrices numériques à l'aide de `csvread`.

Il est aussi possible d'exporter des variables dans des fichiers *.csv* à l'aide de `csvwrite`.
Sur windows, vous pouvez aussi gérer le format *.xls* avec `xlsread` et `xlswrite`.

Olivier. Ricou. Exemple de fichier mex en c, 1997. URL http://www.ricou.eu.org/matlab/cours3/node9.html.

Kermit Sigmon. *Matlab Primer*. Third edition, 1993.

Danilo. Šćepanović. Introduction to matlab, 2010. URL http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-094-introduction-to-matlab-january-iap-2010.