# PROGRAMMATION SOUS MATLAB

D. Legland

29 mai 2015

# Table des matières

# 1. Introduction

Ce document rassemble différents éléments de pratique de programmation sous Matlab. Au fur et a mesure de la rédaction, il apparait qu'il pourrait servir de tutoriel sur la progammation Matlab, en particulier sur la programmation orientée objets (POO).

La première partie concerne des généralités sur le langages. Le chapitre 3 pose les conventions que j'utilise. La deuxième partie reprend mes notes sur les structures de données et la programmation orientée objets (chapitres 4, 5 et 6). La troisième partie aborde des thèmes plus techniques, qui reposent le plus souvent sur la manipulation et / ou la création de classes : la conception d'interfaces graphiques (chapitre 8), la gestion des exceptions et des erreurs (chapitre 9), et enfin une proposition d'architecture pour manipuler les évènements Matlab de manière similaire à Java (chapitre 10).

Première partie .

# Programmer avec Matlab

# 2. Syntaxe de base

Ce chapitre rappelle très brièvement quelques éléments fondamentaux de la programmation sous Matlab : les types de données, et les structures de contrôle (tests, boucles...).

## 2.1. Types de données

Les données manipulées peuvent être :

— des nombres en virgule flottante,
— des nombres complexes (du type $a + ib$)
— des nombres entiers (signés ou non)
— des chaînes de caractères
— des tableaux (de valeurs ou de cellules)

## 2.2. Tests

### 2.2.1. Tests if-then-else

On découpe le programme en plusieurs blocs, et on execute un bloc ou l'autre en fonction d'une condition.

```
1 if a > b
2     res = a;
3 else
4     res = b;
5 end
```

### 2.2.2. switch

Permet de gérer plusieurs tests en même temps. Très pratique avec les chaînes de caractères.

```
1 switch option
2     case 'vrai'
3         disp('OK');
4     case 'faux'
5         disp('Pas bon');
6     default
7         error(['Valeur inconnue pour option: ' option]);
8 end
```

## 2.3. Boucles

### 2.3.1. Boucles for

Outre la boucle « for » classique, on peut utiliser d'autres syntaxes.

#### 2.3.1.1. Boucle for classique

```
1 for i = 1:5
2     disp(2 i);
3 end
```

#### 2.3.1.2. Itération sur des tableaux

Il est possible d'utiliser les tableaux à la manière des collections en Java. Exemple :

```
1 tab = 1:5;
2 for i = tab
3     disp(i);
4 end
```

Note : on itère en fait sur les colonnes (c'est à dire sur la dimension 2). Le reste du tableau est simplifié. On peut aussi itérer sur un tableau de cellules :

```
1 inds = cellstr(num2str((1:5)', 'ind%d'))';
2 for i = inds
3     disp(i{1});
4 end
```

Là aussi on itère sur les colonnes du tableau, d'où la deuxième transposition du tableau « inds ».

#### 2.3.1.3. Boucle for parallèle

Il s'agit de la même syntaxe que la boucle for, mais cette version permet de paralléliser les calculs faits dans la boucle. Ne doit pas être utilisé si les calculs dépendent d'une itération précédente...

```
1 parfor i = 1:5
2     res(i) = maFonction(i);
3 end
```

# 3. Conventions de programmation

Résumé de quelques conventions de codage préconisées pour le langage Matlab. On peut aussi consulter le document de Richard Johnson [1], dont ce document est en grande partie inspiré. Le chapitre sur la POO redonne quelques conventions/bonnes pratiques spécifiques.

## 3.1. Nommage

De manière générale, on évite les caractères spéciaux dans les noms, et on se limite aux caractères alphabétiques. Il est préférable d'employer des noms anglais. Cela permet (1) d'homogénéiser la langue du code et (2) d'éviter les problèmes dus aux accents... On évite les numéros sauf pour des fichiers temporaires ou de données.

Le seul caratère spécial toléré (à part les caractères spéciaux imposés par Matlab, tels « @ » ou « + ») est le caractère souligné « _ ». Il est à utiliser pour nommer les constantes statiques.

Attention au caractère « - », qui est interprété par Matlab comme une soustraction...

### 3.1.1. Noms de fichiers

Les fichiers Matlab peuvent contenir des scripts, des fonctions ou des classes. Ils ont tous l'extension « .m ». Il n'y a pas de limite sur le nombre de caractères.

De manière générale, il est préférable d'écrire les noms de fichier en minuscule. Pour les noms composés, on utilise la syntaxe qui consiste à coller les noms et à mettre en majuscule la première lettre de chaque mot. Exemple : nomDeLaFonction. Exception : les noms de classe, qui commencent par une majuscule.

### 3.1.2. Noms des fonctions

La convention Matlab est de stocker les fonction dans des fichiers qui portent leur nom. On applique donc aux fonctions les mêmes conventions que pour les noms de fichier.

Dans la mesure du possible, on utilise des noms de fonction qui commencent par des verbes. Cela permet de différencier les variables (noms) et les fonctions (verbes). Exemple :

```
1 distance = computeDistance(...)
```

On peut utiliser des préfixes pour des groupes de fonctions qui travaillent sur des types communs. Par exemple, des fonctions qui travaillent sur des données « polygones » peuvent avoir les noms : polygonArea, polygonCentroid... Autre exemple : un préfixe « im » pour les fonctions travaillant sur des images (conventions utilisée par la boîte à outils « Image Processing » de Mathworks).

1. http://www.datatool.com/downloads/matlab_style_guidelines.pdf

### 3.1.3. Noms des classes

Matlab permet la programmation orientée objet. Afin de différencier les classes et les fonction, j'utilise comme convention de nommer les classes en commençant par une majuscule. Les variables membres et les méthodes suivent la convention générale de nommage (« nomDeMembre », « nomDeMethode »...).

## 3.2. Codage

### 3.2.1. Commentaires

Penser à mettre des commentaires dans les codes... Matlab est un langage concis, il est donc très facile d'écrire des lignes courtes mais obscures. L'idée est de résumer l'intention. Une proportion de 30 à 50% de commentaire semble correcte.

On préfèrera les commentaires sur une ligne, commençant par le caractère « % », suivi d'une espace, et du commentaire commençant par une majuscule. Exemple :

```
1 % Exemple de commentaire
```

### 3.2.2. Ponctuation

On termine les instructions par un point-virgule, sans mettre d'espace avant. Cela permet (1) d'avoir une syntaxe similaire à d'autres langages comme C ou Java, et (2) de ne pas afficher les résultats intermédiaires.

On évite les point-virgule à la fin des instructions « end », cela surcharge la lecture. On en met à la fin des mots-clés « return », « continue », qui sont considérées comme des instructions à part entières.

### 3.2.3. Aérer le code

L'écriture du code doit être relativement aérée. Quelques principes généraux :

— une instruction par ligne si possible
— une espace avant et après le signe d'affectation "="
— on évite les parenthèses autant que possible (notamment pour les tests et les boucles)
— on essaie de faire tenir les lignes dans la limite de 80 caractères. L'éditeur de Matlab permet d'afficher la limite, et de passer à la ligne automatiquement.
— on préconise souvent de mettre des espaces autour des opérateurs mathématiques, à voir selon les cas...

### 3.2.4. Structuration du code

On essaie de faire des « blocs fonctionnels », composés de quelques lignes, qui s'occupent d'une tâche donnée, et que l'on commente.

Les structures de contrôle (boucles, tests...) sont indentées avec quatre caractères espace. On évite les tabulations, qui apparaissent différemment selon les éditeurs. L'idéal est de les

remplacer par quatre caractères espace. On évite d'écrire les boucles ou les tests sur une seule ligne.

Exemple :

```
% Initialisation des données
data = zeros(1, N);
for i = 1:N
    value = getInitValue(i);
    data(i) = initData(value);
end
```

### 3.2.5. Ecrire du code court

On évite les encapsulations d'indentation. Un niveau de 3 ou 4 indentations devrait être un maximum. Sinon, essayer de faire des sous-fonctions.

On évite les boucles ou les tests trop longs en terme de nombre de lignes.

## 3.3. Documentation des fonctions

Les commentaires en début des fichiers de fonctions peuvent être utilisés pour construire automatiquement la documentation. Il est donc préférable de les soigner...

La première ligne du fichier contient la déclaration de la fonction, suivie de l'en-tête, puis du code.

### 3.3.1. Ligne d'entête (« h1 »)

La première ligne de la documentation contient le nom de la fonction en majuscule, suivi d'une courte description (qui tient sur la ligne). Pour homogénéiser, il est préférable de l'écrire en anglais, en commençant par un verbe à l'infinitif, avec la première lettre en majuscule.

Exemple :

```
function res = demoFunction(x, n)
%DEMOFUNCTION Compute a sample function of x and n
...
```

### 3.3.2. Description de la fonction

Le reste de l'entête doit contenir l'explication de la syntaxe d'utilisation de la fonction, avec la description des paramètres.

Suggestion de syntaxe : écrire le nom des fonctions tel quel, écrire les paramètres en majuscule. L'idée est de favoriser le copier-coller après avoir tapé un 'help'. La description utilise des verbes conjugués.

Exemple :

```
function res = demoFunction(x, n)
%DEMOFUNCTION Compute a sample function of x and n
%
```

```
4  % RES = demoFunction(X, N)
5  % Computes the N-th power of the number X.
6  %
7  % RES = demoFunction(X, N, METHOD)
8  % Specifies the method to use. METHOD can be one of:
9  % 'normal': use the normal method
10 % 'optimized': use the optimized method
```

Un en-tête de fonction peut aussi contenir un section d'exemples, et une section « See also », qui fait référence à des fonctions voisines. Les sections couramment utilisées dans l'aide de Matlab sont :

- Syntax
- Description
- Examples
- See also
- Algorithm
- References

Il est possible d'inclure des liens hypertexte dans l'aide, et de faire exécuter du code Matlab quand on clique dessus.

```
1 disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

### 3.3.3. Liens entre fonctions

Les noms de fonctions renseignées après la balise « See also » génèrent automatiquement des liens hypertexte. C'est assez pratique pour naviguer entre des fonctions apparentées. Si un fichier « Contents.m » existe (voir ci-dessous), on peut aussi donner le nom du module, ce qui permet créer une ébauche de navigation.

Il est aussi possible d'inclure explicitement des liens hypertexte vers d'autres fonctions. Exemple :

```
1 % See the help for the <matlab:doc('publish') publish> function.
```

### 3.3.4. Cartouche

La fin de l'en-tête devrait contenir le nom de l'auteur, la date de création, et une mention de copyright. Un outil comme « tedit [2] » permet d'ajouter automatiquement un cartouche pré-défini à tout nouveau fichier. Exemple de cartouche :

```
1 % ------
2 % Author: Name and given name
3 % e-mail: my.name@centre.inra.fr
4 % Created: 2011-04-01,    using Matlab 7.9.0.529 (R2009b)
5 % Copyright 2011 INRA - My Laboratory
```

2. http://www.mathworks.com/matlabcentral/fileexchange/8532-tedit

## 3.4. Modules et bibliothèques

On peut regrouper des fonctions voisines dans des répertoires, et créer un paquetage ou module. Il suffit que le répertoire en question soit accessible via le path.

Si le répertoire du module est accessible depuis Matlab, un help avec le nom du module permet d'afficher la première ligne des en-tête des fonctions du module.

### 3.4.1. Fichier Contents.m

On peut aussi créer un fichier « Contents.m », qui est affiché si on appelle l'aide de Matlab sur le module. L'idée d'un tel fichier est de présenter un sommaire des fonctions du répertoire. Chaque fonction apparaît ainsi avec la première ligne de son en-tête.

Le fichier Contents peut être généré automatiquement par Matlab, via un « content report ». Il contient basiquement la liste des fonctions accompagnées de leur description courte. Il peut être retravaillé, pour grouper les fonctions similaires, et pour donner une description générale du module.

Quelques précautions à prendre :

— Le système d'aide de Matlab utilise le fichier « Contents.m », mais ne trouve pas le fichier « contents.m »...

— Matlab n'aime pas les liste à puces commençant par des tirets (« - »), il lui arrive de les faire disparaître (confusion avec les résumés de fonction ?). Recommandation : utiliser des astérisque (« * »).

### 3.4.2. Infos de version

Les deux premières lignes du fichier Contents.m sont analysées par Matlab pour fournir des informations de version (par la commande « ver »).

```
1 % Toolbox description
2 % Version xxx dd-mmm-yyyy .
```

La convention de Matlab est de donner uniquement la description de la bibliothèque. Je préfère ajouter le nom, ça permet de la retrouver a posteriori. Il faut aussi avoir un point à la fin de la deuxième ligne pour que la date soit prise en compte (2009b).

### 3.4.3. Numérotation des versions

Grande question... L'idée est de numéroter par majeur.mineur.correctif. Une alternative est d'utiliser systématiquement la date de release (format yyyy-mm-dd).

## 3.5. Compatibilité

Quelques points pour faciliter le passage entre différents systèmes d'exploitation :

— La gestion des noms de fichiers devrait se baser sur les fonction « fullfile » et fileparts », qui se chargent de gérer les séparateurs de fichier

— L'utilisation des '...' à la fin d'une ligne pourrait causer des problèmes de compatibilité pour les passages de Linux à Windows (à vérifier).

Deuxième partie .

# Programmation orientée objets

Pour regrouper ses fonctions utilitaires, une pratique courante dans certains langages de programmation est d'utiliser la programmation orientée objets (POO). On crée des objets ayant certains attributs, et on les modifie ou on fait des calculs avec en appelant leurs méthodes, qui sont des fonctions spécifiques à l'objet.

Matlab offre plusieurs mécanismes pour manipuler des objets. Les plus simples sont les structures, mais elles ne permettent pas de créer de méthodes spécifiques. Depuis la version 2006 ( ?), Matlab propose une gestion simplifiée des objets, qui permet de créer ses propres types de données, mais qui reste assez limitée. Enfin, depuis 2008, la gestion des objets dans un sens plus classique a été introduite, qui permet plus de possibilités et de rigueur, au prix d'un léger apprentissage.

Un premier chapitre présente les structures de données, ainsi que les objets par valeurs, qui corerspondent à l'ancienne version des objets. Un deuxième chapitre présente la création et l'utilisation des objets par référence (« handle-based ») . Le chapitre suivant présente l'héritage de classe, qui permet de construire des objets en réutilisant une partie du code. On termine ensuite avec quelques bonnes pratiques de programmation objet.

# 4. Structures de données et classes « par valeur »

Il existe plusieurs manière d'aborder la programmation orientée objets sous Matlab. Les structures de données permettent d'encapsuler et d'org des données hétérogènes. Il est ensuite possible de leur associer des fonctions, ce qui donne les objets dit « par valeur ». La gestion des objets par référence sera vue au chapitre suivant.

## 4.1. Structures de données

Les structures permettent d'agglomérer des données hétérogènes (nombres, tableaux, chaînes, autres structures...), et sont à la base du concept de classe.

### 4.1.1. Déclaration et utilisation

La déclaration d'une structure est plutôt simple :

```
1 personne.nom = 'Dupont';
2 personne.prenom = 'Thierry';
3 personne.age = 23;
```

On peut aussi assigner ou récupérer les champs de manière détournée, en plaçant le nom du champ dans une variable :

```
1 champ = 'nom';
2 nom = getfield(personne, champ);
```

Il s'agit d'une méthode simple pour passer des arguments à une fonction.
Il est aussi possible d'utiliser la syntaxe suivante :

```
1 nom = personne.('nom');
```

### 4.1.2. Tableaux de structures

On peut aussi accéder aux éléments d'une structure dans un tableau :

```
1 nom3 = personnes(3).nom;
```

Dans le cas d'un tableau de structures, on peut aussi récupérer l'ensemble des valeurs pour un champ donné, en utilisant une des syntaxes suivantes :

```
1 noms = {personnes.nom};
2 ages = [personnes.age];
```

Un inconvénient est que l'on perd le dimensionnement du tableau de départ : on obtient en sortie un vecteur colonne. Dans les cas où c'est nécessaire, on peut redimensionner avec l'instruction reshape :

```
ages = reshape([personnes.age], size(personnes));
```

### 4.1.3. Fonctions utiles

Matlab fournit quelques fonctions pratiques pour manipuler les structures :

**fieldnames(S)** liste les champs de la structure

**isfield(S, NAME)** pour tester si le champs NAME existe (NAME est une chaîne de caractères)

## 4.2. Objets « par valeur »

Première version des classes sous Matlab. Elles sont appelées 'value object' dans la doc Matlab. On manipule à tout instant une copie des objets. Les conséquences directes sont (1) une occupation mémoire plus grande et (2) la nécessité de renvoyer un objet modifié dès que l'on veut changer son état.

### 4.2.1. Déclaration

On commence par créer un répertoire '@MaClass', qui contient toutes les fonctions qui s'appliqueront à ce type d'objets.

On définit ensuite un constructeur, dont la tâche principale est d'affecter un nom de classe à l'objet. Ce constructeur a le même nom que le répertoire dans lequel il est placé, sans l'arobase.

```
function obj = MaClass(data)
%MACLASS cree un objet de type MaClass
obj.data = data;
obj = class(obj, 'MaClass');
```

Un objet se crée ensuite par un appel à la fonction 'MaClasse'

```
> obj = MaClass('maDonnee');
```

### 4.2.2. Manipulation

Pour agir sur l'objet, on ajoute des fonctions dans le répertoire de la classe. Si la fonction est appelée, c'est qu'au moins un des arguments est un objet de la classe considérée. On peut ainsi récupérer les valeurs des champs de l'objets, et renvoyer ou afficher un résultat.

```
function showData(obj)
%SHOWDATA affiche les donnees de l'objet
disp('Donnees de l objet :');
disp(obj.data);
```

### 4.2.3. Modification

Pour modifier un objet, il faut en créer un nouveau et le renvoyer :

```
1 function obj = modifier(obj, newData)
2 %MACLASS cree un objet de type MaClass
3 obj.data = newData;
```

On modifie un objet de la façon suivante :

```
1 > obj = modifier(obj, 'Nouvelle Donnee');
2 > showData(obj)
3     Donnees de l'objet :
4     Nouvelle Donnee
```

L'inconvénient de cette approche est que pour des objets volumineux (un polygone, une image, un maillage...), la totalité de l'objet est passée en argument, ce qui conduit à une utilisation mémoire intensive.

# 5. Programmation orientée objets

Ce chapitre présente la gestion des objets introduite depuis la version 2008b. Cette version permet notamment de gérer des objets par référence, c'est à dire qui peuvent être modifiés et qui gardent trace de leur modification (contrairement aux objets pre-2008). On commence par présenter la déclaration des objets, puis des différents types de méthodes (les fonctions assoxiées aux objets).

## 5.1. Déclaration de classe

Les classes peuvent être définies soit entièrement dans un fichier, soit dans un répertoire propre. Dans le cas d'un répertoire, le nom du répertoire commence par un arobase (caractère '@'), et le reste correspond au nom de la classe.

### 5.1.1. Déclaration

Pour chaque classe, on créé au minimum un fichier 'NomDeLaClasse.m'. Ce fichier contient la déclaration des champs, des constructeurs, et éventuellement d'autres méthodes. La définition d'une classe se fait dans un bloc « classdef », qui comprend tout le fichier :

```
1 classdef NomDeLaClasse < NomClasseParente
2 ... % déclaration de la classe
3 end
```

Le signe « < » sert à indiquer la ou les classes parentes. Pour manipuler des objets par référence, il faut que la classe hérite de la classe « handle », ou d'une classe qui hérite elle-même de handle.

### 5.1.2. Déclaration des variables membre

Dans la suite du fichier, on déclare les variables locales contenues dans chaque objet. On peut spécifier des valeurs par défaut.

```
1 ...
2 % Déclare les champs à la suite
3 properties
4   x = 0;
5   y = 0;
6 end % fin decl champs
7 ...
```

### 5.1.3. Déclaration des méthodes

Une méthode se déclare dans le corps de la déclaration de la classe. On commence en général par le ou les constructeurs, qui créent une nouvelle instance de la classe à partir des arguments passés en paramètre.

Exemple de déclaration complète :

```
classdef Point2D < handle
...
% Déclare les differentes methodes
methods
  % Déclare le constructeur
  function this = Point2D(x, y)
    this.x = 0;
    this.y = 0;
  end % fin decl. constructeur

  % declare une methode de la classe
  % (ajouter les corodonnees de deux points)
  function res = addCoords(this)
    res = this.x + this.y;
  end % fin decl. de la methode 'addCoords';

  % declare une autre methode
  % (calcul de la distance entre deux points)
  function dist = distance(this, that)
    dist = hypot(this.x-that.x, this.y-that.y);
  end

end % fin decl methodes
end % fin decl classe
```

Il est aussi possible d'écrire chaque méthode dans un fichier séparé. Cela montre mieux quelles sont les méthodes implémentées ou surchargées, et évite d'avoir un fichier de classe trop gros.

## 5.2. Création et utilisation des objets

Pour construire un objet (c'est à dire instancier une classe), on passe par un appel au constructeur :

```
> p1 = Point2D(2, 3);
> p2 = Point2D(5, 7);
```

Pour appeler une méthode, plusieurs syntaxes sont possibles :

```
> res1 = p1.addCoords;
> res2 = p1.addCoords();
> res3 = addCoords(p1);
```

La deuxième syntaxe est préférable. La première introduit une confusion entre les noms de champs et de méthode, tandis que la troisième risque d'introduire des conflits de noms. (Au niveau technique, les deux premières ajoutent aussi un appel à la fonction subsref, à voir comment cela se traduit au niveau des performances).

Pour appeler des méthodes sur plusieurs objets :

```
> d12 = p1.distance(p2)
  d12 =
      5
```

Ici, la méthode est appelée sur l'objet « p1 », avec comme deuxième argument l'objet « p2 ».

## 5.3. Méthodes privées

Les méthodes privées ne sont accessibles que depuis les méthodes de la classe. Elle permettent de partager du code entre les différentes méthodes, sans que ce code ne soit rendu public. Cela concerne typiquement du code d'initialisation, ou du code de vérification de l'intégrité des données de la classe.

### 5.3.1. Méthodes privées dans la déclaration de classe

Si toutes les méthodes sont déclarées dans un même fichier, les fonctions privées doivent être déclarées dans un bloc séparé. On crée un bloc de déclarations de méthodes, pour lequel on spécifie l'accès privé « private », et on place le code des fonctions privées :

```
1 classdef ClassName
2 % dans le bloc des méthodes, on écrit le patron de la méthode
3 methods (Access = private)
4     function res = maFonction(this, arg1, arg2)
5     % ... code de la fonction privée
6     end
7 end
8 end
```

### 5.3.2. Méthodes privées dans un fichier séparé

Si on a déclaré la fonction dans un répertoire spécifique (« @-répertoire »), on peut écrire le code dans un fichier séparé. Il faut quand même déclarer la fonction le bloc de déclaration des méthodes de la classe, afin de pouvoir la rendre privée.

```
1 classdef ClassName
2 % dans le bloc des méthodes, on écrit le patron de la méthode
3 methods (Access = private)
4     res = maFonction(this, arg1, arg2)
5 end
6 end
```

On écrit ensuite le code de la méthode dans un fichier appelé « maFonction.m », qui se trouve dans le répertoire de la classe.

```
1 % fichier "maFonction.m"
2 function res = maFonction(this, arg1, arg2)
3    % ici, le code de la fonction privée
4 end
```

### 5.3.3. Méthodes locales

Quand on définit une classe dans un répertoire, on peut créer un répertoire « private ». Les fonctions présentes dans ce répertoire sont utilisables depuis les méthodes de la classe, mais ne sont pas visible de l'extérieur. Elles se comportent comme des méthodes privées, à la différence que le premier argument ne doit pas être la référence de l'objet.

## 5.4. Méthodes statiques

Il est possible de déclarer des méthodes statiques, qui seront partagées par toutes les instances de la classe. Ces méthodes ne peuvent manipuler les données membres de la classe. Leur intérêt est soit de regrouper des fonctions opérant sur des objets d'une classe donnée, soit de définir des constantes ou des méthodes spécifiques à la classe d'objets utilisée.

Les méthodes statiques se déclarent de la même manière que des méthodes classiques, en ajoutant le mot-clé « Static » (avec la majuscule). Pour utiliser une méthode statique, on utilise le nom de la classe, suivi du nom de la méthode

```
res = NomDeClasse.nomDeMethode(...);
```

### 5.4.1. Exemple 1 : calcul de $\pi$ avec une précision arbitraire

On commence par définir une classe « Math » contenant une méthode « pi », qui accepte un argument de tolérance numérique en entrée.

```
classdef Math
methods (Static)
  % calcule PI avec une precision donnee
  function p = pi(tol)
    [n, d] = rat(pi, tol);
    p = n/d;
  end
end
end
```

On calcule ensuite selon différentes précisions :

```
pi1 = Math.pi(.1)
pi1 =
    3.1429
pi2 = Math.pi(.001)
pi2 =
    3.1416
```

### 5.4.2. Exemple 2 : calcul de la distance entre deux points

écriture de la classe :

```
classdef Point2D
...
methods(Static)
  % Calcule la distance entre deux points
  function d = distancePoints(p1, p2)
    % calcule la distance par Pythagore
    d = hypot(p1.x-p2.x, p1.y-p2.y);
  end % fin de la methode statique
end
```

Exemple d'appel :

```
p1 = Point2D(2, 3);
p2 = Point2D(5, 7):
d = Point2D.distancePoints(p1, p2)
```

```
4 d =
5      5.0
```

### 5.4.3. Limitations

On ne peut pas se passer du nom de la classe pour appeler une classe statique. Cela implique des lignes de code un peu longues, et peut être pénible pour des hiérarchies de classe un peu touffues.

On peut redéfinir une classe statique dans une classe dérivée, à condition que la classe ne soit pas « scellée » (équivalent du mot-clé « final » en java, qui se traduit en Matlab par « Sealed = true »).

On ne peut pas avoir de champs de classe statiques. Cela peut être pénible pour certaines classes (typiquement une classe singleton). On peut s'en sortir en utilisant des variables persistantes dans le corps d'une méthode statique.

## 5.5. Autres méthodes particulières

Certaines méthodes spécifiques sont appelées dans certains cas particuliers : set, loadobj, saveobj... Un « help handle » permet de les lister, afin éventuellement de ré-implémenter celles qui le nécessitent.

# 6. Héritage de classe

Utiliser l'héritage permet de gagner en modularité, et parfois en clarté. Ce chapitre présente la gestion de l'héritage de classes sous Matlab, et se consacre ensuite à quelques techniques particulières que l'on peut utilsier pour l'héritage.

## 6.1. Créer et utiliser une hiérarchie de classes

### 6.1.1. Déclaration de l'héritage

On utilise le signe « < », suivi du nom de la classe parente :

```
1 classdef ClasseFille < ClasseParente
```

Si on déclare un objet de la classe ClasseFille, alors elle bénéficiera de toutes les fonctionalités déjà implémentées par la classe parente.

Matlab supporte l'héritage multiple, il faut utiliser le caractère « & » :

```
1 classdef ClasseFille < ClasseMere & ClassePere
```

Par contre, il faut prendre quelques précautions pour la gestion des conflits (par exemple, si les deux classes parentes implémentent une même méthode, laquelle faut-il appeler...).

### 6.1.2. Appeler le constructeur parent

Exemple de constructeur appelant le constructeur de la classe parente :

```
1 methods
2     function this = ClasseFille(varargin)
3         % calls the parent constructor
4         this = this@ClasseMere(varargin{:});
5     end % constructor
6 end % methods
```

Cet exemple est minimaliste, l'idée est de pouvoir rajouter un traitement spécifique des entrées avant ou après l'appel au constructeur. Pour l'héritage multiple, on appelle les constructeurs parents successivement.

```
1 methods
2     function this = ClasseFille(varargin)
3         % calls the parent constructor
4         this = this@ClasseMere(); % sans argument
5         this = this@ClassePere(varargin{1}); % avec argument
6     end % constructor
7 end % methods
```

Pour une classe parente située dans un package, il faut ajouter le nom du package. Ex :

```
this = this@lePackage.ClasseMere(varargin{:});
```

Il faudrait vérifier si le constructeur vide de la classe parente est appelé par défaut ? A priori oui...

### 6.1.3. Appeler la méthode parente

L'appel à une méthode de la classe parente suit la même logique que pour le constructeur :

```
classdef ClasseFille < ClasseMere
methods
    function uneMethode(obj)
        % ... pretraitements
        uneMethode@ClasseMere(obj); % Appelle la methode de la classe parente
        % ... post-traitements
    end
end
end
```

## 6.2. Afficher les objets

Pour gérer la manière dont les objets peuvent être affichés, on peut surcharger les méthodes disp et display :

**disp** affiche un résumé de l'objet, s'utilise de manière explicite (en tapant « disp obj » ou « disp(obj) »)

**display** fonction d'affichage, qui fait le plus souvent appel à disp, et qui est appelée quand le nom de l'objet termine une ligne de commande non terminée par un point-virgule.

**char** le comportement par défaut de cette méthode est de convertir un objet en un tableau de caractères. Cela peut être utilisé par les fonctions d'affichage.

Si on tape une ligne de commande contenant uniquement le nom d'un objet, on a ainsi une chaîne d'appel : display->disp->char

## 6.3. Surcharger les opérateurs mathématiques

Quand une classe a pour vocation de représenter des données calculables (par exemple, un vecteur mathématique, un quaternion...), on peut pouvoir utiliser des opérations arithmétiques classiques de manière naturelle. Cela est possible en surchargeant certaines fonctions.

Les fonctions mathématiques classiques (additions, multiplication...) peuvent être surchargées en suivant le tableau 6.1. Les fonctions se présentent souvent sous deux formes, l'une opérant sur le tableau global (nom de fonction préfixé par un « m »), l'autre sur les éléments du tableau (pour les opérateur préfixés par un point).

Il est aussi possible de surcharger des fonctions plus complexes, comme le logarithme ou l'exponentielle (voir le tableau 6.2). On peut aussi surcharger les fonctions trigonométriques, qui sont relativement nombreuses...

La manipulation des tableaux binaires se fait grâce aux fonctions données dans les tableaux 6.3 et 6.4.

| opérateur | fonction matlab | note |
|---|---|---|
| a .\ b | ldivide.m | |
| a - b | minus.m | |
| a \ b | mldivide.m | |
| a ^ b | mpower.m | |
| a / b | mrdivide.m | |
| a * b | mtimes.m | |
| a + b | plus.m | |
| a .^ b | power.m | |
| a ./ b | rdivide.m | |
| a .* b | times.m | |
| -a | uminus.m | opérateur unaire |
| +a | uplus.m | opérateur unaire |

TABLE 6.1. – Surcharge des opérateurs arithmétiques.

| fonction matlab | note |
|---|---|
| sqrt | |
| nthroot | |
| exp | |
| log | logarithme naturel |
| log2 | logarithme binaire |
| log10 | logarithme décimal |

TABLE 6.2. – Surcharge de quelques fonctions mathématiques classiques.

## 6.4. Surcharger subsref et subsasgn

Il est possible sous Matlab de modifier et ajuster le comportement des opérateurs d'indexation comme les parenthèses et le point. Cela peut améliorer la facilité d'utilisation, mais cela requiert un peu de technicité.

Les quelques méthodes à modifier sont les suivantes :

**subsref** permet de spécifier le retour des appels du type monObjet(3, 3) ou monObjet.truc

**subsasgn** permet de spécifier le retour des appels du type monObjet(3, 3)=2 ou monObjet.truc = 'demo'.

**end** permet de changer la manière dont les indices sont calculés dans un tableau (ex : monObjet( :, 4 :end))

**subsindex** permet d'utiliser un objet comme un index. Ex : res = obj1(obj2).

**numel** renvoie le nombre d'éléments dans un tableau.

**size** pour avoir la taille de l'objet ou du tableau d'objets

**length** renvoie la taille selon la plus grande dimension

| opérateur | fonction matlab | note |
|---|---|---|
| a == b | eq.m | |
| a >= b | ge.m | |
| a > b | gt.m | |
| a <= b | le.m | |
| a < b | lt.m | |
| a ~= b | ne.m | |

TABLE 6.3. – Surcharge des opérateurs de comparaison. Ils fournissent un résultat binaire.

| opérateur | fonction matlab | note |
|---|---|---|
| a & b | and.m | |
| ~a | not.m | opérateur unaire |
| a \| b | or.m | |
| | xor.m | pas de symbole |

TABLE 6.4. – Surcharge des opérateurs logiques. Ils travaillent sur des arguments binaires.

### 6.4.1. Modifier la lecture des données

Quelques idées de ce qu'il serait bien d'avoir pour des tableaux d'objets. On considère une classe « Point » avec deux champs x et y, et un tableau de points de 2 lignes et 3 colonnes.

```
1 obj1 = Point(10, 1);
2 obj2 = Point(20, 1);
3 obj3 = Point(30, 1);
4 obj4 = Point(10, 2);
5 obj5 = Point(20, 2);
6 obj6 = Point(30, 2);
7 array = [obj1 obj2 obj3 ; obj4 obj5 obj6];
```

**array(1, 1)** renvoie obj1

**array(2 :3, 1 :2)** renvoie un sous-tableau de même classe

**obj1.x** renvoie 10

**array(1, 1).x** renvoie 10

**array.x** renvoie un tableau [10 20 30 ;10 20 30]

### 6.4.2. Modifier les données

à utiliser :

```
1 array(2, 3) = p2;              % allocation simple
2 array(2, 3).x = 4;             % allocation de donnees
3 array(23, 12) = p24;           % copie d un point
```

### 6.4.3. Utiliser subsindex

Cette méthode permet d'utiliser un objet comme un index. Ex :

```
1 res = obj1(obj2);
```

La méthode accepte un seul argument : l'objet servant d'index. Elle renvoie un ensemble d'indices compris entre 0 et numel(obj)-1.

## 6.5. Surcharge des opérations de tableau

La plupart de ces fonctions concernent principalement les tableaux définis par deux coordonnées. Les fonctions qui se traduisent par des surcharges d'opérateurs sont listées dans le tableau 6.5.

Méthodes de concaténation :

**vertcat** concaténation verticale

**horzcat** concaténation horizontale

**cat** concaténation selon une dimension arbitraire

**repmat** réplique un tableau pour faire une mosaïque

**kron** une autre manière de répéter des tableaux

| opérateur | fonction matlab | note |
|---|---|---|
| a' | ctranspose.m | |
| [a b] | horzcat.m | |
| a.' | transpose.m | |
| [a ; b] | vertcat.m | |
| a :b | colon.m | |

TABLE 6.5. – Surcharge des opérateurs manipulant la forme des tableaux des objets

Si cela a un sens pour les objets manipulés, on a aussi des fonctions permettant de changer la taille des objets

**permute** permute les dimensions du tableau

**reshape** ré-organise les éléments du tableau pour avoir une nouvelle taille de tableau

**flipdim** retourne selon une direction spécifique

**fliplr** retourne de droite à gauche

**flipud** retourne de haut en bas

**rot90** rotation par 90 degrés

**transpose** retournement par rapport à la diagonale

**ctranspose** le même...

**sort** renvoie une version triée de l'objet

## 6.6. Méthodes et classes abstraites

### 6.6.1. Méthode abstraite

On peut déclarer des méthodes comme abstraites. Les méthodes ne sont pas implémentées par la classe, mais seront implémentées par les sous-classes. L'intérêt est de manipuler les différentes classes concrètes à travers les méthodes déclarées dans les classes parentes, sans se soucier des détails d'implémentation.

```
1 classdef Group
2 % La classe Group déclare deux méthodes 'add' et 'times'
3 % Une classe héritant de Group devra les implémenter toutes
4 % les deux avant de pouvoir être instanciée
5 methods (Abstract)
6   result = add(g1, g2)
7   result = times(group, k)
8 end
9 end
```

### 6.6.2. Classe abstraite et interface

Une classe est dite **abstraite** si elle comporte des méthodes abstraites. Une classe abstraite ne peut pas être instanciée (on ne peut pas créer d'objet de cette classe).

Matlab ne permet apparemment pas de déclarer comme abstraite une classe qui ne contient pas de méthode abstraite.

Si une classe ne contient que des classes abstraites, on l'appelle une **interface**.

### 6.6.3. Méthode scellée

Il est possible de déclarer une méthode comme scellée (*sealed*). Une telle méthode ne pourra pas être redéfinie par une classe dérivée.

```
1 classdef sealedDemo
2 methods (Sealed)
3   function res = sealedMethod(this, value)
4     res = this.value + value;
5   end
6 end
7 end
```

# 7. Bonnes pratiques en programmation orientée objet

J'essaie de noter ici ce que je considère comme des bonnes pratiques de programmation objet sous Matlab.

## 7.1. Conventions de nommage

J'utilise globalement les conventions de nommage java.

**classes** la première lettre en majuscule. Exemple : « NomDeClasse »

**méthodes** la première lettre en minuscule. Exemple : « nomDeMethode »

**membres** pas de préfixe, mais j'évite d'utiliser des noms de fonctions matlab pour éviter les conflits. Dans le corps des fonctions, je référence systématiquement les champs par « this.nomDuChamp ».

## 7.2. Construction des objets

### 7.2.1. Constructeurs

Il faut toujours prévoir un constructeur vide : dans le cas où on déclare un tableau d'objets, le constructeur vide est appelé automatiquement.

De plus, il est préférable d'avoir systématiquement un constructeur de copie, qui équivaut à cloner les variables membres.

### 7.2.2. Constructeurs statiques

Les constructeurs statiques offrent une alternative intéressante à l'appel direct des constructeurs. Ils consistent à déclarer dans la classe une ou plusieurs méthodes statiques qui retournent une instance de la classe.

On leur trouve plusieurs avantages :

— il est possible de donner un nom plus explicite pour la méthode (par exemple, une classe Point peut proposer les méthodes createCartesian ou createPolar, selon l'interprétation que l'on veut faire des arguments)

— on peut renvoyer une instance d'une classe dérivée à la place de la classe appelante. Cela permet de renvoyer un objet bien adapté aux arguments d'entrée, et facilite les évolutions ultérieures des objets.

En particulier sous Matlab, il est même possible de déclarer des classes totalement abstraites (des interfaces), mais qui proposent des fabriques statiques. Les fabriques statiques renvoient (en général) une instance d'une implémentation particulière de l'interface.

## 7.3. Gérer des tableaux d'objets

### 7.3.1. Créations d'un tableaux d'objets

Lorsque l'on itère sur plusieurs objets, il peut être préférable de pré-allouer la mémoire afin d'éviter les redimensionnements de tableau.

```
dim = num2cell(size(input));          % recupere la memoire
result(dim{:}) = NomDeLaClasse();     % alloue un tableau, appelle le constructeur vide
for i=1:length(input(:))              % boucle pour créer chaque objet
  % traitement sur chaque objet
  result(i) = NomDeLaClasse(input(i));
end
```

Le fait de passer par un tableau de cellules pour calculer la taille permet de gérer les cas de tableaux en dimension quelconque. Remarque : il ne faut pas oublier de prévoir un constructeur vide, qui sera appelé automatiquement quand le tableau sera créé.

### 7.3.2. Appels de méthodes sur un tableau d'objets

Il est assez agréable d'avoir des fonctions « vectorisées », qui calculent le résultat sur tous les éléments d'un tableau sans devoir passer par une boucle. De plus, l'idéal est d'avoir en sortie un tableau de même taille que le tableau d'entrée.

Par exemple, on souhaite calculer la norme de plusieurs vecteurs stockés dans un tableau appelé « vectors ». On souhaite en sortie un tableau de double. Le contenu du fichier « getNorm », stocké dans le répertoire « Vector2D » serait le suivant :

```
function res = getNorm(this)
%GETNORM Renvoie la norme d'un vecteur
dim = num2cell(size(this));      % recupere la memoire du tableau
res = zeros(dim{:});       % cree un tableau vide
for i=1:length(this(:))          % calcule la norme de chaque vecteur
  res(i) = hypot(this(i).x, this(i).y);
end
```

Une alternative (ici plus concise) est d'utiliser la fonction « reshape » :

```
res = reshape(hypot([this.x], [this.y]), size(this));
```

Dans les cas où le résultat de la fonction appelée sur un objet unique est un tableau, on ne peut plus utiliser cette stratégie. Une possibilité est de renvoyer soit le tableau dans le cas où l'entrée est un objet unique, soit un tableau de cellules si l'entrée est un tableau d'objets.

## 7.4. Paquetages

Depuis la version 2008a, Matlab permet aussi l'utilisation de modules « importables », à la Java par exemple. Le nom du répertoire doit commencer par le caractère « + ».

```
+mypack
+mypack/pkfonc.m % une fonction de paquetage
+mypack/@pkClass % une classe dans un paquetage
```

L'utilisation des classes du modules se fait soit via une directive d'importation, soit en utilisant une syntaxe du type « nomDuModule.NomDeClasse ».

Exemple :

```
res = mypack.pkfunc(data)
```

est équivalent à :

```
import mypack. ;
res = pkfunc(data)
```

### 7.4.1. Encapsulation des classes

L'utilisation des paquetages permet de faciliter la manipulation des espaces de nom. On peut aussi placer dans des sous-modules les classes utilitaires, qui ne seront ainsi pas visibles par l'utilisateur sans être explicitement importées.

#### 7.4.1.1. Exemple 1

Conidérons un package '+Truc', qui contient :

— un objet Truc

— une fonction create

La fonction create peut être appelée de la manière suivante :

```
Truc.create;
```

et renvoie un objet de classe Truc.

#### 7.4.1.2. Exemple 2

Autre possibilité est ce créer une classe Graph, et de placer dans un répertoire utilitaire '+GraphImpl' les classes des objets 'VertexHandle', 'EdgeHandle'... La classe principale Graph importe à façon les classes nécessaires, mais l'utilisateur final n'a pas à se préoccuper du contenu de '+GraphImpl', et ne le voit même pas dans le chemin courant.

# Troisième partie .

# Concepts avancés

# 8. Interfaces graphiques

Matlab permet de développer des interfaces graphiques relativement conviviales, ce qui permet de faciliter la diffusion des outils. Par contre le développement est un peu plus techniques qu'avec certains autres langages.

## 8.1. Concevoir ses interfaces

On peut développer ses interfaces à la main, c'est-à-dire en positionnant chaque composant individuellement, ou bien s'aider d'outils exterieurs, tels que Guide, ou des gestionnaires de mise en page (layout managers).

### 8.1.1. Conception manuelle

Une méthode pour concevoir ses interfaces graphiques est de les programmer soi-même à la main. Long et vite fastidieux, mais on sait ce que l'on fait.

### 8.1.2. Utilitaire « Guide »

L'outil Guide est un environnement de conception d'interfaces utilisateur graphiques. Il permet de générer très rapidement de petites interfaces graphiques. Par contre le code généré devient très vite difficile à maintenir.

### 8.1.3. Utilisation de « layouts »

Possibilité de gérer des layouts « à la java ». Le gros avantage est que cela permet de mixer des composants redimensionnables et des composants de taille fixe, ce qui donne un aspect plus fini aux interfaces.

## 8.2. Les composants

### 8.2.1. Composants de fenêtre

On dipose de plusieurs types d'objets graphiques :

**figure** c'est le conteneur principal, qui va encapsuler les autres objets graphiques

**uicontrol** composant de base pour la plupart des widgets : boutons, boîtes à cocher, boutons radio, boîtes de texte, labels, sliders, listes...

**axes** objet graphique qui contient les graphes et / ou les images

**uipanel** permet d'encapsuler des widgets, et d'ajouter un peu de décorations

**uibuttongroup** utilitaire qui permet de grouper les boutons radios pour n'en avoir qu'un seul de sélectionné

**uitable** permet d'afficher un tableau de données.

On peut aussi trouver sur internet la manière d'afficher une arborescence (composant **uitree**).

Quelques attributs de composants :

**TooltipString** petite bulle d'aide qui s'affiche lorsque le curseur de la souris passe par dessus le composant

### 8.2.2. Menus

On peut aussi créer ses menus et barres d'outils.

**uimenu** pour générer la barre de menus d'une fenêtre

**uicontextmenu** il s'agit d'un menu « flottant », typiquement après un clic droit

### 8.2.3. Barre d'outils

Les fenêtres ont par défaut une barre d'outils prédéfinie. On peut la supprimer, ou définir la sienne.

**uitoolbar** permet de créer sa propre barre d'outils

**uipushtool** ajouter un bouton dans une barre d'outils

**uitoggletool** ajoute un bouton à deux états dans une barre d'outils

### 8.2.4. Boîtes de dialogue pré-définies

On dispose d'un grand nombre de boîtes de dialogue prêtes à l'emploi :
— message d'erreur, d'avertissement ou d'information
— lire ou sauver un fichier
— fenêtre affichant un menu

## 8.3. Évènements graphiques (callbacks)

Les callback sont les portions de code qui seront exécutées lors de l'activation de l'objet (appui sur le bouton, sélection du menu...). Il s'agit le plus souvent d'un pointeur de fonction.

### 8.3.1. Callback dans un objet

En utilisant la programmation orientée objet, on peut définir le callback sur une méthode de l'objet. C'est assez pratique, car cela permet de stocker les données dans la classe, et d'éviter de jongler entre les UserData des différents objets graphiques.

### 8.3.2. Partager une fonction callback entre plusieurs objets

On peut se trouver dans le cas où plusieurs objets graphiques ont des actions très similaires. Dans ce cas on peut vouloir partager la fonction « callback » entre les menus. Pour différencier les menus, on peut spécifier des 'UserData' différents. Exemple de déclaration de deux menus :

```
uimenu(menuTransform, ...
    'Label', 'Rotate Y Left', ...
    'UserData', [2 1], ...
    'Callback', @this.onRotateImage);
uimenu(menuTransform, ...
    'Label', 'Rotate Y Right', ...
    'UserData', [2 -1], ...
    'Callback', @this.onRotateImage);
```

Dans le corps de la classe, on définti la fonction de callBack « onRotateImage » :

```
function onRotateImage(this, hObject, eventdata)
    data = get(hObject, 'UserData');
    this.rotateImage(data(1), data(2));
end
```

## 8.4. Évènements fenêtre

Quelques pointeurs de fonctions que l'on peut renseigner pour les objets de type fenêtre :

**CreateFcn** appelé quand le fenêtre s'ouvre. Vide par défaut. Si on l'initialise après que la fenêtre est créée, cela n'a pas d'effet.

**DeleteFcn** appelé quand la fenêtre se ferme. Vide par défaut.

**CloseRequestFcn** appelé quand on demande à fermer la fenêtre. Par défaut, il s'agit d'une fonction « closereq ». Possibilité de surcharger, par exemple pour demander confirmation de la fermeture, vérifier que le document est sauvé...

**ResizeFcn** appelé quand on redimensionne la fenêtre à la souris. Vide par défaut.

On trouve aussi des fonctions pour la gestion de la souris et du clavier.

## 8.5. Gestion de la souris

### 8.5.1. Clic souris

On intercepte les clics de la souris en renseignant l'attribut « ButtonDownFcn » de l'objet graphique considéré. On applique pour cela le principe des pointeurs de fonction (function handle). On utilise en général un objet de type axis ou image.

**ButtonDownFcn** (objet graphique) appelée quand le bouton est enfoncé

Attention : quand on utilise ButtonDownFcn sur un objet de type axes, l'utilisation de fonctions telles que plot peut re-initialiser le callback. Il vaut donc mieux spécifier cette propriété sur un objet autre que axes (button, image...), qui lui ne sera pas affecté.

Pour savoir si le clic s'est fait avec le bouton droit, ou si les touches CTRL ou SHIFT ont été utilisées, il faut passer par la propriété « SelectionType » de la fenêtre courante. Les valeurs de cette propriété sont :

**Normal** un clic gauche

**Extend** clic gauche avec la touche shift enfoncée

**Alternate** un clic gauche avec la touche control, ou un clic droit

**Open** les deux boutons en même temps

Exemple de l'aide de Matlab :

```
fh_cb = @newfig; % Create function handle for newfig function
figure('ButtonDownFcn', fh_cb);

function newfig(src,evnt)
   if strcmp(get(src,'SelectionType'),'alt')
      figure('ButtonDownFcn',fh_cb)
   else
      disp('Use control-click to create a new figure')
   end
end
```

### 8.5.2. Déplacement souris

Pour la gestion du mouvement, on passe par l'attribut « WindowButtonMotionFcn ». Par contre cet attribut n'est disponible que pour des objets de type « figure ». Du coup il faut stocker quelque part la référence de l'objet courant.

La fin de trainée de la souris est interceptée par la fonction « WindowButtonUpFcn », attribut de l'objet figure.

Attention : les fonction de type « drawnow » semblent causer un appel supplémentaire aux fonctions de type « WindowButtonXXXFcn ». Il faut donc en tenir compte lors de l'écriture du corps des fonctions.

**WindowButtonDownFcn** (figure) appelée quand le bouton est enfoncé

**WindowButtonUpFcn** (figure) appelée quand le bouton est relaché

**WindowButtonMotionFcn** (figure) appelé quand la souris est déplacée

### 8.5.3. Position du curseur

La position du curseur est obtenue en récupérant la valeur de l'attribut « CurrentPoint » de l'objet axes considéré. Il s'agit d'un tableau de 2 lignes et 3 colonnes, qui correspond aux coordonnées des deux points 3D du début et de la fin du fenêtrage de la droite d'interception. C'est pratique pour les graphes 3D. Si le graphe est 2D, on n'utilise que les valeurs « point(1,1 :2) ».

### 8.5.4. Molette de la souris

Il est possible de gérer les évènements associés à la molette de défilement de la souris. Il faut utiliser l'attribut « WindowScrollWheelFcn ». Là aussi, ce n'est disponible que pour les

objets de type « figure ». Pour récupérer la direction du scroll (vers le haut ou vers le bas) on utilise la propriété VerticalScrollCount de l'argument eventdata, qui vaut -1 ou +1.

Exemple :

```
function mouseWheelScrolled(hObject, eventData)
    if eventData.VerticalScrollCount == 1
        disp('Wheel up');
    else
        disp('Wheel down');
    end
end
```

Et on ajoute le pointeur de fonction à la fenêtre courante par :

```
set(gcf, 'WindowScrollWheelFcn', @mouseWheelScrolled);
```

# 9. Exceptions et gestion des erreurs

Les mécanismes à base d'exceptions permettent une certaine souplesse dans le traitement des erreurs pouvant survenir lors de l'exécution d'un programme.

## 9.1. Principe des exceptions

Quand une erreur survient dans un programme, le comportement par défaut est d'arrêter le fonctionnement du programme, et d'afficher l'erreur. Exemple :

```
>> surf
??? Error using ==> surf at 50
Not enough input arguments.
```

Il est possible d'intercepter les erreurs, ce qui permet de L'idée est d'encapsuler les traitements générants des erreurs dans des blocs « try ».

Quand une erreur survient lors de l'éxecution d'un programme ou d'une fonction, une exception est levée. Le traitement de la fonction s'interrompt, et l'erreur est prise en charge par un bloc « catch ». Si aucun bloc catch n'est trouvé, un message d'erreur est affiché avec la liste des fonctions appelées pour produire l'erreur.

Exemple

```
try
    % bloc d'instructions pouvant poser des problèmes
    fid = fopen(filename, 'r');
    d_in = fread(fid);
catch ME
    % bloc d'interception de l'exception
    fprintf('Unable to access file %s\n', filename);
    return;
end
% on reprend le programme principal
```

L'objet ME est une instance d'un objet MException, qui contient quelques informations pour faciliter le traitement et le debuggage.

## 9.2. Traiter une exception

Les informations sur le problème sont encapsulées dans un objet de type MException. Cet objet contient les champs (publics) suivants :

**identifier** un identifiant d'erreur, de la forme « matlab :lib :function »

**message** le message expliquant le type d'erreur

**stack** la liste des fonctions appelées avant que l'erreur ne se produise

**cause** une liste d'exceptions, ce qui permet éventuellement d'encapsuler les traitements d'erreur

## 9.3. Créer ses erreurs

C'est en fait assez simple, il suffit d'utiliser la fonction « error ».

Matlab préconise d'utiliser un système d'identification d'erreurs. Cela permet en théorie de mieux aiguiller le traitement des erreurs. Syntaxe du type : « library :function :ErrorType ».

# 10. Gestion des évènements

La gestion des évènements permet une grande souplesse dans la programmation. L'idée est qu'à un certain moment de son déroulement, le programme puisse envoyer un signal qui sera intercepté par des objets qui « écoutent » les évènements. Les écouteurs vont traiter l'évènement, puis le programme principal reprendra la main.

On a ainsi trois catégories d'entités :

— les objets qui envoient des évènements. Typiquement, ce sont des instances de classes assez grosses, qui stockent une liste d'instances d'écouteurs, et qui diffusent les évènements quand il y a lieu

— les évènement eux-mêmes. En Java, ce sont des objets minimalistes, avec une référence à l'objet appelant et éventuellement quelques données d'état.

— les écouteurs (listeners), qui traitent les évènements. Sous Java, ils implémentent des méthodes qui prennent en entrée un évènement, et qui ne renvoient rien.

Sous Matlab, la classe « handle » (classe de base pour les objets) propose nativement plusieurs méthodes pour gérer les évènements. Les points à retenir sont :

— on définit les évènements avec le mot-clé « events »

— on ajoute les écouteurs avec la méthode « addlistener »

— on diffuse un évènement par la méthode « notify »

— les évènements sont soit une instance de « event.EventData », soit des sous-classes de la classe « event.EventData ».

Ce chapitre présente la définition des évènement et des écouteurs, puis le mécanisme de diffusion des évènements. Enfin, une proposition de classe de type « Ecouteur » est présentée.

## 10.1. Définir un évènement

On définit un évènement comme pour les propriétés de classe, par la directive « events ». On doit juste donner l'identification de l'évènement.

```
1 events
2     Overflow    % definition de l'evenement
3 end
```

C'est tout !

## 10.2. Diffuser un évènement

On lance un appel aux fonctions écouteur de la façon suivante :

```
notify(obj, 'Overflow');
```

Ou en spécifiant le type d'évènement :

```
notify(obj, 'Overflow', SpecialEventDataClass(value));
```

Un exemple un peu plus fourni : on crée un nouvel objet avec un champ nommé « prop1 », et on envoie un évènement si on modifie ce champ avec une valeur numérique supérieure à 10.

```
classdef DemoGestionEvent < handle
% Ne marche que pour des classes derivees de handle
properties
   Prop1 = 0;  % une propriete de classe
end
events
   Overflow   % identification de l'evenement
end
methods
   function set.Prop1(obj,value) % surcharge de la modification de la propriete
      orgvalue = obj.Prop1;
      obj.Prop1 = value;
         if (obj.Prop1 > 10)
            % On envoie un evenement personnalisé
            notify(obj, 'Overflow', SpecialEventDataClass(orgvalue));
         end
   end
end % fin des methodes
end % fin du classdef
```

## 10.3. Définir un listener

On a deux possibilités pour définir ses écouteurs : par pointeur de fonction et par objet spécialisé.

### 10.3.1. Par pointeur de fonction

L'idée est de fournir à la classe principale un pointeur vers une fonction de traitement. La fonction doit suivre le format suivant : le premier argument correspond à l'objet qui envoie l'évènement, le deuxième argument correspond à l'objet évènement proprement dit.

Un exemple est donné dans le listing suivant :

```
function overflowHandler(eventSrc, eventData)
   disp('The value of prop1 is verflowing');
   disp(['Its value was: ' num2str(eventData.OrgValue)]);
   disp(['Its current value is: ' num2str(eventSrc.Prop1)])
end
```

Pour associer l'évènement à la classe, on utilise la méthde « addlistener », à laquelle on précise le nom de l'évènement et le pointeur de fonction qui gère l'évènement.

```
addlistener(obj, 'EventName', CallbackFunction);
```

Dans l'exemple courant, cela correspond au code suivant :

```
dge = DemoGestionEvent; % objet à écouter
addlistener(dge, 'Overflow', @overflowHandler);
```

### 10.3.2. Par classe spécialisée

Une alternative est de créer sa classe écouteur sur mesure. Il faut définir une classe qui implémente une méthode d'interception d'évènements de la manière suivante :

```
methods
   function listenMyEvent(this, source, event)
      % this   - instance de la classe écouteur
      % source - objet qui génère l'évènement
      % event  - les données associées à l'évènement
      ...
   end
end
```

On peut ensuite ajouter l'écouteur (c'est à dire le pointeur de fonction de l'objet) à l'objet qui envoie les évènements.

```
hlistener = addlistener(eventSourceObj, 'Overflow', @obj.listenMyEvent)
```

## 10.4. Regrouper la gestion des évènements

Le fait d'utiliser une classe pour intercepter les évènements a aussi l'avantage que l'on peut définir des écouteurs pour plusieurs évènements, et les regrouper.

Exemple pour la définition d'une classe abstraite pour gérer les évènements d'une procédure d'optimization :

```
classdef OptimizationListener
methods (Abstract)
    optimizationStarted(this, source, event)
    optimizationIterated(this, source, event)
    optimizationTerminated(this, source, event)
end
```

On crée ensuite une ou plusieurs classes dérivées qui implémentent ces 3 méthodes.

Pour un objet « optimizable », on ajoute 3 évènements, et une méthode addOptimizationListener :

```
classdef OptimizableClass < handle
...
events
    OptimizationStarted
    OptimizationIterated
    OptimizationTerminated
end
...
methods
    function addOptimizationListener(this, listener)
```

```
11          addlistener(this, 'OptimizationStarted', @listener.optimizationStarted);
12          addlistener(this, 'OptimizationIterated', @listener.optimizationIterated);
13          addlistener(this, 'OptimizationTerminated', @listener.optimizationTerminated);
14      end
15  end
```

Exemple d'utilisation dans un script :

```
1  optimizer = NelderMeadSimplexOptimizer();
2  listener = OptimizedValueDisplay(figure(1));
3  optimizer.addOptimizationListener(listener);
4  optimizer.start();
```

## 10.5. Personnaliser un évènement

Par défaut, un évènement est une instance de classe « event.EventData ». Si on veut un traitement plus fin, il est possible de créer une classe d'évènement personnalisée :

```
1  classdef SpecialEventDataClass < event.EventData
2     properties
3        OrgValue = 0;
4     end
5     methods
6        function eventData = SpecialEventDataClass(value)
7              eventData.OrgValue = value;
8        end
9     end
10 end
```

Comme on le voit ici, une classe d'évènement doit hériter de « event.EventData ».