UNIT 6

Advanced Programming

1. Introduction

The aim of this unit is to show other possibilities of MATLAB. There are no Exercises associated with this unit.

Next we see issues such as the relationship between MATLAB and object-oriented programming, the interface for communication with other languages and software (APIs) and communication with hardware elements.

2. Object oriented programming

MATLAB classes: The main data types (classes) in MATLAB are the following:

- **`double`** (*double-precision floating-point number array*),
- **`single`** (*single-precision floating-point number array*),
- **`char`** (*character array*),
- **`logical`** (*array of true and false values*),
- **`int8`** y **`uint8`** (*8-bit signed integer array, 8-bit unsigned integer array*),
- **`int16`** y **`uint16`** (*16-bit signed integer array, 16-bit unsigned integer array*),
- **`int32`** y **`uint32`** (*32-bit signed integer array, 32-bit unsigned integer array*),

- `int64` y `uint64` (*64-bit signed integer array, 64-bit unsigned integer array*),
- `cell` (*cell array*),
- `struct` (*struct array*),
- `function_handle` (*array to call functions*)

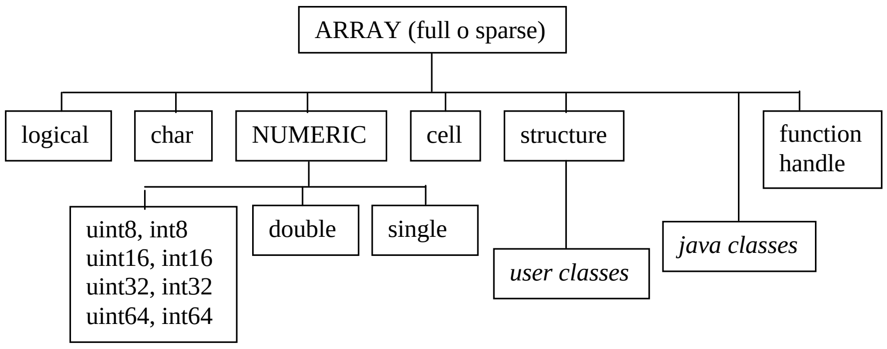


Fig. 1

To see the class of an object use the **`class`** command (objects class are also available in the workspace window),

```
>> x=@hello
x =
    @hello
>> class(x)
ans =
function_handle
```

Operations: For each class, MATLAB defines some specific operations. For example, you can add double's but not cell's. You can concatenate char's but not struct's.

Objects: From the v5 MATLAB allows the creation of new classes by the user and the possibility to define new operations for basic data types. The variables in each class (or data type) are called objects. The object-oriented programming (OOP) involves creating and using such objects.

Methods: The collection of rules or M files that redefine operators and functions are called methods. Operations on objects are specified by methods that encapsulate the data and redefine (overload) operators and functions. The encapsulation prevents certain properties of objects to be visible from the command window. To access them you must use the methods defined for the class.

Overload: You can override the internal rules of operation or functions. This is called overload and the resulting operation or function is said to be overloaded.

When the MATLAB interpreter finds an operator (e.g., product), or a function with one or more input arguments, it first checks the data type or class of operators and then it acts accordingly with the internally defined rules. For example, by default the product is defined for variables of numeric type, but the product operation can be overloaded to also make the product possible between variables of type char (we must define what a char product is).

To facilitate the overloading of operators +, -, .*, *, ... MATLAB has defined the functions plus, minus, times, mtimes, ... (see example below). Functions can be overloaded in the same way than operators.

Class Directory: The rules for interpreting operators and functions are M-files that are stored on the so-called class directory. These directories are named @class (where class is the name of the variable). For example directory @char contains definitions for functions and operations on char data.

These directories cannot be directly seen in the MATLAB search path but they are subdirectories of directories that are listed in the MATLAB search path. There may be multiple directories @class for the same type of data (in this case, when MATLAB search functions in these directories it follows the order given in the search path).

Example 1. Sum operator. Overload and class directory [1]

In MATLAB the addition operator (or function plus) is defined for numeric values:

```
>> plus(3,8)
ans =
    11
>> 3+8
ans =
    11
```

When trying to use with strings, for example, 'asi' + 'asa', what it does is find the ASCII numeric equivalent of each operand,

```
>> x=double('asi')
x =
    97   115   105
>> y=double('asa')
y =
    97   115    97
```

and add them:

```
>> x+y
ans =
   194   230   202
>> plus('asi','asa')
ans =
   194   230   202
>> 'asi'+'asa'
ans =
   194   230   202
```

Next we overload this operator so that when the input arguments are two strings, instead of switching to ASCII and add, what it does is concatenate two char.

This function is defined in a new plus.m

```
function s=plus(s1,s2)

if ischar(s1)&ischar(s2)
    s=[s1(:)',s2(:)'];
end
```

For this new function can be used when the input arguments are char data, you must save it in any subdirectory bearing the name <@char>. For example, within the directory <work> create the <@char> and save this function within it. Then, verify its operation:

```
>> 'asi'+'asa'
ans =
asiasa
>> plus('asi','asa')
ans =
asiasa
```

Note that if instead of being in <work\@char> it is in <work> MATLAB is not aware of the overload.

As MATLAB loads all subdirectories of class on startup, if a newly created directory is not seen, then you can or restart MATLAB or run the **`rehash`** command.

Other useful functions are: **`methods`**, **`isa`**, **`class`**, **`loadobj`** and **`saveobj`**.

Classes created by the user: To create a new class, for example, the class **`polynomio`** it is necessary to create the class directory **`@polynomial`**. This directory should contain at least two function files: **`polinomio.m`** and **`display.m`**. The first one is the builder of the class while the second is used to display the new variable in the command window. Apart from these two files, method M-files must be defined to operate the new class.

Instance: The builder file **`polinomio.m`** creates an instance of the class. The instance is an object that uses the methods that overload the operation of the operators and functions.

Class builder file: This file has the same name than the class, **`polinomio.m`**. It must manage three types of entries: (1) if no input arguments are passed it must generate an empty variable, (2) if the input argument is of the same class it should be passed directly to the output, (3) if the input argument are data to create the new class, it must generate a variable of that class. This requires checking that the input arguments are valid and then store them in the fields of a structure. The new variable is created when these fields are filled in and the function class is executed.

Example 2. Classes defined by the user. Methods [2]

In the <work> directory we have created the subdirectory <@ polynomio>.

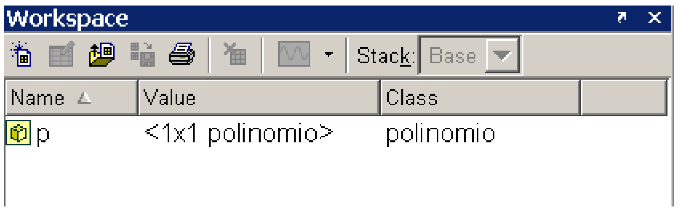
Then in that subdirectory, we create the builder file polinomio.m

```
function p=polinomio(vector_coefs)

if nargin == 0
   p.c = [];
   p = class(p,'polinomio');
elseif isa(vector_coefs,'polinomio')
   p = vector_coefs;
else
   p.c = vector_coefs(:).';
   p = class(p,'polinomio');
end
```

We check it:

```
>> p = polinomio([1 0 -2 -5])
p =
  polinomio object: 1-by-1
```

Yes, it appears at the workspace:



```
Name      Value             Class
p         <1x1 polinomio>   polinomio
```

Thus, polinomio class displays polynomials by means structures with a single field .c that contains the coefficients. This field is only available for the methods in the directory @polinomio.

```
>> p.c
??? Access to an object's fields is only permitted within its methods.
```

Apart from the builder method polinomio.m, for the class to be useful we must be able to manipulate objects. Therefore, we will implement the following methods: Method for converting a polinomio to double, method for converting polinomio to char polynomial, method of display, * operator overloading.

Method for converting a polinomio to double,

```
function c = double(p)
% POLINOMIO/DOUBLE
% Convierte el objeto polinomio en un vector de coeficientes
% Sintaxis:  c=double(p)
c = p.c;
```

Check it:

```
>> p=polinomio([1 0 1]);
>> c=double(p)
c =
     1     0     1
```

Method to convert **`polinomio`** to **`char`**,

```
function s = char(p)
% POLINOMIO/CHAR Convierte el objeto polinomio en char
% Sintaxis:  s=char(p)

if all(p.c==0),
    s='0';
else
   d=length(p.c)-1;%orden
   s=[];
   for a = p.c;
      if a ~= 0;
         if ~isempty(s)
            if a > 0,s = [s ' + '];
            else,s = [s ' - '];a = -a;
            end
         end
         if a ~= 1 | d == 0
            s = [s num2str(a)];
            if d > 0,s = [s '*'];end
         end
         if d >= 2,s = [s 'x^' int2str(d)];
         elseif d == 1, s = [s 'x'];
         end
      end
      d = d - 1;
   end
end
```

Check it:

```
>> p=polinomio([1 0 3 0 -2]);
>> s=char(p)
s =
x^4 + 3*x^2 - 2
```

The method named **`display`** uses the latter function:

```
function display(p)
% POLINOMIO/DISPLAY Muestra el objeto polinomio
% en la ventana de comandos
disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp(['    ' char(p)])
disp(' ');
```

Check it:

```
>> p=polinomio([1 0 3 0 -2])
```

```
p =
   x^4 + 3*x^2 - 2
```

Product overload:

```
function z = mtimes(x,y)
% POLINOMIO/MTIMES  Implementa x*y para objetos polinomio
x = polinomio(x);
y = polinomio(y);
z = polinomio(conv(x.c,y.c));
```

Check it:

```
>> x=polinomio([1 1])
x =
   x + 1
>> y=polinomio([1 2])
y =
   x + 2
>> x*y
ans =
   x^2 + 3*x + 2
```

It is also suggested to overload the addition operator (function plus) so that it add up polinomio objects directly.

Other functions and operators that could be overloaded are: minus, plot, roots, diff, polyval, subsref (subscribers reference) ...

To list the methods associated with the class polinomio, you can use the methods command:

```
>> methods('polinomio')
Methods for class polinomio:
char       display    double     mtimes     polinomio
```

Precedence: The user-defined classes have precedence over the built-in classes. In simple applications there is usually no conflict but as the number and complexity of classes grows, you should use the functions inferiorto and superiorto (in the builder file) to force the precedence of the classes.

Inheritance: You can create a hierarchy of parents and children where the latter inherit data fields and methods of their parents. A child class can inherit from one parent or more.

Aggregation: Classes can be created by aggregation, i.e., an object type can contain other objects.

3. Application Program Interface

Application Program Interface (API): Sometimes we need to communicate MATLAB with external data and other software. This is the role of the so-called *Application Program Interface* (API). It allows executing C and FORTRAN routines, exporting/importing data and establishing client/server relations between MATLAB and other programs. Other applications are the communication with external hardware (data acquistion cards or DSPs). See, e.g., the Real Time Toolbox.

3.1 MATLAB array

Only the object **`mxArray`** is considered. All variables (`double`, `char`, `sparse`, `uint8`, `cell` or `struct` types) are included inside it. Data are stores by columns.

For instance, the variable

$$x = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

is stored as the following column vector: $\begin{bmatrix} a & c & e & b & d & f \end{bmatrix}^T$.

A `mxArray` object includes two columns. In the first one, real values are stored; in the second one complex values are stored.

All functions that handle `mxArray`‘s include the prefix `mx`. For instance:

```
    x=mxGetScalar(prhs[0]);
    y=mxGetPi(prhs[0]);
```

The vector `prhs` (*pointer right hand side*) contains the pointers of the input arguments (*right hand side*). Note that the element `prhs[0]` refers to the first component of vector `prhs` (recall that in m-files the first index of a vector is 1, not 0). The function `mxGetScalar` searches for the position indicated by `prhs[0]`, extracts the data (which must be a scalar value) and stores it in the `x` variable.

On the other hand, in `mxGetPi`, `prhs[0]` is a pointer over the column corresponding to complex numbers and it establishes that the elements of complex variable `y` start in `prhs[0]`. `mxGetPr` is the same as `mxGetPi`, but applied to the real column of the `mxArray`.

3.2 MEX files

MEX files are routines written in C or FORTRAN that can be executed from MATLAB. They are dynamical linked routines (in Windows their extension is `*.dll`). The MATLAB interpreter automatically loads them and executes them.

C MEX files consists of three parts:

- Header: It contains the corresponding `include`, `define`,... The command `#include “mex.h”` is mandatory.
- Computational subroutine: C instructions that execute the program.
- Gateway subroutine: It order to make it interpretable by MATLAB.

The gateway routine is declared as:

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[],)
     {
     }
```

where

`nlhs` is the *number of left hand side arguments*.

`nrhs` is the *number of right hand side arguments*.

`plhs` is the *pointer to left hand side arguments*: This vector contains the pointers to the values of the output arguments. For instance, `plhs[0]=6` means that the first output argument starts at the 6th position of the `mxArray`; `plhs[1]=40`, means that the second output argument starts at the 40th position of the `mxArray`; `plhs[0]=NULL`, means that there is no output arguments.

`prhs` is the *pointer to left hand side arguments*.

It is on the gateway subroutine where the number, type and consistency of the input and output arguments are checked. Warning and error messages are generated here. The gateway subroutine gets the input arguments, creates the matrices that will contain the output arguments, calls the computational subroutine, and stores the output arguments in the corresponding position of the `mxArray`.

Example 3. MEX files

To create a C MEX file that multiplies a scalar value by 2, do the following: Edit a text file of name `mult2.c` with the following instructions:

```
#include “mex.h”

/*computational subroutine*/
```

```
void mult2(double y[], double x[])
{
      y[0]=2.0*x[0];
}

/*gateway subroutine*/
void mexFunction(int nlhs, mxArray *plhs[],
 int nrhs, const mxArray *prhs[],)
{
      double *x, *y;
      int mfil,ncol;

 /* check that the input argument is double type*/
 mfil=mxGetM(prhs[0]);
 ncol=mxGetN(prhs[0]);
if(!mxIsDouble(prhs[0]))
      {mexErrMsgTxt("La entrada ha de ser ‘double’");}

 /* create the matrix that will contain the output */
 plhs[0]=mxCreateDoubleMatrix(mfil,ncol,mxREAL);

 /* assign pointers to the input and output */
 x=mxGetPr(prhs[0]);
 y=mxGetPr(plhs[0]);

 /* call the computational subroutine */
 mult2(y,x);
}
```

Once the code is written, save the file and turn back to the MATLAB command window. Call the **`mex`** utility to compile, link, and generate the executable file `mult2.dll`:

```
mex mult2.c
```

Finally, check the usage:

```
» mult2(4.3)
ans =
    8.6000
```

Exercise 1. Consider the code of Example 3. Write the additional instructions that generate error messages for the following cases:

1) More than one input argument, e.g., `mult2(3,4)`. (Use `nrhs`).
2) More than one output argument, e.g., `[x,y]=mult2(5)`. (Use `nlhs`).
3) A complex input argument, e.g., `mult2(3*j)`. (Use `mxIsComplex`).
4) A non-scalar input argument, e.g., `mult2(ones(3)).` (Use `mfil`, `ncol`).

Generate the new file `mult2.dll` and check the results. □

4. Extension to JAVA

(Hanselman,05) Java is very integrated to the MATLAB environment. The Java virtual machine is the foundation of the MATLAB user interface. Java classes, objects and methods can be manipulated within MATLAB. See (Hanselman, 2005) for more details. Next examples are based on it.

Example 2. Use of the java.net package

```
>> me=java.net.InetAddress.getLocalHost;
>> myip=me.getHostAddress

myip =

192.168.1.33
```

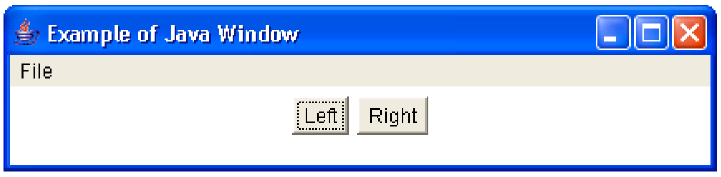
Suggestion: try also with getHostName.

Example 3. Use of the Java Abstract Window Toolkit

```
function example3(varargin)

import java.awt.*

persistent ventana mq x y w h bl br

if isempty(ventana)
    ventana=Frame('Example of Java Window');
    ventana.setLayout(FlowLayout);
    %window dimensions and colour
    x=450;y=550;w=430;h=100;
    ventana.setLocation(x,y);
    ventana.setSize(w,h);
    set(ventana,'Background',[.5 .5 .5]);
    %Menu bar
    mf=Menu('File');
    mq=MenuItem('Quit');
    set(mq,'ActionPerformedCallback','example3(''quit'')');
    mf.add(mq);
    mb=MenuBar;
    mb.add(mf);
    ventana.setMenuBar(mb);
    %Buttons
    bl=Button('Left');
    br=Button('Right');
    set(bl,'MouseClickedCallback','example3(''left'')');
    set(br,'MouseClickedCallback','example3(''right'')');
    ventana.add(bl);
    ventana.add(br);
    %Show the window
    ventana.setVisible(1);
elseif nargin==0
```

```
    x=450;y=550;w=430;h=100;
    ventana.setLocation(x,y);
    ventana.setSize(w,h);
    ventana.setVisible(1);
elseif nargin==1
    switch varargin{1}
        case 'left', x=x-4;
        case 'right',x=x+4;
        case 'quit', ventana.setVisible(0);return
    end
    ventana.setBounds(x,y,w,h);
    ventana.setVisible(1);
end
```

The result id the following window:



5.   Interface with Windows programs

(Hanselman,05) The communication between applications depends on the particular operating system: UNIX and Linux use “pipes” to connect applications. The standard output of one program is redirected to the standard input of another. Microsoft Windows uses the DDE (Dynamic Data Exchange) to directly communicate applications in order to send commands or exchange data.

In Windows, OLE (Object Linking and Embedding) is a component framework built upon DDE and Visual Basic Extensions (VBE). Later, some parts of the OLE relating to graphical user interfaces (GUI) and the Internet were renamed ActiveX. Microsoft later renamed the entire component framework the Component Object Model (COM).

A COM object is an instance of a component object class that runs on a server application and is controlled by a client application. A COM object encapsulates all data and methods of the object and uses interfaces to access the methods of the object. An interface is a pointer to the methods of an object. COM objects can have multiple interfaces.

MATLAB supports COM objects, ActiveX controls, and DDE in a limited capacity. It supoorts the creation of COM objects within MATLAB to control other applications and can also act as a COM server and respond to requests from other applications.

Example 4. MATLAB Notebook.
It is a way to embed Matlab commands, results, and graphics in a word document. Microsoft Word is the only word processor supported by the Matlab Notebook. Matlab

Notebook connects to a Matlab session from a word document called an M-book that uses some Word macros and COM controls.

To activate the Matlab notebook, simply type:

```
>> notebook -setup

Welcome to the utility for setting up the MATLAB Notebook
for interfacing MATLAB to Microsoft Word

Choose your version of Microsoft Word:
[1] Microsoft Word 97 (will not be supported in future
releases)
[2] Microsoft Word 2000
[3] Microsoft Word 2002 (XP)
[4] Microsoft Word 2003 (XP)
[5] Exit, making no changes

Microsoft Word Version: 4
```

Maybe Matlab is unable to automatically locate Normal.dot. In such a case, a dialog box will be presented to select the directory in which Normal.dot it is.

```
Notebook setup is complete.
```

Note that a mbook.dot file has appeared in the same directory where normal.dot is.

To create a m-book file, you can use Matlab or directly Word.

In the first case, use the notebook command,

```
>> notebook
Warning: MATLAB is now an automation server
>>
```

which opens a new word document.

It is also possible to create a new M-book directly from Word: Open Word, then go to Tools > Templates and Add-ins, and include mbook.dot in the global templates area. Notice that in the Word menu bar, now appears the option Notebook. Menu items are: Define input cell, define autoinit cell,…

Type some Matlab commands in the Word Document, for example:

t=linspace(2,4*pi);
y=sin(t);
plot(t,y)

Use Notebook menu items to declare that these commands as a cell, and then evaluate them. When an input cell is evaluated, the statement is sent to the Matlab server for evaluation, and the results are inserted into the M-book document.

Notebook preferences can also be set from the Notebook menu.

Reference

[1] D. Hanselman and B. Littlefield, *Mastering MATLAB 7*, Pearson Prentice Hall, International Edition, 2005.
[2] Matlab documentation, The MathWorks.