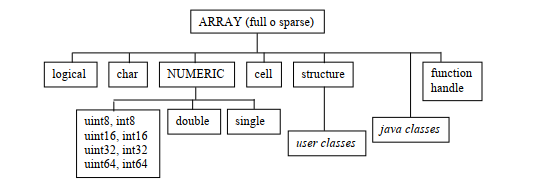
**PROGRAMACIÓN ORIENTADA A OBJETOS**

*Clases en MATLAB:* Los tipos de datos fundamentales (clases) en MATLAB son los siguientes:

* **double** (*double-precision floating-point number array*),
* **single** (*single-precision floating-point number array*),
* **char** (*character array*),
* **logical** (array de valores verdadero y falso),
* **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 de valores para hacer llamadas a funciones)



La función **class** indica la clase de un objeto (esta información también es visible en la ventana *workspace*), por ejemplo:

>> x=@hola

x =

@hola

>> class(x)

ans =   
function\_handle

*Operaciones:* Para cada una de las clases, MATLAB define unas operaciones concretas. Por ejemplo, se pueden sumar **double**’s pero no **cell**’s. Se pueden concatenar **char**’s pero no **struct**’s.

*Objetos:* A partir de la v5 MATLAB permite la creación de nuevas clases por parte del usuario y la posibilidad de definir nuevas operaciones para los tipos de datos básicos. Las variables de cada clase (o tipo de datos) se llaman *objetos*. La programación orientada a objetos (OOP, *Object Oriented Programming*) consiste en crear y usar dichos objetos.

*Métodos:* La colección de reglas o ficheros M que redefinen operadores y funciones reciben el nombre de *métodos*. Las operaciones sobre objetos se especifican por medio de métodos que encapsulan los datos y redefinen (*sobrecargan*) operadores y funciones. El encapsulado de los objetos impide que ciertas propiedades sean visibles desde la ventana de comandos con lo que para acceder a ellas hay que usar los *métodos* definidos para la clase.

*Sobrecarga:* Es posible redefinir las reglas internas de una operación o función. Ello recibe el nombre de sobrecargar (overload) y la operación o función resultante se dice que está sobrecargada.

Cuando el intérprete de MATLAB se encuentra con un operador como, por ejemplo, el producto, o una función con uno o más argumentos de entrada, primero mira el tipo de datos o clase de los operadores y, a continuación, actúa en consecuencia de acuerdo con las reglas definidas internamente. Por ejemplo, por defecto el producto está definido para variables de tipo numérico pero la operación producto se puede sobrecargar para que también sea posible realizar el producto de variables de tipo char (habrá que definir en qué consistirá disho producto).

Para facilitar la sobrecarga de los operadores **+**, **-**, **.\***, **\***,… MATLAB tiene definidas las funciones **plus**, **minus**, **times**, **mtimes**. Para sobrecargar funciones se hace igual que para sobrecargar operadores.

*Directorio de clase:* Las reglas redefinidas para interpretar operadores y funciones son ficheros M de tipo función que se guardan en los directorios de clase (*class directories*) del MATLAB de nombre **@class** (donde **class** es el nombre de la variable). Por ejemplo los ficheros M del directorio **@char** redefinen funciones y operaciones sobre cadenas de caracteres.

Estos directorios no pueden estar directamente en el *search path* de MATLAB pero son subdirectorios de directorios que sí están en el *search path* de MATLAB. Es posible que haya múltiples directorios **@class** para el mismo tipo de datos (en ese caso, cuando MATLAB busque funciones en estos directorios seguirá el orden dado en el *search path*).

**Ejemplo 1. Operador suma. Sobrecarga y directorio de clase [1]**

En MATLAB el operador suma (o función plus) está definido para valores numéricos:

>> plus(3,8) ans =

11

>> 3+8

ans =

11

Cuando se intenta usar con cadenas de caracteres, por ejemplo, 'asi'+'asa', lo que hace es encontrar el equivalente numérico ASCII de cada operando,

>> x=double('asi')

x =

97 115 105 >> y=double('asa')

y =

97 115 97

y sumarlos:

>> x+y

ans =

194 230 202

>> plus('asi','asa') ans =

194 230 202

>> 'asi'+'asa'

ans =

194 230 202

A continuación se va a sobrecargar este operador para que, cuando los argumentos de entrada sean dos cadenas de caracteres, en vez de pasar a ASCII y sumar, lo que haga sea concatenar las dos cadenas de caracteres.

Para ello se crea una nueva función **plus.m**

|  |
| --- |
| function s=plus(s1,s2) |
|  |
| if ischar(s1)&ischar(s2) |
| s=[s1(:)',s2(:)']; |
| end |

Para que esta nueva función pueda usarse cuando los argumentos de entrada son cadenas de caracteres, hay que guardarla en cualquier subdirectorio que lleve el nombre **<@char>**. Por ejemplo, dentro del directorio **<work>** creamos el directorio **<@char>** y guardamos esta función dentro de él. A continuación, verificamos su funcionamiento:

>> 'asi'+'asa'

ans =

asiasa

>> plus('asi','asa') ans =

asiasa

Notar que si en vez de estar en **<work\@char>** está solo en **<work>** MATLAB no se entera de la sobrecarga.

Como MATLAB carga todos los subdirectorios de clase en el **startup**, si un directorio recién creado no se ve se puede, o bien reiniciar MATLAB, o bien ejecutar **rehash**.

Otras funciones útiles son: **methods**, **isa**, **class**, **loadobj** y **saveobj**.

*Clases creadas por el usuario:* Para crear una nueva clase, por ejemplo, la clase **polinomio** hay que crear el directorio de clase **@polinomio**. Este directorio debe contener como mínimo dos ficheros M de tipo *function*: **polinomio.m** y **display.m**. El primero de ellos es el *constructor* de la clase mientras que el segundo se usa para visualizar la nueva variable en la ventana de comandos. Aparte de estos dos ficheros habrá que definir ficheros M de *métodos* que permitan operar con la nueva clase creada.

*Instancia:* El fichero constructor **polinomio.m** lo que hace es crear una instancia de la clase. La instancia es un objeto que utiliza los métodos que sobrecargan el funcionamiento de los operadores y funciones cuando hay que aplicarlos al objeto considerado.

*Fichero constructor de clase:* Debe llamarse como la clase, es decir, **polinomio.m**. Debe manejar tres tipos de entradas: (1) si no se le pasan argumentos de entrada, debe generar una variable vacía; (2) si el argumento de entrada es de su misma clase debe pasarlo directamente a la salida; (3) si el argumento de entrada son datos para crear la nueva clase, debe crear una variable de dicha clase. Para ello hay que comprobar que los argumentos de entrada sean válidos y, a continuación, almacenarlos en los campos de una estructura. La nueva variable se crea cuando dichos campos son rellenados y se ejecuta la función **class**.

**Ejemplo 2. Clases creadas por el usuario. Métodos [2]**

En el directorio **<work>** hemos creado el subdirectorio **<@polinomio>**.

A continuación, en dicho subdirectorio, se crea el fichero constructor **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 |

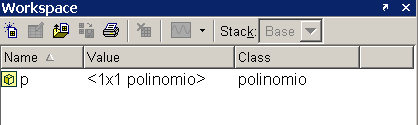
Lo probamos:

>> p = polinomio([1 0 -2 -5])

p =

polinomio object: 1-by-1

Vemos que, efectivamente, aparece en el workspace.



Así pues, la clase polinomio representa polinomios por medio de estructuras con un solo campo .c que contiene los coeficientes. A este campo solo pueden acceder los métodos del directorio **@polinomio**.

>> p.c

??? Access to an object's fields is only permitted within its methods.

Aparte del método contructor **polinomio.m**, para que la clase sea útil debemos poder hacer cosas con sus objetos. Por ello, vamos a implementar los siguientes métodos: método para la conversión de polinomio a double, método para la conversión de polinomio a char, método de display, sobrecarga del operador \*.

Método para la conversión de **polinomio** a **double**,

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

Verificamos su funcionamiento:

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

c =

1 0 1

Método para la conversión de **polinomio** a **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 |

Verificamos su funcionamiento:

>> p=polinomio([1 0 3 0 -2]); >> s=char(p)

s =

x^4 + 3\*x^2 - 2

El método **display** hace uso de la función anterior:

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

Verificamos su funcionamiento:

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

p =

x^4 + 3\*x^2 - 2

Sobrecarga del operador producto

|  |
| --- |
| 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)); |

Verificamos su funcionamiento:

>> x=polinomio([1 1]) x =

x + 1

>> y=polinomio([1 2]) y =

x + 2

>> x\*y

ans =

x^2 + 3\*x + 2

Se suguiere sobrecargar también el operador suma (función **plus**) a fin de que pueda sumar directamente objetos polinomio.

Otras funciones y operadores que se podrían sobrecargar son: **minus**, **plot**, **roots**, **diff**, **polyval**, **subsref** (*subscripted reference*),…

Para listar los métodos asociados a la clase polinomio, se puede usar la función **methods**:

>> methods('polinomio')

Methods for class polinomio:

char display double mtimes polinomio

*Precedencia:* Las clases definidas por el usuario tienen precedencia frente a las clases *built-in*. En aplicaciones simples no suele haber conflictos pero a medida que el número y complejidad de las clases crece, se recomienda usar las funciones **inferiorto** y **superiorto** (dentro del fichero constructor) para forzar la precedencia de las clases.

*Herencia:* Se puede crear una jerarquía de padres e hijos donde estos últimos heredan campos de datos y métodos de los padres. Una clase hija puede heredar de un solo padre o de varios.

*Agregación:* Se pueden crear clases mediante agregación, es decir, un tipo de objeto puede contener a otros objetos.

**RELACIÓN CON JAVA**

(Hanselman) Java y MATLAB están muy relacionados. La máquina virtual de Java (JVM, *Java Virtual* *Machine*) es el intérprete que está detrás del GUI de MATLAB y MATLAB puede manipular las clases, objetos y métodos de Java.

*Máquina virtual de Java:* Java es un lenguaje de programación orientado a objetos, diseñado para entornos distribuidos, donde hay diferentes tipos de ordenadores y diferentes sistemas operativos. Los programas de Java se compilan en un *bytecode* independiente del tipo de ordenador o sistema operativo. La máquina virtual de Java es el programa que se encarga de interpretar este *bytecode* en el código máquina particular de cada ordenador, de manera que mientras el programa se ejecuta se ajustan las posibles diferencias en cuanto a la longitud de los comandos, formas de almacenamiento de datos, etc.

*Librerías:* Reciben el nombre de *class libraries*, *toolkits* o *packages*. Son colecciones de clases de Java relacionadas. La *Abstract Windowing Toolkit* (java.awt.\*) es la que proporciona los servicios para crear y manejar ventanas y los servicios de GUI. La *Net* *Class Library* (java.net.\*) proporciona servicios de comunicación e Internet.

**Ejemplo 9. Java Net Package [1]**

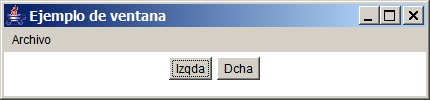
|  |
| --- |
| >> id=java.net.InetAddress.getLocalHost |
| id = |
| UPCNET-6D1E/127.0.0.1 |
| >> id.getHostName |
| ans = |
| UPCNET-6D1E |
| >> id.getHostAddress |
| ans = |
| [127.0.0.1](http://127.0.0.1) |

**Ejemplo 10. Java Abstract Window Toolkit [1]**

|  |
| --- |
| function vent\_movil(varargin) |
|  |
| import java.awt.\* |
| persistent ventana mq x y w h bl br |
|  |
| if isempty(ventana) |
| ventana=Frame('Ejemplo de ventana'); |
| ventana.setLayout(FlowLayout); |
| %dimensiones y color |
| 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('Archivo'); |
| mq=MenuItem('Salir'); |
| set(mq,'ActionPerformedCallback','vent\_movil(''salir'')'); |
| mf.add(mq); |
| mb=MenuBar; |
| mb.add(mf); |
| ventana.setMenuBar(mb); |
| %botones |
| bl=Button('Izqda'); |
| br=Button('Dcha'); |
| set(bl,'MouseClickedCallback','vent\_movil(''izqda'')'); |
| set(br,'MouseClickedCallback','vent\_movil(''dcha'')'); |
| ventana.add(bl); |
| ventana.add(br); |
| %mostrar ventana |
| 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 'izqda', x=x-20; |
| case 'dcha',x=x+20; |
| case 'salir', ventana.setVisible(0);return |
| end |
| ventana.setBounds(x,y,w,h); |
| ventana.setVisible(1); |
| end |

El resultado es la siguiente ventana:



que se mueve hacia la izquierda o hacia la derecha conforme se hace clic en el botón correspondiente. Se sugiere ampliar la función para que también se pueda mover hacia arriba y hacia abajo.

**RELACIÓN CON WINDOWS**

Con respecto a *Windows* (win32), MATLAB puede usar objetos COM (*Component Object Model*), controles ActiveX y DDE (*Dynamic Data Exchange*). MATLAB puede actuar como un servidor COM Automation a fin de comunicar con visual basic (VB) o con visual basic para aplicaciones (VBA) como es el caso de Microsoft Excel, PowerPoint y Word.

En UNIX y Linux para comunicar programas entre sí se usan *pipes*. En ellas, la salida estándar de un programa se redirecciona a la entrada estándar del otro programa.

En *Windows*, el DDE permite a las diversas aplicaciones comunicarse entre ellas para enviar comandos e intercambiar datos. El llamado OLE (*Object Linking and Embedding*) se construyó a partir de DDE y VBX (*Visual Basic Extensions*). Más tarde, algunas partes del OLE relacionadas con Internet y con las interficies gráficas de usuario GUI (*Graphical User Interface*) fueron renombradas como *ActiveX*. Y, finalmente, Microsoft renombró todo el entorno de componentes como COM (*Component Object Model*).

1 MATLAB como cliente o servidor COM (*Component Object Model*)

MATLAB puede funcionar como cliente COM. Para ello, la función **actxserver** crea un objeto COM y abre la conexión con un *COM Automation server* (éste puede ser Word, PowerPoint,...). Una vez creado el objeto se puede acceder a sus propiedades con **set** y **get**. Por ejemplo:

>> s=actxserver('powerpoint.application'); >> s=actxserver('excel.application');

>> s=actxserver('word.application');

>> get(s)

Para ver los métodos asociados a los objetos COM, basta con teclear

>> methods COM

Methods for class COM:

addproperty events invoke propedit set delete get load release   
deleteproperty interfaces move save

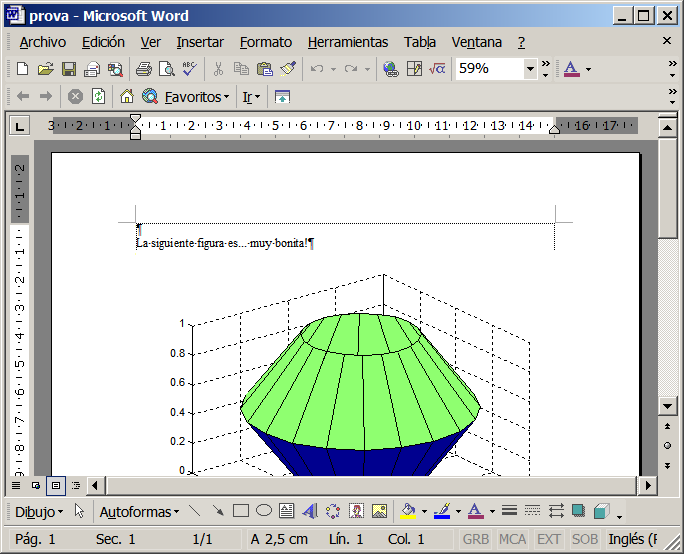
Para ver la ayuda de cada uno de estos métodos: >> help COM/invoke.

**Ejemplo 11. MATLAB como cliente y WORD como servidor [1]**

El siguiente fichero M, **figu\_word.m**, crea una figura, abre el word, pega la figura, y cierra el word.

|  |
| --- |
| function figu\_word(nomf) |
|  |
| %FIGU\_WORD abre un doc de word y pega la figura actual |
|  |
| %creamos la figura |
| close all,clc,cylinder([1 2 1]); |
|  |
| if nargin<1 |
| [fitxer,directori]=uiputfile('\*.doc','Indicar o crear el fichero |
| word:'); |
| if isequal(fitxer,0),return,end |
| nomf=fullfile(directori,fitxer); |
| end |
|  |
| %copiamos la figura como metafile |
| if nargin<2 |
| print('-dmeta'); |

|  |
| --- |
| end |
|  |
| %iniciar word |
| wrd=actxserver('word.application'); |
| wrd.Visible=1; |
|  |
| %abrir el nomf.doc (o crearlo si no existe) |
| if exist(nomf,'file'); |
| docu=invoke(wrd.Documents,'Open',nomf); %abrir nomf.doc |
| else |
| docu=invoke(wrd.Documents,'Add'); %o crearlo |
| end |
|  |
| %añadir texto a nomf.doc |
| s=docu.Content; |
| s.InsertParagraphAfter; %punto y aparte |
| invoke(s,'InsertAfter','La siguiente figura es... muy bonita!'); |
| s.InsertParagraphAfter; |
| invoke(s,'Collapse',0); %para que la fig se pegue después del texto |
|  |
| %pegar la figura: (1): picture format, (3): float over text |
| invoke(s,'PasteSpecial',0,0,1,0,3); |
|  |
| %salvar y cerrar nomf.doc |
| if exist(nomf,'file') |
| docu.Save; |
| else |
| invoke(docu,'SaveAs',nomf,1); |
| end |
| docu.Close; |
|  |
| %salir de word y cerrar la conexión |
| wrd.Quit; |
| delete(wrd); |
| return |

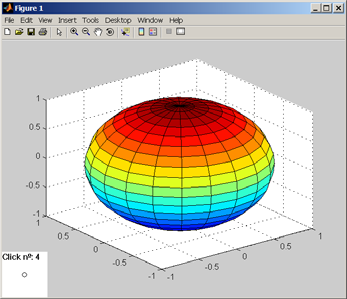
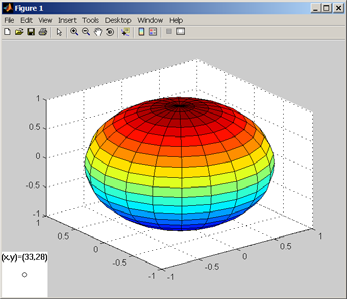
El resultado es:

El MATLAB básico dispone de dos controles ActiveX muy simples: mwsamp.ocx y mwsamp2.ocx con sus librerías asociadas: mwsamp.tlb y mwsamp2.tlb. La clase mwsamp solo tiene un evento (Click) mientras que la clase mwsamp2 es una ampliación que incluye los eventos DblClick y MouseDown.

**Ejemplo 12. Controles ActiveX [1]**

La función cdemo.m ilustra el funcionamiento de la clase mwsamp2. Con un clic simple sobre el control ActiveX lo que se muestra son las coordenadas del clic. Si se hace un doble clic lo que se muestra es el número de dobleclics realizados hasta el momento.

|  |
| --- |
| function cdemo(varargin) |
|  |
| persistent numclicks h |
|  |
| if nargin==0 |
| clc,close all, |
| f=figure;sphere;numclicks=0; |
|  |
| h=actxcontrol('MWSAMP.MwsampCtrl.2',[0 0 90 90],f,'cdemo'); |
| set(h,'Label','Clicar'); |
| h.Radius=5; |
| invoke(h,'Redraw'); |
|  |
| else |
|  |
| if strcmp(varargin{end},'DblClick') |
| numclicks=numclicks+2; |
| h.Label=['Click nº: ',num2str(numclicks)]; |
| elseif strcmp(varargin{end},'MouseDown') |
| h.Label=['(x,y)=(',num2str(varargin{5}),',',num2str(90- |
| varargin{6}),')']; |
| end |
|  |
| h.Redraw; |
|  |
| end |



2 *Notebook* de MATLAB

El *notebook* de MATLAB permite incrustar comandos, resultados y gráficos de MATLAB en un documento de *Microsoft* *Word* (Nota: el *notebook* de MATLAB está pensado sólo para *word*).

*Configuración:* Teclear la siguiente instrucción en la ventana de comandos:

|  |
| --- |
| >> notebook -setup |
|  |
| Welcome to the utility for setting up the MATLAB Notebook |
| for interfacing MATLAB to Microsoft Word |
|  |
| Setup complete |

Es posible que pida especificar la versión. En ese caso aparecerán las siguientes opciones:

|  |
| --- |
| >> 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 |

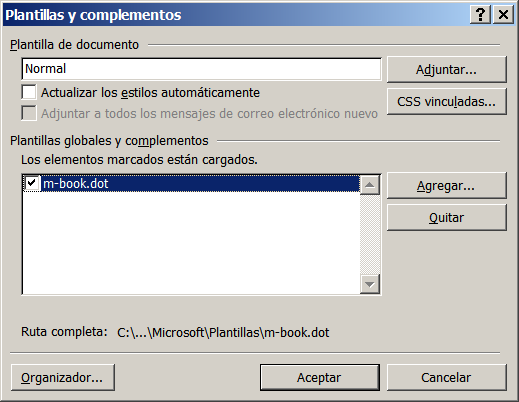
Es posible que MATLAB no pueda encontrar automáticamente el directorio donde se encuentra la plantilla **Normal.dot**. En ese caso, aparecerá un cuadro de diálogo a fin de seleccionar el directorio donde se encuentra la plantilla **Normal.dot** (por ejemplo, en C:> \Documents and Settings \ Administrador \ Datos de Programa \ Microsoft \ Plantillas).

Una vez localizada, la configuración se habrá completado,

Notebook setup is complete.

y un fichero de nombre **m-book.dot** habrá aparecido en el mismo directorio donde se encuentra la plantilla **normal.dot**

A continuación, hay que abrir el *Word* y seleccionar Herramientas à Plantillas y complementos… (Tools à Templates and Add-ins…) a fin de añadir la plantilla global **m-book.dot**. Notar que en la barra de menús de *Word* habrá aparecido la opción **Notebook**. Ver que las opciones son: Define Input Cell, Evaluate Cell, etc.



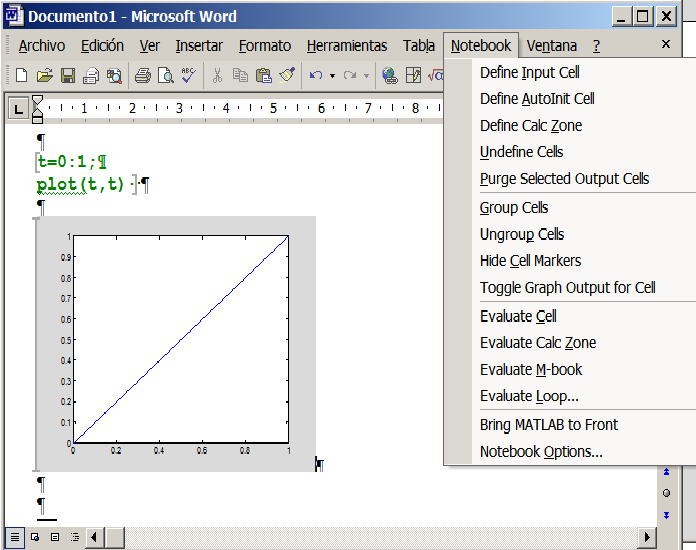
*Iniciar el notebook*: Si se teclea >>notebook en la ventana de comandos de MATLAB aparece el siguiente mensaje

|  |
| --- |
| >> notebook |
| Warning: MATLAB is now an automation server |
| >> |

y se abre el Word. Si el MATLAB está cerrado, abrir el Word, añadir la plantilla m- book.dot y ver que aparece la opción de menú Notebook.

*Uso del notebook:* Escribir algunas instrucciones de MATLAB en el documento Word. Seleccionarlas y con ayuda de las opciones de Notebook, definirlas como Input Cell. A continuación evaluarlas (Evaluate Cell).

Aparecerán los mensajes "MATLAB Automation Server Startup" (se abrirá el MATLAB) y "MATLAB computing". Finalmente, se ejecutarán las instrucciones tecleadas:



**ESTRUCTURAS (Struct)**

Las estructuras, también llamadas registros en muchos lenguajes de programación, son un tipo de dato soportado por todos los lenguajes de programación modernos. Una estructura permite crear variables con datos de distintos tipos. El poder almacenar datos de distintos tipos hace que una estructura pueda representar cualquier entidad real o imaginaria, como un alumno, un coche, una coordenada cartesiana ...Cada variable de una estructura se llama campo y tiene asociada un nombre. A diferencia de un array de celdas, en el que se utiliza un índice para acceder a sus datos, una estructura accede a sus campos a través de sus nombres, lo que hace que las estructuras tengan asociado un código muy fácil de leer y comprender. También es posible crear arrays de estructuras para almacenar una colección de estructuras, como los alumnos de una clase.

## Descripción

Un array de estructura es un tipo de datos que agrupa datos relacionados mediante contenedores de datos denominados campos . Cada campo puede contener cualquier tipo de datos. Acceda a los datos en un campo utilizando la notación de puntos del formulario structName.fieldName.

## Creación

Cuando tenga datos para poner en una nueva estructura, cree la estructura utilizando la notación de puntos para nombrar sus campos uno a la vez:

s.a = 1;

s.b = {'A','B','C'}

s = struct with fields:

a: 1

b: {'A' 'B' 'C'}

También puede crear una matriz de estructura mediante la función struct , que se describe a continuación. Puede especificar varios campos simultáneamente o crear una matriz de estructura no escalable.

### Sintaxis

[s = struct](https://la.mathworks.com/help/matlab/ref/struct.html#d122e76961)

[s = struct(field,value)](https://la.mathworks.com/help/matlab/ref/struct.html#d122e76970)

[s = struct(field1,value1,...,fieldN,valueN)](https://la.mathworks.com/help/matlab/ref/struct.html#d122e77042)

[s = struct([])](https://la.mathworks.com/help/matlab/ref/struct.html#d122e77098)

[s = struct(obj)](https://la.mathworks.com/help/matlab/ref/struct.html#d122e77109)

### Descripción

s = struct crea una estructura escalar (1-por-1) sin campos.

s = struct([field](https://la.mathworks.com/help/matlab/ref/struct.html" \l "d122e77135),[value](https://la.mathworks.com/help/matlab/ref/struct.html#d122e77155)) crea una matriz de estructura con el campo y los valores especificados. El argumento de entrada value puede ser cualquier tipo de datos, como un array numérico, lógico, de carácter o de celda.

* Si value es no una matriz de celdas, o si value es una matriz de celdas escalares, s es una estructura escalar. Por ejemplo, s = struct('a',[1 2 3]) crea una estructura 1 por 1, donde s.a = [1 2 3].
* Si value es una matriz de celdas no escalar, entonces s es una matriz de estructura con las mismas dimensiones que value. Cada elemento de s contiene el elemento correspondiente de value. Por ejemplo, s = struct('x',{'a','b'},'y','c') devuelve s(1).x = 'a', s(2).x = 'b', s(1).y = 'c', s(2).y = 'c'.
* Si value es una matriz de celdas vacías {}, s es una estructura vacía (0 por 0).

s = struct(field1,value1,...,fieldN,valueN) crea varios campos. Cualquier matriz de celdas no escalar del conjunto value1,...,valueN debe tener las mismas cotas.

* Si ninguna de las entradas de value son matrices de celdas, o si todas las entradas value que son matrices de celdas son escalares, s es una estructura escalar.
* Si alguna de las entradas de value es una matriz de celdas no escalar, s tiene las mismas dimensiones que la matriz de celdas no escalar. Para cualquier value que sea una matriz de celdas escalares o una matriz de cualquier otro tipo de datos, struct inserta el contenido de value en el campo correspondiente para todos los elementos de s .
* Si cualquier entrada value es una matriz de celdas vacía, {}, a continuación, la salida s es una estructura vacía (0 por 0). Para especificar un campo vacío y conservar los valores de los demás campos, utilice en su lugar [] como una entrada value .

s = struct([]) crea una estructura vacía (0 por 0) sin campos.

s = struct([obj](https://la.mathworks.com/help/matlab/ref/struct.html" \l "d122e77184)) crea una estructura escalar con nombres de campo y valores que corresponden a las propiedades de obj. La función struct no convierte obj, sino que crea s como una nueva estructura. Esta estructura no conserva la información de clase, por lo que las propiedades privadas, protegidas y ocultas se convierten en campos públicos en s. La función struct emite una advertencia al utilizar esta sintaxis.

Una *estructura* en matlab es una variable que agrupa varios *contenedores de datos* llamados campos. Cada campo puede contener cualquier tipo de datos (números, cadenas de texto, vectores, matrices …), y puede tener cualquier dimensión. Por ejemplo, si la información que se recoge para cada alumno de la universidad está formada por su nombre, apellidos, dni, año de nacimiento y cursos en que está matriculado, dicha información podría almacenarse en una estructura que llamaremos alumno definida del siguiente modo:

Construcción de una estructura

>> alumno.nombre="María";

>> alumno.apellido="Rodríguez";

>> alumno.DNI=12345678;

>> alumno.anioNac=2000;

>> alumno.cursosMatriculado=[1 2 3];

>> alumno

alumno =

scalar structure containing the fields:

nombre = María

apellido = Rodríguez

DNI = 12345678

anioNac = 2000

cursosMatriculado =

1 2 3

Obsérvese que los campos de la estructura alumno se especifican mediante su nombre separado por un punto del nombre de la estructura. En la misma estructura hay variables de tipo *string* (nombre y apellidos), variables numéricas (DNI y año de nacimiento) y vectores (cursos en que está matriculado)

En el ejemplo, hemos creado la estructura definiendo en cada linea el contenido de un campo. De manera equivalente podemos definir la estructura en una sola línea utilizando la función struct:

>> alumno=struct("nombre", "María","apellido","Rodríguez","DNI",12345678,"anionac",2000,"cursosMatriculado",[1 2 3]);

>> alumno

alumno =

scalar structure containing the fields:

nombre = María

apellido = Rodríguez

DNI = 12345678

anionac = 2000

cursosMatriculado =

1 2 3

Nótese que para utilizar esta sintaxis los nombres de los campos deben ir entrecomillados; a continuación de cada campo se introduce su valor, que sólo se entrecomilla si es de tipo carácter.

Es posible definir *vectores de estructuras* (*struct array*); así, por ejemplo, si quisiéramos guardar la lista de alumnos de la clase podríamos seguir añadiendo términos a la estructura anterior como si de un vector se tratase:

>> alumno(2)=struct("nombre","Pedro","apellido","González","DNI",87654321,"anionac",1999,"cursosMatriculado",[2 3]);

>> alumno

alumno =

1x2 struct array containing the fields:

nombre

apellido

DNI

anionac

cursosMatriculado

>> alumno.nombre

ans = María

ans = Pedro

>> alumno.apellido

ans = Rodríguez

ans = González

Existe la posibilidad de convertir todos los valores del mismo campo en un vector; por ejemplo, si queremos extraer todos los nombres de los alumnos a un único vector empleamos la sintaxis:

>> [alumno.nombre]

ans =

1×2 string array

"María" "Pedro"

Para acceder a los datos de algún campo particular de un alumno concreto utilizamos la siguiente sintaxis:

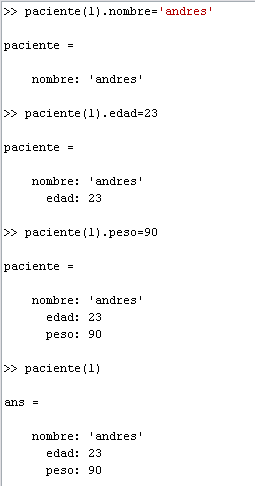
>> alumno(1).apellido

ans = Rodríguez

>> alumno(2).DNI

ans = 87654321

**Ejemplo 12:**



Algunos comandos útiles para estructuras son los que se explican a continuación.

* FIELDNAMES (nombre\_estructura): Nombra los campos que tienen la estructura, es decir, lo introducido anteriormente después del punto.
* SETFIELD, y GETFIELD.
* Setfield: sirve para cambiar un dato concreto en una estructura que ya tenías creada.

Hay que indicar el nombre de la estructura, el elemento, el campo, el elemento del campo que quiero modificar y el nuevo valor a introducir.

Por ejemplo, una vez creada la estructura la estructura:

>> persona(1).pesos = [ 20 30 40],

deseo cambiar el segundo dato por un cero:

>> persona = Setfield(persona,1,pesos,2,0)

Así, queda modificado el array persona, pero sólo en el elemento deseado.

* Getfield: extrae el campo indicado, en la forma:

Getfield (estructura, elemento, ‘campo’)

Por ejemplo, siguiendo con la estructura anterior, si se desea extraer el campo pesos:

>> Getfield (persona, 1, ‘pesos’)

>> ans

[20 0 40]

* CLASS (<nombre>): indica el tipo de array que es el introducido por su nombre.

Ejemplo:

>>class(persona)

>>ans

Sruct

* ISA: tiene una función similar a comando anterior pero de una forma más elaborada: El usuario preguntará por una variable en concreto y un tipo de dato. Si la variable coincide con el tipo de dato la respuesta será 1 y en caso contrario 0.

La sintaxis ha utilizar es la siguiente:

(<variable>, “tipo de dato”)

Cell

Char

Numeric

Por ejemplo si introducimos:

>> isa(persona,'struct')

ans =

1

* RMFIELD (<estructura>, ‘campo’): Elimina el campo de la estructura

**Ejemplo 13:**

Quiero guardar un listado de Alumnos, Edad y Calificaciones en Física, Álgebra y Fundamentos de Informática

Y hay que hallar:

1. Media por alumno.
2. Media de toda la clase en Física.
3. Alumno con mayor nota en Física.

Primero se crearían las diferentes estructuras:

ALUMNO (1). NOMBRE: ANTONIO

EDAD

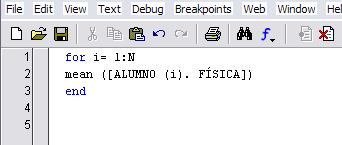
FÍSICA

ÁLGEBRA

INFORMÁTICA

1. MEAN ([ALUMNO (1). FÍSICA, ALUMNO (1).ÁLGEBRA, ALUMNO (1).INFORMÁTICA])

Para hacer con todos los alumnos se podría hacer con un programa como el siguiente:



b) MEAN ([ALUMNO. FÍSICA]).

c) Sería:

>> MAX([alumno.física])

Pero lo normal, es que lo que interese saber es cual es el alumno. Para saber ello:

[V, n]= MAX ([ALUMNO. FÍSICA])

Siendo V el valor y n la posición.

DISP (ALUMNO (n). NOMBRE)

**TAREA**

**SOLUCION DE SISTEMAS DE ECUACIONES**

La practica subir tanto en archivo pdf como y \*.m