

Introducción al lenguaje MATLAB

1. Descripción del entorno de programación MATLAB.
2. Datos en MATLAB:
 - a) Carácter vectorial de MATLAB.
 - b) Operadores de creación de matrices.
 - c) Tipos de datos usados en MATLAB
3. Programación en MATLAB:
 - a) Entrada / Salida de datos.
 - b) Operadores.
 - c) Sentencias de control de flujo.
4. Programas: scripts y funciones.
5. Gráficos en MATLAB:
 - a) Jerarquía de Objetos Gráficos
 - b) Funciones de alto nivel para gráficos

¿Por qué MATLAB?

- Interfaz de usuario “amigable” que facilita su uso por primera vez.
- Consta de un entorno muy completo con diversas aplicaciones integradas (editor, ventana de comandos, historial de los comandos tecleados, visor de variables en uso, etc.)
- Ampliamente usado en entornos académicos y profesionales.
- Es básicamente un lenguaje interpretado, lo que nos proporciona un inmediato “feedback”. Tras cada orden podemos ver de forma inmediata sus resultados (o los errores cometidos).
- Librerías de funciones muy completas: al poco de familiarizarnos con el podemos estar haciendo cálculos muy complejos.
- Librerías Gráficas incorporadas, facilitando presentación de datos con multitud de formatos gráficos.


ENTORNO de MATLAB

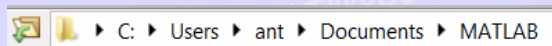
- Como en la mayoría de los lenguajes actuales, la versión actual de MATLAB es mucho más que un compilador, comprendiendo diversas aplicaciones, tales como:

1. La ventana de ordenes (**command window**)
2. Espacio de trabajo (**workspace**)
3. Historial de comandos (**history window**)
4. Editor (**edit**)
5. Escritorio de ayuda (**helpdesk**)

- Para empezar a trabajar, debemos entrar en nuestra primera sesión de MATLAB.

Sesión de Matlab

- Al lanzar MATLAB (icono ) nos aparece un entorno con varias aplicaciones en una única ventana.
- Estas aplicaciones pueden independizarse (undock) creándose ventanas independientes que pueden ocultarse por separado.
- Al iniciar la sesión es conveniente cambiar el directorio de trabajo



que será donde se guardan por defecto los programas/datos.

- La ventana principal es la de ORDENES o COMANDOS, donde introducimos ordenes tras el cursor (**>>**)
- Ventanas adicionales pueden abrirse como consecuencia nuestros comandos (hacer un plot provoca la creación de una nueva ventana), o al lanzar nuevas aplicaciones desde los menús.
- Para finalizar la sesión podemos cerrar la aplicación o teclear **exit** en la ventana de comandos.

Ventana de Ordenes (Command Window)

- En ella podemos introducir diversos tipos de ordenes:
 1. Sentencias de Matlab, (`x=(0:0.1:pi);y=sin(x);plot(x,y);`) que se ejecutarán al pulsar enter. Así podemos probar el efecto (o los errores) de cada orden de nuestro futuro programa.
 2. Llamadas a scripts o funciones previamente escritas por nosotros en el EDITOR.
 3. Ordenes como `helpdesk`, `edit`, `workspace` lanzan aplicaciones del entorno o nos permiten cambiar directorio de trabajo o el path (`cd`, `path`). En general esto es más cómodo hacerlo desde los menús.
 4. Comandos precedidos por ! (p.e. `!dir`, `!format c`;) no son procesados por MATLAB: se envían directamente al sistema operativo.

En la primera clase daremos ordenes directamente en la ventana de comandos.

En cuanto los programas se compliquen un poco (3/4 líneas) es mejor trabajar con el EDITOR, usando un script (fichero conteniendo una serie de comandos).

Primeros comandos en MATLAB

- El comando más sencillo de entrada de datos es la asignación (=)

```
>> a=0.5;
```

```
>> a=a+1;
```

```
>> b=1; c=a+b;
```

- En MATLAB no es necesario declarar previamente las variables. Por defecto se crean de tipo double (números reales en doble precisión).
- En una asignación el contenido de la derecha del = (que debe ser una expresión evaluable) se asigna a la variable de la izquierda.
- Obviamente, pese a usar el mismo símbolo (=) no supone una identidad matemática (un comando como `a=a+1` no tendría sentido).
- Si quitamos el punto y coma (;) al final de una asignación, el valor final de la variable se vuelca por pantalla. Esto es útil al principio para verificar nuestros resultados.


Espacio de trabajo (workspace)

- El workspace de MATLAB es el conjunto de las variables visibles en un momento dado.
- La aplicación **workspace** es una ventana que muestra el nombre, tipo de datos (entero, double, etc.) y tamaño de las variables presentes.
- Desde la línea de comandos, la orden **whos** nos ofrece la misma info.
- Los comandos **save/load** permiten guardar/restaurar las variables del workspace a disco. Son útiles si queremos cerrar MATLAB y restaurar el estado en el que estábamos en la próxima sesión.
- Si sólo queremos guardar algunas de las variables, usaremos la orden **save fichero x y z**. Al hacer un **load fichero** recuperaremos las variables x, y, z en nuestro workspace con los valores anteriores.
- En ambos casos MATLAB guarda la info en un formato propio .mat
- El comando **clear** limpia el workspace y elimina TODAS las variables.
- El comando **clear a b c** elimina las variables especificadas.

Historial de comandos (Command history)

- En la ventana de comandos se pueden recuperar los comandos anteriores con las teclas de los cursores: ↑ ↓
- También, todos los comandos introducidos en la ventana principal se almacenan en un buffer y se pueden consultar en la ventana del historial de ordenes.
- De dicha ventana podemos seleccionar un grupo de ordenes y con click en el botón derecho del ratón elegir entre:
 - Volver a ejecutar las ordenes.
 - Copiar, para luego pegar en la ventana de comandos o en un programa.
 - Crear un programa (script) a partir de dichas ordenes.

Editor de Matlab

- Se invoca con [edit](#) desde la línea de comandos y es un editor de texto que nos permite crear o modificar programas ya existentes.
- Los programas MATLAB llevan la extensión .m y se llaman m-files.
- Iremos construyendo nuestros programas poco a poco, verificando cada pocas líneas que van haciendo lo que queremos.
- Para ejecutar un programa basta teclear su nombre (+ Enter) en la línea de comandos o pulsar el botón  (Run) desde el editor.
- Además de comandos en nuestros programas podemos añadir comentarios. Cualquier texto tecleado después del símbolo (%) será ignorado.
- El editor está especialmente integrado con MATLAB, utilizando un código de colores para distinguir entre comentarios y texto, palabras claves y variables, etc.

Ayuda en MATLAB

- La ayuda en Matlab es muy, muy completa.
- Usando [helpdesk](#) o [doc](#) abrimos una ventana de ayuda con las típicas opciones de un índice, búsqueda de palabras claves, y una tabla de contenidos.
- También puede pedirse información sobre el uso de funciones individuales desde la línea de comandos usando p.e `>> help cos`
- Más allá del nivel básico, otro aspecto importante de MATLAB es la existencia de programas de demostración. Pueden invocarse desde [helpdesk](#) o directamente desde la línea de comandos con [demos](#).
- Dichos programas, organizados en temas permiten obtener una idea clara de las potencialidades de matlab en las diferentes áreas.
- Consultando el código fuente de las demos, podemos aprender como funcionan.

Organización de datos en MATLAB

- Cualquiera que sea el tipo de datos a usar (enteros, reales, caracteres, o tipos más complejos como estructuras o punteros), MATLAB siempre los almacena en matrices o tablas (arrays).
- Para MATLAB un número escalar es una matriz 1x1, un vector fila es una matriz 1xN, un vector columna, una matriz Nx1, una imagen en blanco y negro, una matriz NxM, una imagen RGB una matriz de NxMX3 (alto, ancho y los tres canales de color), etc.
- MATLAB deriva de MATrix LABoratory, indicándonos que en su origen MATLAB era fundamentalmente una herramienta de Álgebra Lineal numérica (siendo muy potente en ese campo). Luego se le han añadido muchas otras funcionalidades, pero la organización de datos sigue ligada a su pasado matricial.
- Veremos a continuación los operadores básicos para crear, modificar y acceder a matrices. En los ejemplos trabajaremos con números, pero los operadores nos serán válidos independientemente del tipo de datos que usemos.

Creación de matrices en MATLAB

- Una matriz es un contenedor de datos de varias dimensiones, a cuyos elementos accederemos a través de subíndices.
- El operador fundamental para crear una matriz es []

```
>> A=[1 2 3 ; 4 5 6; 7 8 9; 10 11 12];    % Matriz 4x3
>> B=['a' 'b' 'c' 'd']                    % Matriz 1x4
>> C=[1; 2; 3; 4; 5];                     % Matriz 5x1
>> D=7;                                    % Matriz 1x1
>> Q=[];                                   % Matriz vacía
```

- El símbolo ; dentro de [] indica un salto de fila.
- Para crear vectores columna podemos usar múltiples saltos de línea o el operador de transponer matrices ' :

```
>> C=[1; 2; 3; 4; 5];                     % Matriz 5x1
>> C=[1 2 3 4 5]';                         % Equivalente
```

Funciones especiales para crear matrices

- Para ciertas matrices típicas existen funciones especiales:

```
>> A = ones(2,3)      % Matriz 2x3 llena de unos
>> B = zeros(4,2);    % Matriz 4x2 llena de ceros
>> C = zeros(5);      % Matriz 5x5 con ceros
>> D = rand(2,3);     % Matriz de aleatorios (uniforme [0 1])
>> E = randn(N,M);    % Matriz de aleatorios (distribución normal)
>> F = eye(5);        % Matriz identidad tamaño 5x5
>> v=[1 2 3 4]; H=diag(v); % Matriz diagonal a partir del vector v
```

- Las funciones `size`, `length`, `ndims`, `numel` nos permiten saber detalles como número de dimensiones, tamaño, número de elementos, etc:

```
>> X = rand(1,5);
>> size(X), ans = 1 5 >> size(X'), ans = 5 1 >> length(X), ans = 5
>> A=ones(2,3);
>> ndims(A), ans=2, >> numel(A), ans=6
>> size(A), ans = 2 3 >> size(A,1), ans = 2, >> size(A,2), ans = 3
```

El operador : para la creación de tablas

- El operador (:) permite crear vectores con datos equiespaciados:

$x = (\text{inicio} : \text{incremento} : \text{final})$

- Si no se especifica el incremento, se entiende que es la unidad.

```
>> x=(0:6);           % Equivalente a x=[0 1 2 3 4 5 6];
>> x=(0:2:6);         % Equivalente a x=[0 2 4 6];
```

- El incremento pueden ser valores negativos o fraccionarios:

```
>> y = (10:-2:0);     % Equivalente a y = [10 8 6 4 2 0];
>> x=(0:0.1:pi);      % Crea vector desde 0 a pi con salto 0.1
>> y=sin(x);          % Calcula seno de todo el vector anterior
>> plot(x,y)           % Muestra resultados de forma gráfica
```

- **IMPORTANTE:** la mayoría de las funciones de MATLAB pueden trabajar de forma vectorial: si reciben un vector o matriz aplican la función a todos los elementos de la matriz, ahorrándonos bucles.

Concatenación de matrices

- El operador [], además de crear matrices puede usarse para concatenar matrices ya existentes, generando matrices mayores.
- La única restricción es que las dimensiones sean compatibles

```
>> A = ones(2,4); B=zeros(2,4); C=ones(2,2);  
>> D = [A B]; % Matriz resultante es 2 x 8  
>> D = [A; B]; % Matriz resultante es 4 x 4  
>> D = [A C]; % Matriz 2 x 6  
>> D = [A; C]; % Error, dimensiones no casan  
>> a='hola '; b = 'que tal'; c=[a b], c = hola que tal
```

- En el último ejemplo se ve que para MATLAB una cadena de texto es como una matriz (vector) de caracteres, pudiéndose concatenar.
- Una alternativa más avanzada para concatenar matrices es usar funciones específicas como `cat`, `vertcat`, `horzcat`, `repmat`, etc.

Indexado de matrices o tablas

- Para acceder a los elementos individuales de una matriz se usa una notación similar a la matemática con subíndices:

```
>> A=[1 2 3 ; 4 5 6; 7 8 9; 10 11 12]; % Matriz 4x3  
>> A(1,2) % Primera fila, segunda columna (2)  
>> A(3,3) % Tercera fila, tercera columna (9)
```

- La notación de MATLAB permite acceder a varios elementos simultáneamente, usando una lista de índices:

```
>> A=[1 2 3; 4 5 6; 7 8 9; 10 11 12]; % Matriz 4x3  
>> A(1,[1 2]) % Primera fila, elementos 1 y 2  
>> A([1 3],3) % Tercer elemento (columna) de filas 1 y 3  
>> A(1,:) % Todos (:) los elementos de la fila 1  
>> A(:,2) % Todos (:) los elementos de la columna 2  
>> A([1:3],[1:2]) % Submatriz 3x2 de la matriz original  
>> A([4 2],[1 3]) % Elementos 1 y 3 de las filas 4 y 2  
>> A(2:end,3) % Elementos desde 2 al último de la columna 3
```


Reserva previa de memoria

- MATLAB no requiere declaración ni reserva previa de memoria pero si se conoce el tamaño final de una matriz es mejor crearla previamente y luego “llenar” sus casillas en vez de ir añadiendo datos (y hacer crecer la matriz) sobre la marcha.
- La razón es que cada vez que añadimos datos MATLAB debe reservar memoria para la nueva matriz resultante (más grande). Si esto se repite mucho es muy costoso computacionalmente:

```
>> for k=1:10000, x(k)=k; end % Crea vector de 1 a 10000  
>> x=zeros(1,10000); for k=1:10000, x(k)=k; end  
>> x=(1:10000);
```

Opción 1) aumenta el tamaño del vector x a cada paso del bucle.

Opción 2) usa un bucle, pero reserva previamente la memoria.

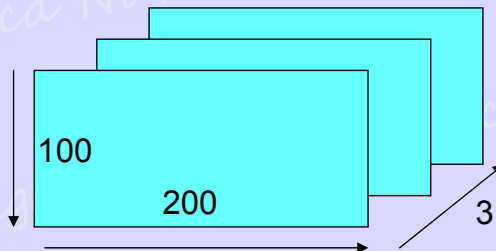
Opción 3) usa una sola orden, x=(inicio:fin), sin usar un bucle for.

Las tres líneas anteriores son equivalentes pero hay diferencias significativas de tiempo (medidas con `tic`, `toc`) entre 1 y 3.

Matrices multidimensionales

- No estamos limitados a matrices 2D, pudiendo usar más dimensiones, que se indexarán con 3, 4, ... índices.

```
>> im=zeros(100,200,3);  
% Tabla 3D de 100 x 200 x 3
```

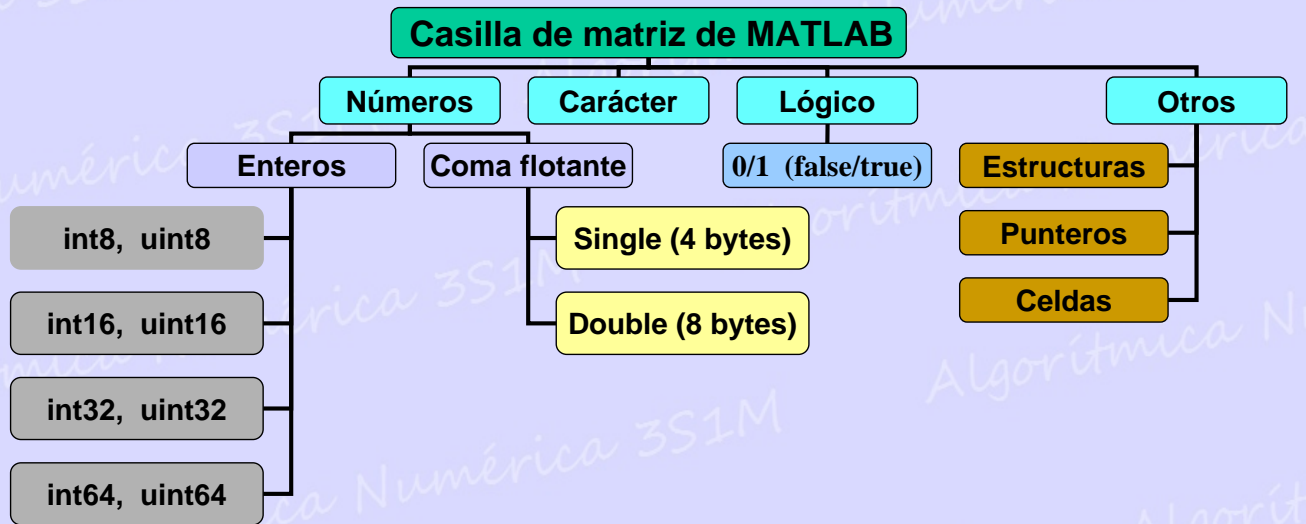


- La matriz anterior podría ser el contenedor de una imagen RGB, de alto 100 píxeles, ancho 200, y los tres canales (R, G, B) de color.
- El indexado es similar al que hemos visto anteriormente:

```
>> a=im(1:50,1:50,:); % Subimagen 50x50  
>> a=im(:, :, 2); % Componente verde(2) de TODA la imagen  
>> a=im(1:2:end,1:2:end,:); % imagen reducida en factor 2  
>> a=im(:, :, [2 3 1]); % Imagen con los planos de color permutados
```

Tipos de datos en MATLAB

- ¿Qué tipos de datos podemos guardar en las casillas de un array?
- En un principio MATLAB sólo trabajaba con reales (double), pero actualmente se ha ampliado a otros muchos otros tipos.



Números reales (coma flotante)

- Los números reales se guardan en coma flotante (notación científica), como un signo, una mantisa y un exponente (base 2).

Tipo	Tamaño	Mant.	Exp.	Signo	Precisión	Rango
single	4 bytes (32 bits)	23 bits	8 bits	1 bit	6/7 cifras	10^{38}
double	8 bytes (64 bits)	52 bits	11 bits	1 bit	15/16 cifras	10^{300}

- Por defecto MATLAB usa datos de tipo double (máxima precisión), aunque pueden convertirse a single usando la función `single()`.
- Es importante entender que toda operación con números en coma flotante (incluso su simple asignación) conlleva un error, ya que no todos los números reales pueden expresarse con números máquina.
- La función `eps(x)` nos da la separación entre números máquina para los valores alrededor de x .
- A entender este problema y sus consecuencias dedicaremos el 1er tema de la asignatura.

Números enteros

- Podemos usar varios tipos de datos enteros, dependiendo del espacio reservado en memoria, lo que determina su posible rango.

Tipo	Tamaño	Signo	Rango
uint8	1 byte	+	$[0, 2^8)$ [0-255]
int8	1 byte	-/+	$[-(2^7), 2^7)$ [-128,127]
uint16	2 bytes	+	$[0, 2^{16})$ [0-65535]
int16	2 bytes	-/+	$[-(2^{15}), 2^{15})$ [-32768,32767]
uint32	4 bytes	+	$[0, 2^{32})$
int32	4 bytes	-/+	$[-(2^{31}), 2^{31})$
uint64	8 bytes	+	$[0, 2^{64})$

- La elección del tipo depende del rango de los datos a manejar.
- Las funciones `intmax()`, `intmin()` nos dan los límites de un tipo:

```
>> intmin('int8'), ans = -128
>> intmax('uint32'), ans = 4294967295
```

Tipo booleano (lógico)

- Sólo tomar dos valores true (1) o false (0), aunque al asignarse se puede usar true o false, para una mejor lectura:

```
>> a = true, b = false, a = 1, b = 0
```

- Aparecen en el resultado de comparaciones (`>`, `>=`, `<`, `<=`, `==`, `~=`) o de las operaciones lógicas AND (operador `&&`) o OR (operador `||`).
- Se usan como condiciones para controlar el flujo del programa (en los comandos **if then**, o **while**) o para el **indexado lógico** dentro de una matriz o vector.

```
>> A=randn(1,1000); % Vector de números aleatorios
>> positivos=(A>=0); % Vector con valores de tipo lógico, 1 si se
                    % cumple la condición dada, 0 si no se cumple.

>> B=A(positivos); % Extracción de valores positivos
>> B=A(A>=0); % Equivalente.
```

Indexado lógico de tablas y matrices

- MATLAB permite indexar una matriz o vector en función de un resultado lógico, lo que permite asignaciones muy compactas.
- Si hacemos una comparación (p.e. mayor que) entre una matriz de MATLAB y otra obtenemos una matriz de elementos lógicos (1 / 0) que nos indica si la condición se cumple (1) o no (0).
- Dicha matriz lógica puede usarse como índice para extraer o operar sólo con aquellos elementos que verifiquen la condición dada:

```
>> x = rand(1,1000);           % Crea vector de 1000 aleatorios entre 0 y 1
>> x(x>0.5) = 1;               % Pone a 1 aquellos que superen el valor 0.5
>> x(x<=0.5) = 0;              % Pone a 0 aquellos que quedan por debajo

>> a = rand(50); b = rand(50); % Crea dos matrices aleatorias 50x50
>> a(a>b) = b(a>b); % Los elementos de a mayores que el correspondiente
                    % elemento de b (condición a>b) son substituidos por
                    % el correspondiente elemento de b.
```

Indexado lógico de tablas y matrices

- En el cuadro siguiente podemos apreciar el estilo más compacto que nos proporciona el indexado lógico.

```
a(a>b) = b(a>b); % Usando indexado lógico y enfoque vectorial
for k=1:50, for j=1:50, % Usando bucles de forma más tradicional
    if a(k,j)>b(k,j), a(k,j)=b(k,j); end
end; end;
```

- Además de operar con un subconjunto de elementos de la matriz, el indexado lógico nos permite extraer aquellos elementos que verifican cierta condición:

```
>> x = rand(1,1000);           % Crea vector de 1000 aleatorios entre 0 y 1
>> y = x(x>0.75)               % Nuevo vector con los elementos de x > 0.75
>> length(y)                   % Número de elementos por encima de 0.75.
                                % Deberían ser unos 250 (1000/4)

>> find (x>0.75)               % Índices (en x) de los valores > 0.75
```


Programación en Matlab

- Un programa (script) de MATLAB no es más que una sucesión de ordenes (sentencias) guardadas en un fichero con extensión .m
- Al teclear el nombre del fichero (lo que en MATLAB se conoce como un script) todas sus sentencias se ejecutan secuencialmente.
- Un script es la forma más habitual de escribir el código solución de un ejercicio (salvo en ejercicios muy sencillos, como los del LAB1)
- En esta sección presentaremos los principales elementos que nos permitirán construir nuestros primeros programas en matlab:

1. Variables (etiquetas que permiten acceder a los datos)
2. Entrada / salida en la ventana de comandos.
3. Operadores (aritméticos, lógicos, comparaciones)
4. Sentencias de control de flujo del programa: condiciones y bucles

Variables

- Las variables en cualquier lenguaje de programación son etiquetas que se asignan a un dato (o tabla de datos) y nos permiten acceder a su contenido, operar con él, modificarlo, etc.
- En MATLAB no necesitamos una declaración previa al principio del programa de las variables que vamos a usar. En cualquier momento podemos inicializar una nueva variable que precisemos simplemente asignándole un valor.
- Si intentamos usar una variable no inicializada, MATLAB da error.
- Las variables creadas dentro de la ventana de comandos pueden observarse en el workspace y están siempre disponibles a menos que se eliminen con el comando **clear**:

```
>> x=2; y=3; z=[1:10]
>> clear z;    % Elimina variable z
>> clear      % Elimina todas las variables
```

Nombres de variables

- Los nombres de las variables pueden ser de hasta 63 caracteres y se distingue entre mayúsculas/minúsculas.
- Hay que evitar variables con el nombre de funciones ya existentes. Al hacer `>> sin=3;` creamos una variable llamada sin y la función sin() deja de funcionar. Para arreglarlo haremos `>> clear sin`
- También existen algunos nombres reservados que debemos evitar:

```
pi    -> valor de pi (3.141592...)
ans   -> matlab guarda los resultados no asignados en ans
computer, version -> variables con info sobre sistema
```

- Superando nuestra pereza debemos tratar de dar nombres auto-explicativos a las variables:
- Supone una gran ayuda para que alguien pueda seguir y entender nuestro programa. Lo más probable es que ese alguien seáis vosotros mismos una semana después intentando modificarlo.

Operadores

- Una vez asignados o introducidos los datos en variables el siguiente paso en un programa es operar con ellos.
- Un operador actúa sobre uno o más operandos (variables) y genera un resultado que suele ser asignado a una variable distinta (o a alguno de las variables iniciales, que es actualizada):

```
>> x = rand(1,10); y = rand(1,10);
>> z = x+y; % Asignacion a una tercera variable
>> x = x+y; % Actualización de uno de las variables
```

- Existen tres clases fundamentales de operadores en MATLAB:

Aritméticos: operan sobre números (suma, multiplicación, etc.)

Comparaciones: mayor que, menor que, etc.

Sus resultados son variables lógicas.

Lógicos: operan sobre variables lógicas (AND, OR, etc)

Operadores aritméticos

- Operan con números (o arrays) y el resultado son más números.
- Los símbolos son los comúnmente utilizados: +, -, *, /, ^, etc
- En MATLAB es fundamental distinguir entre operadores matriciales (que siguen las reglas del álgebra lineal) y operadores punto a punto (que operan sobre matrices o arrays elemento por elemento).
- Para diferenciar los operadores punto a punto se les añade un punto . delante del operador (por ejemplo .* frente a *)
- Dadas dos matrices A, B, $C=A*B$ es la matriz producto, con sus elementos definidos por:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

- Si usamos el correspondiente operador punto a punto (.*), la matriz resultante $C=A.*B$ tiene por elementos:

$$C_{ij} = A_{ij} \times B_{ij}$$

Operadores matriciales vs. punto a punto

- Al operar con operadores matriciales se deben respetar las reglas del álgebra en las dimensiones. En la multiplicación matricial $C=A*B$, el número de columnas de A debe ser igual al número de filas de B y la matriz resultante C tendrá las filas de A y las columnas de B.
- En cambio si hacemos $C=A.*B$, ambas matrices A y B deben ser de iguales dimensiones (para poder multiplicar cada elemento de A con el correspondiente de B). Las dimensiones de C serán las mismas.
- Igualmente, si se usa la multiplicación $a*b$ con vectores, a puede ser un vector fila (1 x N) y b un vector columna b (Nx1). El resultado será un escalar (el producto escalar de los vectores a y b):

$$c = \sum_{k=1}^N a_k \cdot b_k$$

- Si intentamos multiplicar (*) dos vectores fila MATLAB da un error. Si los multiplicamos usando .* obtendremos un tercer vector fila del mismo tamaño con componentes:

$$c_k = a_k \cdot b_k$$

Operadores matriciales vs. punto a punto

- En algunos casos (+ / -) la suma de matrices coincide con la suma elemento a elemento, por lo que no existen los operadores (.+) ni (-.).

$A + B$	Suma de dos matrices $n \times m$ (elemento a elemento)
$A - B$	Resta de matrices $n \times m$ (elemento a elemento)
$-A$	Cambio de signo de A (de todos sus elementos)
$A * B$	Multiplicación matricial $(n \times k) * (k \times m) \rightarrow n \times m$
$A .* B$	Multiplicación punto a punto $(n \times m) .* (n \times m) \rightarrow n \times m$
A / B	“Division” matricial. En general equivale a $A * \text{inversa}(B)$
$A ./ B$	División punto a punto $A(i,j)/B(i,j)$
$A ^ k$	Potencia k -ésima de A . Equivale a $A * A * \dots * A$ k veces.
$A .^k$	Eleva a la potencia k cada elemento de A , $A(i,j)^k$
$A '$	Traspuesta de A (traspuesta conjugada para complejos)

Operando con matrices de distintas dimensiones

- Al hacer una operación entre una matriz y un escalar se entiende que se hace una operación punto a punto entre ese escalar y cada uno de los elementos de la matriz.
- Si A es una matriz $n \times m$ y c un escalar, $A+c$ es la matriz $n \times m$ que resulta de sumar c a todos los elementos de A . Igualmente, $A*c$ o A/c son matrices $n \times m$ resultado de multiplicar (o dividir) cada elemento de A por el escalar c .
- En las últimas versiones de MATLAB esto se ha generalizado. Al operar una matriz A ($n \times m$) con un vector fila x ($1 \times m$), se entiende que queremos operar el mismo vector x con cada una de las filas de A :

```
>> A=[1 2 3 4;  
      5 6 7 8];  
>> x=[2 2 2 2]
```

```
>> A+x,      3      4      5      6      >> A.*x,      2      4      6      8  
           7      8      9     10           10     12     14     16
```


Operadores de comparación

- MATLAB proporciona los siguientes operadores de comparación:

Operador	Descripción	Operador	Descripción
>	Mayor que	<=	Menor o igual que
>=	Mayor o igual que	==	Igual que
<	Menor que	~=	Distinto de

- El resultado de una comparación es un valor lógico (1 si se cumple o 0 si no se cumple). Si se opera con arrays, ambos deben ser del mismo tamaño y el resultado será un array de 0's o 1's lógicos resultado de las comparaciones individuales de todos los elementos.
- La excepción a lo anterior es la comparación de una tabla con un escalar, donde se comparan todos los elementos con dicho escalar.

```
>> x=(1:5); y = (5:-1:1); % x=[1 2 3 4 5]; y=[5 4 3 2 1];  
>> (x<=y), ans = 1 1 1 0 0  
>> (x~=3), ans = 1 1 0 1 1
```

Operadores lógicos

- Realizan las típicas operaciones lógicas AND, OR, NOT, etc., usadas para combinar varias condiciones y determinar si todas se han cumplido (con un AND), si al menos una se cumple (OR), etc.

Operación	Símbolo	Descripción
AND	a && b	TRUE sólo si a y b son TRUE
OR	a b	TRUE si a o b son TRUE
NOT	~ a	TRUE si a es FALSE y viceversa.
XOR	xor(a,b)	TRUE sólo si una entrada lo es y la otra no

- Suelen usarse como condiciones en los operadores de control de flujo que veremos más adelante (if, while).

Precedencia de operadores

- Si una expresión en MATLAB es compleja (muchos operadores) debemos tener en cuenta la precedencia entre dichos operadores.

Por ejemplo $5*3/2*5$ se evalúa como $5*(3/2)*5$ y no como $(5*3)/(2*5)$.

- En los manuales de MATLAB podéis encontrar detalles sobre el orden en que MATLAB evalúa una expresión que combine diversos operadores (incluso mezclando operadores aritméticos, lógicos, etc)
- Sin embargo, la recomendación es no complicar las expresiones, evaluando las diferentes partes por separado y combinándolas posteriormente.
- Ante cualquier duda, usar liberalmente los PARÉNTESIS.
- Los paréntesis tienen la más alta precedencia en Matlab (siempre se evalúa su contenido antes de seguir operando).

Salida de datos (fprintf)

- Omitiendo el punto y coma MATLAB vuelca el contenido de una variable a la ventana de comandos. El formato de esta salida (nº de decimales, notación científica, etc) se cambia con el comando `format`.
- Para un control más preciso de la salida se usa el comando `fprintf`, que nos permite además mezclar valores de variables (de todo tipo) y un texto descriptivo en la salida.
- Los argumentos básicos `fprintf` consisten en una cadena de texto de formato y la lista de variables a volcar.

```
>> fprintf(formato, var1,var2,var3,...)
```

- La cadena formato se delimita con comillas simples e incluye texto descriptivo junto con instrucciones (precedidas por %) sobre como formatear cada variable.
- En la salida los valores de las variables (cada una con un formato propio) se intercalan con el texto descriptivo apareciendo en los lugares donde insertamos su descripción de formato.

Formatos básicos de fprintf

Comando	Descripción	Comando	Descripción
\n	Salto de línea	\t	Inserta espacio (tab)
%d, %i	Volcar entero	%c	Volcar un solo carácter
%x, %X	Formato hexadecimal	%s	Cadena de texto
%f	Volcar double	%e, %E	Notación científica

- Una misma variable puede formatearse con diferentes descriptores:

```
>> a=65;
>> fprintf('INT %d HEX %x CHAR %c FLOAT %f EXP %e\n',a,a,a,a,a)
INT 65 HEX 41 CHAR A FLOAT 65.000000 EXP 6.500000e+001
```

- Lo normal es usar tantos descriptores (%) como variables (5)
- El texto intercalado entre descriptores (%) se vuelca sin cambios.
- También podríamos haber usado un array [a a a a a] de cinco datos:

```
>> fprintf('INT %d HEX %x CHAR %c FLOAT %f EXP %e\n',a*ones(1,5))
```

Modificadores en los formatos básicos

- Los formatos básicos pueden modificarse fácilmente:

Comando	Descripción
%5d	Entero con 5 cifras (o espacios hasta completar 5 columnas)
%05d	Entero con cinco cifras insertando ceros delante
%7.3f	Double con siete espacios, 3 de ellos reservados a decimales.
%+10.5f	Double en 10 columnas, volcando signo y 5 decimales.
%12.3e	Double en 12 columnas, notación científica, 3 decimales en mantisa.
%04X	Formato hexadecimal en 4 columnas, rellenando con ceros

- Si hay más descriptores (%) que variables los descriptores de sobra no provocan ninguna salida. Si hay más variables que descriptores el comando se repite hasta agotar las variables especificadas:

```
>> fprintf(' %d ',[1 2 3 4 5])
1 2 3 4 5 >>
```

Sentencias de control de flujo en el programa

- Una parte fundamental de cualquier programa es la posibilidad de actuar de forma distinta ante diferentes condiciones.
- Se entiende por sentencias de control de flujo aquellas que permiten ejecutar cierto código en función de una condición, repiten unas instrucciones muchas veces (bucle), terminan el programa, etc.
- En MATLAB destacaremos:
 1. Saltos condicionales: `if then else` , `switch case`
 2. Bucles: `for`, `while`, `continue`, `break`
 3. Terminación de un programa: `return`

Condiciones (if, if else)

- La sentencia `if` evalúa una expresión lógica y ejecuta bloques de código basándose en su resultado. Debe terminarse con un `end`

```
if (expresión lógica)
    código;
end

if (expresión lógica)
    código 1;
else
    código 2;
end
```

- En el primer caso si la expresión es cierta (true ó 1) se ejecuta el código y si no lo es (false ó 0) se salta. En el segundo se ejecuta el código 1 si la expresión es cierta y el código 2 si resulta ser falsa.

```
if mod(a,2)==0, % Si el resto de a/2 es cero
    fprintf('%d es par\n',a);
else
    fprintf('%d es impar',a);
end
```


Condiciones con múltiples casos (switch)

- La sentencia `if else end` es adecuada en una decisión binaria. Si hay que ejecutar acciones distintas en función de que una variable valga p.e 1, 2, ó 3 el uso de `if else` se hace confuso, siendo preferible usar la construcción `switch, case, end`:

CÓDIGO IF ELSE END

```
if (opt==1), COD1;
else
    if (opt==2), COD2;
    else
        if (opt==3), COD3;
        else
            OTROS CASOS;
        end
    end
end
```

CÓDIGO SWITCH CASE END

```
switch (opt)
case 1: COD1;
case 2: COD2;
case 3: COD3;
otherwise: OTROS CASOS;
end
```

A cada alternativa (case) le sigue el código correspondiente. El código siguiendo a otherwise (opcional) solo se ejecuta si no se verifica ninguna de las condiciones listadas previamente. .

La variable opt usada en el switch() puede ser un escalar o una cadena de texto.

Iteraciones y bucles

- Los bucles permiten repetir una tarea un número determinado de veces o hasta que consigamos que se cumpla una cierta condición.
- En MATLAB, para el primer caso se usa la sentencia `for`, que nos permite dar una **lista de valores** a un índice y un **fragmento de código** que se ejecutará para los valores especificados del índice:

```
for k = lista
    CODIGO;
end
```

Típicamente lista es un vector equiespaciado expresado como (inicio : inc : final). k varía desde inicio hasta fin con salto inc. Si se omite el incremento (ini : fin) se usa la unidad.

Aproximación del número e sumando N términos de una serie:
$$e \approx \sum_{k=0}^N \frac{1}{k!}$$

```
>> temp=1; e_aprox=1; N=50;
>> for k=1:N, temp=temp/k; e_aprox = e_aprox + temp; end
>> e=exp(1); dif = e-e_aprox,
    dif = 0.0000
```

Bucles FOR

- La lista barrida por k puede ser cualquier vector, no precisándose que sea equiespaciado ni monótono. Por ejemplo podríamos tener for k=[1 -4 8 1], donde k tomaría los valores 1, -4, 8, y 1 de nuevo.
- Podemos anidar bucles, teniendo en cuenta de que para cada valor del índice del bucle más exterior se ejecutarán todos los valores del índice del bucle anidado:

```
for k=1:1000, for l=1:1000,  
    A(k,l) = A(k,l) - 5; % Por aquí se pasa un millón de veces  
end, end
```

Conviene tener en cuenta que en muchos casos nuestro programa puede optimizarse eliminando bucles FOR, al aprovechar el carácter vectorial de MATLAB.

Por ejemplo, el bucle anterior puede cambiarse por A=A-5;

Bucles WHILE

- Un bucle WHILE nos permite repetir cierto código hasta que una condición lógica se verifique:

```
while (condicion),  
    CODIGO;  
end
```

Las sentencias comprendidas entre el while y end se repiten mientras la condición siga evaluándose como cierta.

- Si la condición es falsa la primera vez, el código dentro del bucle no se llega a ejecutar nunca.
- El código siguiente es una mejora en la aproximación al número e evitando operaciones innecesarias:

```
temp=1; e_aprox=1; k=1;  
while (temp>1e-16)  
    temp=temp/k;  
    e_aprox = e_aprox + temp;  
    k=k+1;  
end
```

Saltando pasos y saliendo de un bucle

- La sentencia **continue** nos permite saltar una iteración dentro de un bucle FOR. En cuanto aparece se deja de ejecutar la iteración en la que nos encontramos y se pasa a la siguiente.
- La sentencia **break** (dentro de un bucle FOR o de un WHILE) hace que abandonemos el bucle y no se ejecuten las iteraciones que faltan.

```
for k = 1:10,
```

```
    if k==7, continue; end
```

```
    disp(k^2)
```

```
end
```

```
for k = 1:10,
```

```
    if k==7, break; end
```

```
    disp(k^2)
```

```
end
```

- El código de la izquierda vuelca los cuadrados de los números del 1 al 10, pero se salta el del 7 debido al **continue**.
- El código de la derecha vuelca los cuadrados del 1 al 6, ya que al llevar al **break** salimos y no seguimos con el bucle.
- En el caso de bucles anidados un **break** no sale de todos ellos, sólo devuelve el control al bucle inmediatamente por encima.

Programas en MATLAB

- Una vez conocidos los bloques básicos de MATLAB solo queda combinarlos en una estructura superior (programa).
- En su forma más simple un programa de MATLAB es una sucesión de ordenes (sentencias) guardadas en un fichero con extensión .m
- Al teclear el nombre del fichero en la ventana de comandos las sentencias se ejecutan de forma secuencial.
- En principio, cualquier programa, por complicado que sea podría escribirse en un único fichero.
- Sin embargo, una buena práctica de programación aconseja subdividir el programa en diversos módulos
- Cada uno de dichos módulos cumple una función determinada y se debe guardar en ficheros .m separados.
- Hay dos tipos principales de ficheros .m : scripts y funciones.

Scripts

- Un fichero de tipo script (o macro) es una simple lista de comandos.
- Su efecto es idéntico a insertar los comandos que contiene el script en el punto del programa donde le llamamos:

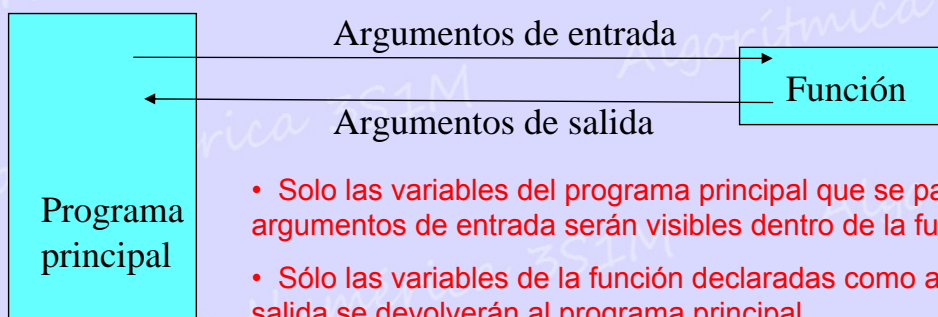
```
%% Fichero script llamado contraste.m  
t1=(a+b); t2=abs(a-b);  
c=t2/t1;
```

`a=4;b=5;
contraste
disp(c)` → `a=4;b=5;
t1=(a+b); t2=abs(a-b);
c=t2/t1;
disp(c)`

- Un script **comparte variables** con el programa que le llama (se ven las que había y las que se crean se añaden al workspace): en este caso se precisa que existan las dos variables a y b antes de llamar al script, y, al terminar, aparecen variables adicionales (c, t1 y t2)
- No acepta argumentos de entrada (solo trabajan con a y b) ni devuelve argumentos de salida (el resultado siempre está en c).

Funciones en MATLAB

- Los scripts serán la opción más habitual para escribir el código que resuelva un ejercicio de laboratorio o de un examen.
- Sin embargo, debido a que comparten las variables con el programa que les llama los scripts no son una buena opción para programas complejos, donde se trata de modularizar el código.
- Las funciones en MATLAB tienen su propio espacio de variables y no ven las variables del programa que las llama. La única forma de relacionarse es a través de los argumentos de entrada y salida.



- Solo las variables del programa principal que se pasen como argumentos de entrada serán visibles dentro de la función.
- Sólo las variables de la función declaradas como argumentos de salida se devolverán al programa principal.

Funciones en MATLAB

- Repitamos el ejemplo anterior usando una función en vez de un script

```
function c = contraste(a,b)
% Fichero .m con una función llamada contraste.m
t1=(a+b); t2=abs(a-b); c=t2/t1;
return;
```

% Llamada desde el programa principal (o la ventana de comandos)

```
M=5; m=2; q=contraste(m,M);
```

- a y b representan los argumentos de entrada en la función, pero podemos pasar cualquier par de variables desde el programa principal: la función hará con ellas lo que haga con a y b en su definición.
- El valor del argumento de salida (c) se asignará a la variable que se especifique en el programa principal (q).
- Las variables usadas internamente t1, t2, c dejan de existir una vez que se termina de ejecutar la función.

Definición de una función MATLAB

- Veamos en detalle los elementos de un fichero .m que contenga la definición de una función

Keyword que indica que es una función MATLAB

```
function f = fact(n)
```

Argumento/s salida (pointing to 'f')

Nombre función (pointing to 'fact')

Argumento/s entrada (entre paréntesis) (pointing to '(n)')

```
% Calcula el factorial de n: n!
```

Comentarios explicativos sobre uso de la función, argumentos entrada/salida, etc. Aparecen al hacer un help fact

```
f=1;
```

```
for k=1:n, f=f*k; end
```

Núcleo de la función:

Aquí se resuelve el problema dado. Es fundamental que durante el proceso se asignen valores a la variable/s declarada como argumentos de salida (en este caso f)

```
return
```

Keyword indicando el fin del código de la función.

Ventajas del uso de funciones

- Con el uso de funciones podemos dividir nuestro programa en módulos “estancos”, que sólo compartan la mínima información necesaria: argumentos de entrada y salida.
- Permite dividir el problema en tareas sencillas con objetivos claros: por ejemplo, recibir datos (x, y) para calcular un cierto resultado z.
- El cómo codificar dicha función no es un problema que nos preocupe al diseñar el programa principal. Puede dejarse para luego o incluso encargárselo a otra persona. No hay peligro de que las variables intermedias usadas interfieran con las que yo uso en el principal.
- Esta es la clave para el desarrollo de programas complejos: subdividirlos en unidades más pequeñas con cometidos más sencillos y con la menor interacción posible entre ellos: unos pocos argumentos de entrada y de salida.
- Las unidades sencillas (funciones) son más fáciles de verificar.

Uso de scripts/funciones en ESTE curso

En general los problemas planteados son los suficientemente sencillos para que normalmente puedan ser resueltos en un script.

Si es conveniente (o quiero saber si sabéis hacerlo) escribir una función para hacer una tarea específica se indicará en el ejercicio.

La idea general será escribir un script por cada ejercicio a resolver (ejer1.m, ejer2.m, etc.). En caso de ser necesario se pueden escribir una o varias funciones independientes que luego se llamarán desde el script principal.

Es una buena práctica empezar el script de un nuevo ejercicio con el comando `clear;` Así se borra el *workspace* y os aseguráis de que el código para crear todas las variables necesarias esté incluido. Esto es especialmente importante en una entrega de un ejercicio de examen.

ES MUY IMPORTANTE acostumbrarse a usar el editor para cualquier ejercicio que requiera más de 3 líneas de código. De esta forma tenéis siempre disponible vuestra última versión en vez de tener que buscar los comandos correctos en el historial de comandos.

Argumentos de entrada

- Si hay varios argumentos de entrada se separan por comas.
- Usando `nargin` (nº arg in) podemos saber con cuantos argumentos de entrada se ha hecho la llamada, dando más flexibilidad a las funciones:

```
function x = aleat(N,sigma,m)
% Genera un vector de N aleatorios con desviación sigma y media m
% Si no se especifican sigma=1 y m=0
x=randn(1,N); if nargin==1, return; end
x=x*sigma;    if nargin==2, return; end
x=x+m;
return
```

```
>> a = aleat(100);      % Vector 1x100 con sigma=1 y media 0
>> a = aleat(100,2);    % Vector 1x100 con sigma=2 y media 0
>> a = aleat(100,3,-1) % Vector 1x100 con sigma=3 y media -1
```

- El uso de `nargin` evita realizar trabajo extra si el usuario no lo requiere y permite asignar valores por defecto a los parámetros de entrada.

Argumentos de salida

- Una función puede devolver varios resultados: se especifican como varios argumentos de salida entre corchetes y separados por comas.
- Igualmente, usando `nargout` (num arg out) podemos saber cuantos argumentos de salida nos ha requerido el usuario:

```
function [m,s]= stat(x)
% Calcula media m y (opcionalmente) varianza s de un vector x
m=mean(x);
if nargout==2, s=std(x); end
return
```

```
>> m = stat(x);      % Calcula la media de x
>> [m s] = stat(x); % Calcula la media Y la desviación estándar de x
```

- De nuevo, el uso de `nargout` evita que se realice trabajo adicional (cálculo de la desviación standard) si el usuario no lo requiere.

Funciones predefinidas en MATLAB

- Además de las funciones que podamos escribir nosotros MATLAB cuenta con una enorme librería de funciones predefinidas.
- Solo hay que conocer las más sencillas. Si se necesita alguna función especial se recordará en el ejercicio correspondiente.
- **Exponenciales y logarítmicas:** exp(x), log(x), log2(x), log10(x)
- **Trigonómicas:** sin(), cos(), tan(), asin(), acos(), atan(), atan2()
- **Hiperbólicas:** sinh(), cosh(), tanh(), etc.
- **Otras:** valor absoluto (abs), raíz cuadrada (sqrt), redondeo al entero más próximo (round), redondeo al entero inferior (floor) o superior (ceil),

Estas funciones de MATLAB aplicadas a un vector o matriz se aplican a todos sus elementos resultando en un vector/matriz del mismo tamaño.

Funciones predefinidas en MATLAB

Otras funciones como max (máximo), min (mínimo), mean (media), sum (suma), etc. al aplicarse a un vector dan como resultado un único número:

```
>> x=randn(1,1000);  
>> max(x), 3.266    min(x), -2.819    mean(x), 0.029,    std(x), 0.989
```

Al aplicar estas funciones a una matriz trabajan columna por columna, obteniéndose p.e. un vector con el mínimo o máximo de cada columna.

```
>> A = [1 2 3 4;  
        5 6 7 8];  
>> sum(A)→[6 8 10 12]  >> x=mean(A)→[3 4 5 6]  >> min(A) → [1 2 3 4]
```

Para calcular el mínimo, máximo, etc. de TODA la matriz se debe usar:

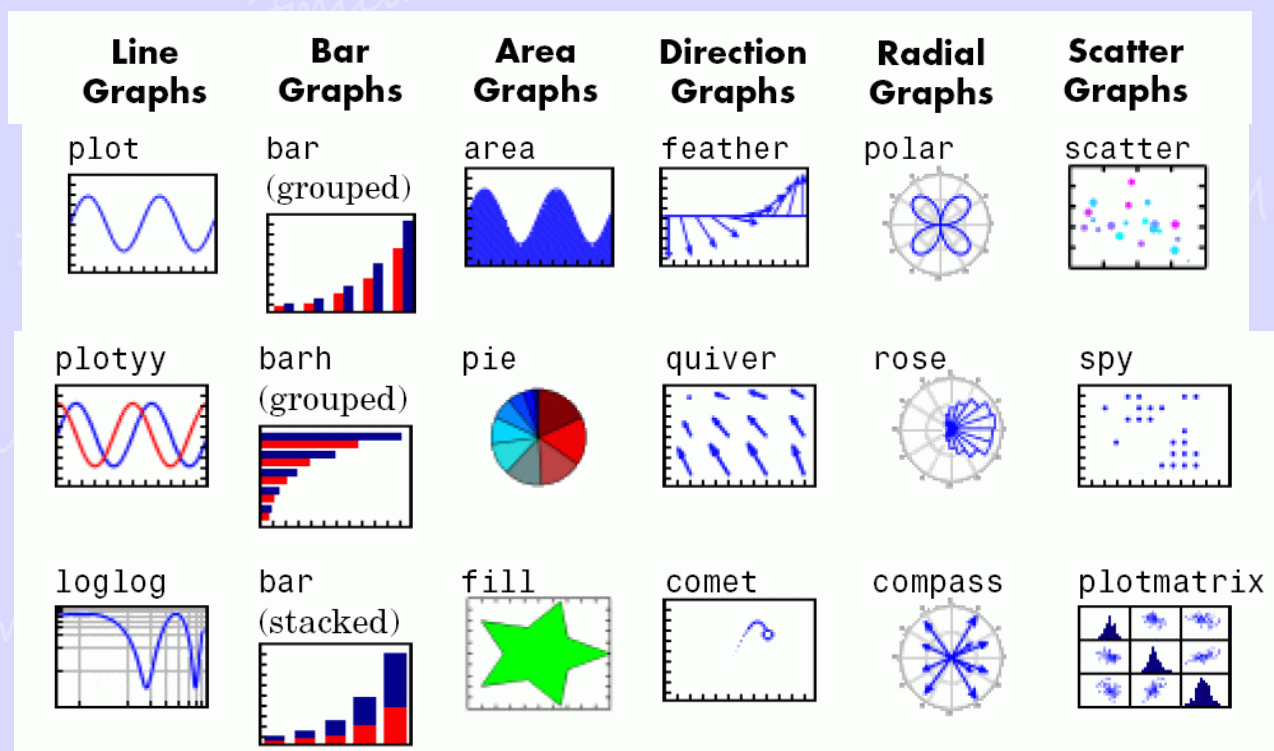
```
>> min(A(:))→ 1  >> max(A(:))→ 8  >> mean(A(:))→ 4.50  >> sum(A(:))→36
```

A(:) reduce una matriz multidimensional a un único vector columna

Gráficos en MATLAB

- MATLAB ofrece unas librerías y herramientas muy potentes para la visualización de datos en varias dimensiones.
- Una de las ventajas de MATLAB es que podemos crear gráficos con un mínimo de esfuerzo usando comandos de “alto nivel” como plot o surf.
- Si queremos podemos también utilizar accesos de bajo nivel que permiten leer/modificar cualquier característica de un gráfico usando las funciones get/set aplicadas al correspondiente puntero (o handle) del objeto gráfico correspondiente
- Para poder aprovechar al máximo las posibilidades de MATLAB hay que tener una idea básica del enfoque orientado a objetos que usa MATLAB en sus aplicaciones gráficas, así como la jerarquía entre los distintos objetos gráficos que existen.
- Pero para este curso NO ES NECESARIO conocer estos detalles.

Ejemplos de gráficos en MATLAB (2D)



Ejemplos de gráficos en MATLAB (3D)

Line Graphs

ezplot3

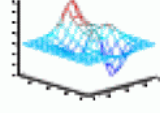


waterfall

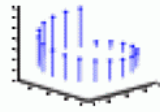


Mesh Graphs and Bar Graphs

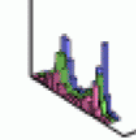
ezmesh



stem3



bar3

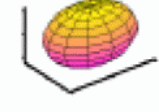


Area Graphs and Constructive Objects

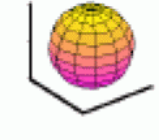
cylinder



ellipsoid

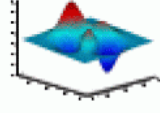


sphere

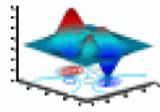


Surface Graphs

ezsurf



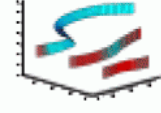
ezsurfz



Direction Graphs

Volumetric Graphs

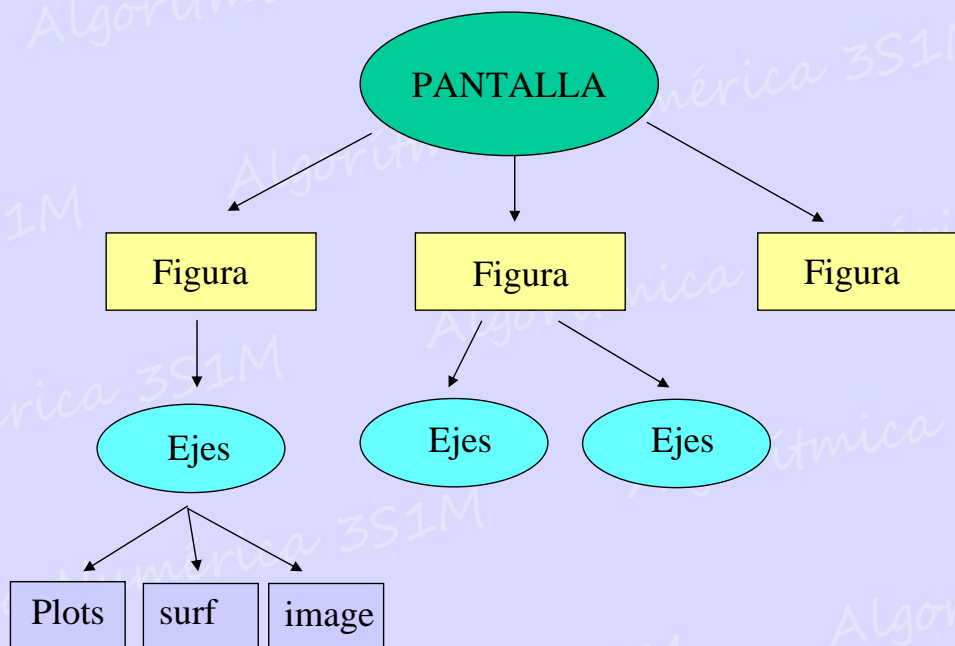
streamribbon



streamtube



Jerarquía de gráficos en MATLAB



Gráficos en 2D y 3D,
superficies, imágenes, etc.

Objetos gráficos.

- Cada uno de los niveles del gráfico anterior (figuras, ejes, plots) son objetos independientes cuyas propiedades pueden modificarse individualmente.
- Hay una jerarquía de padres e hijos: un plot sólo puede pintarse en unos ejes que a su vez deben estar en una figura.
- Si eliminamos (cerramos) una figura, eliminamos todos los objetos que contenga. Por el contrario, podemos eliminar unos ejes de una figura sin afectar a otros componentes (otros ejes, controles, etc.)
- No se puede crear un objeto sin los correspondientes ancestros. Esto es, podemos crear una figura por si sola pero no un plot aislado.
- En este curso NOS LIMITAREMOS a usar funciones de alto nivel (tipo plot). En ese caso MATLAB automáticamente se encargará de crear una figura y unos ejes donde pintar el gráfico.
- De esta forma haremos gráficos en MATLAB fácilmente sin tener que ser conscientes de la existencia de diferentes objetos gráficos.

Creación de figuras

- Las figuras (o ventanas) de MATLAB se crean (vacías) con la función `figure()` que devuelve un puntero a la ventana creada.
- `figure()` crea una nueva figura. Si se usa como argumento un puntero a una ventana ya existente, se hace activa esa ventana.
- El comando `gcf` (get current figure) nos da la ventana activa:

```
>> f1=figure,      f1 = 1
>> f2=figure,      f2 = 2
>> gcf,            ans = 2
>> figure(f1); gcf, ans = 1
```

- La función `close(f1)` cierra la ventana `f1`. Usando `close all` cierra todas
- Un comando gráfico (p.e. un plot) se ejecuta (pinta) sobre la ventana que esté activa.
- Si no existe ninguna figura se crea (junto con unos ejes), porque obviamente necesitamos una figura y unos ejes para pintar un gráfico.

Copiar Figuras y Gráficos

En muchas ocasiones se os pedirá adjuntar el gráfico resultante en vuestra entrega. Hay varias opciones:

1. Exportar la figura como un fichero JPG o similar usando el menú de la figura: File/Save As y eligiendo el tipo de fichero deseado. Luego se puede importar en vuestro documento.
2. Más sencillo es copiar la figura (Edit/Copy Figure) y hacer el correspondiente Ctrl-v en vuestro documento.
3. Otra opción hacer una captura de pantalla de la figura en cuestión y luego como antes un ctrl-v a vuestro documento. Para capturar estrictamente la figura de interés asegurarnos de que es la figura activa (pinchad sobre ella) y pulsar simultáneamente las teclas Ctrl-Alt-Impr Pant

Ejes

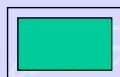
- Al llamar a un plot, el gráfico se pinta en los ejes activos. Si no existe ningún eje se crean en la figura activa (y si hace falta se crea la figura correspondiente). Dichos ejes por defecto ocupan toda la figura.

- Para poner varios ejes en una figura se usa `>> subplot(N,M,k);`

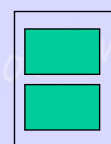
- `subplot(N,M,k)` organiza la figura para que quepan NxM ejes de igual tamaño en N filas y M columnas, crea el eje k (desde 1 hasta NxM) y lo hace activo, por lo que el siguiente plot pintará sobre él:

```
>> x=(0:0.01:1)*pi; s=sin(x); c=cos(x);
```

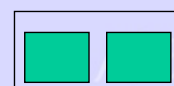
```
>> plot(x,s);
```



```
>> subplot(211); plot(x,s); subplot(212); plot(x,c);
```



```
>> subplot(121); plot(x,s); subplot(122); plot(x,c);
```



Comandos gráficos más comunes ($y=f(x)$)

- Dentro de los comandos que nos permiten hacer una grafica de una función $y=f(x)$, el más usado es `plot()`.
- Notar que MATLAB pinta colecciones de números, no funciones. Nosotros debemos generar previamente los x 's e y 's a pintar.

```
>> x=(0:100)*pi/100; y=sin(x); % Vectores a pintar (y frente a x)
>> figure; plot(x,y);
>> figure; plot(y);
```

- En el primer caso se pinta $y=\sin(x)$ frente a x (0 a π). En el segundo, al no tener información de x se pinta y frente al número de índice.
- Una opción es añadir una cadena texto con una letra indicando color elegido ('r','g','b') y un símbolo si deseamos línea no continua ('-' ':' ':') o si queremos usar un marcador especial ('o' círculo, 's' cuadrado, etc.)
- Otras propiedades del plot pueden cambiarse en el mismo comando, usando la notación ('Nombre propiedad', nuevo valor)

```
>> plot(x,y,'r:', 'LineWidth',3)
```

Superposición de varios plots en unos ejes

- Dentro de unos ejes podemos superponer varios gráficos (plots) al mismo tiempo (MATLAB usa automáticamente colores distintos).

```
>> plot(x,sin(x),x,cos(x),x,exp(-x)); % tres gráficas
```

- El comando `legend` permite identificar a los tres plots:

```
>> legend('sin(x)', 'cos(x)', 'exp(-x)');
```

- Si tratamos de pintar varios gráficos con sucesivas llamadas a `plot()` veremos que sólo el último gráfico aparece (machaca a los anteriores).
- La solución es usar el comando `hold on` que indica a MATLAB que a partir de ahí los nuevos plots no deben machacar a los antiguos. Cuando terminemos de superponer gráficos haremos un `hold off`:

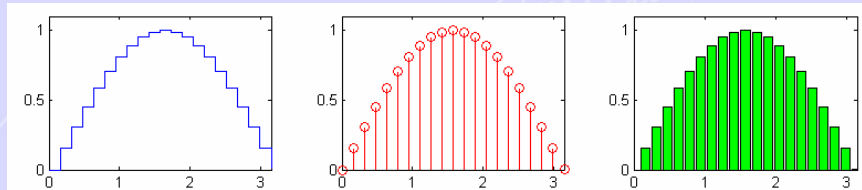
```
plot(x,sin(x),'r'); hold on;
plot(x,cos(x),'b');
plot(x,exp(-x),'g'); hold off
```

- En este caso se deben especificar nosotros colores distintos, ya que al ser plots separados MATLAB usaría siempre el color por defecto (azul).

Variantes de un plot

- Hay varios comandos que pintan gráficos similares a un plot de datos {y} frente a datos {x}, pero con una visualización distinta:

```
x=(0:20)*pi/20; y=sin(x); % Vectores a pintar (y frente a x)
subplot(131); stairs(x,y,'b');
subplot(132); stem(x,y,'r');
subplot(133,'g'); bar(x,y);
```



- Más usados durante este curso serán los gráficos usando escalas logarítmicas en uno u otro eje (o en ambos):

`semilogx(x,y)`, `semilogy(x,y)`, `loglog(x,y)`

Plots en ejes logarítmicos

- Muchas veces es interesante usar ejes en escala logarítmica:
 - `semilogy` (escala log en Y),
 - `semilogx` (escala log en X)
 - `loglog` (ambos ejes en escala log).
- Los parámetros de estas funciones son los mismos que un plot

```
>> x = (0:0.1:10); y = exp(-x);
>> subplot(311); plot(x,y,'r.')
>> subplot(312); semilogy(x,y,'b')
>> subplot(313); loglog(x,y,'ko')
```

