Introducción a `matlab/octave`

Prácticas de Cálculo Numérico I (doble grado)

16 de febrero de 2021

1. Datos generales

Nombre de la asignatura: Prácticas de Cálculo Numérico I.
Profesor: Fernando Chamizo.
Horario: Martes 11:30–13:30.
Modalidad de docencia: En *streaming* via TEAMS. Se intentará grabar todas las clases, las cuales quedarán almacenadas en Microsoft Stream hasta la fecha de caducidad impuesta por esta plataforma.
Página web del curso: `http://matematicas.uam.es/~fernando.chamizo/` (seleccionando "Prácticas de Cálculo Numérico I").

2. Instalación y primer uso de `matlab/octave`

Entre el *software* empleado en cálculo numérico con fines académicos, sobre todo dentro del orientado a ingeniería, el que ha alcanzado una mayor difusión es `matlab` cuyo nombre abrevia *matrix* y *laboratory*. El nombre da una idea de que, al menos originariamente (ya tiene casi 40 años), estaba orientado a cálculos con matrices. Actualmente su ámbito es más amplio, especialmente cuando se instalan paquetes adicionales, sin embargo las matrices se siguen reflejando en la propia sintaxis y en mi experiencia (quizá sesgada) son los cálculos con ellas lo que mejor hace.

A pesar de que internamente esos cálculos se llevan a cabo con una adaptación de rutinas que hoy son de dominio público, `matlab` es *software* comercial propietario. Un punto a favor es que las licencias para estudiantes no son astronómicas y muchas universidades, como la UAM, tienen licencias generales. De todas formas, una alternativa libre y gratuita es (GNU) `octave`. Los comandos y programas básicos son intercambiables hasta el punto de que fuera de un ámbito relativamente avanzado, te parecerá que `octave` es un emulador perfecto de `matlab` (salvo alguna cuestión fina en

las representaciones gráficas). En particular, es de prever que todo lo que veremos en estas prácticas no establecerá ninguna diferencia. Más adelante habrá algunos comentarios dedicados `octave`, a pesar de que con seguridad la mayor parte de vosotros utilizaréis `matlab`. Un uso más avanzado muestra ventajas a favor de `matlab`, sobre todo con el uso de librerías, llamadas *toolboxes*, para temas especiales.

Para instalar `matlab` necesitamos una licencia. La UAM tiene una de campus disponible para todos los estudiantes. Las instrucciones para emplearla, por supuesto teniendo una dirección de correo institucional, están descritas siguiendo el enlace correspondiente en:

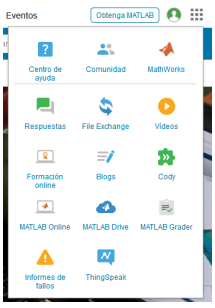
`https://faq.uam.es/index.php?action=show&cat=38`

Es bastante posible que muchos no deseéis ocupar disco duro en vuestro ordenador con programas grandes si hay alternativa. En este caso sí la hay, y es emplear `matlab` *online*. En principio es el formato que pienso utilizar en las clases porque reflejará la situación de la mayoría y además me permitirá no tener que transportar el ordenador si cambio de lugar en el campus.

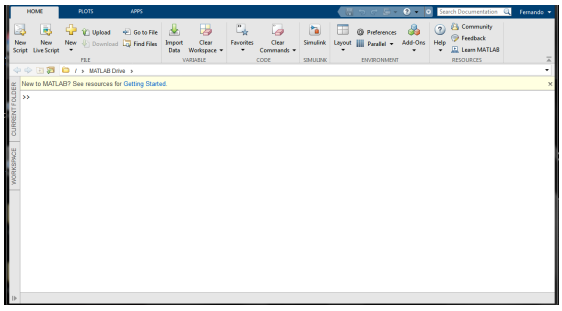
En realidad dispones de dos formas de usar `matlab` a través de la red. Una es con el servidor jupyter del departamento de matemáticas (`https://jupyter.mat.uam.es/hub/login`) pero esta no es la mejor idea porque requiere que sepas algo de jupyter y mi experiencia es que hay algunos problemas (sobre todo con los gráficos). Más conveniente es utilizar la web oficial de `matlab`. Una vez registrados, accederemos a través de

`https://matlab.mathworks.com/`

y tras entrar en nuestra cuenta, en el menú de la derecha, elegimos MATLAB Online que nos lleva al interfaz principal.



MATLAB Online



Aspecto inicial de la interfaz

Una posibilidad un poco más directa es acceder con `https://www.mathworks.com/products/matlab-online.html` y pulsar *Start using MATLAB Online*.

Si deseamos instalar `octave`, la página del proyecto es:

`https://www.gnu.org/software/octave/`

Allí están los instaladores para diferentes sistemas operativos. La instalación básica es más ligera que la de de `matlab`, la última versión para Windows ocupa 1.8 G en el disco duro. Al parecer, `octave` también se puede usar *online* en `https://octave-online.net/`, pero no tengo experiencia al respecto.

Al ejecutar `octave` obtenemos una interfaz parecida a la de `matlab` pero más espartana. Insisto, no obstante, que es difícil notar diferencias moviéndose en un nivel básico.

3. Uso como una calculadora avanzada

El *prompt* está pidiendo que escribamos algo y ahí podemos teclear operaciones como en una calculadora. Ni que decir tiene que el producto se indica con `*` y, por otro lado, las potencias se indican con `^`, como en `sagemath` de la asignatura de Laboratorio. Por ejemplo, tecleando `1+1` y pulsando *enter*, obtendremos nuestra primera sesión

```
>> 1+1
ans = 2
```

Aquí `ans` es la abreviatura de *answer* (para decir toda la verdad, en realidad en la versión *online* la respuesta sale menos compacta, ocupando varias líneas). Por supuesto, existen todas las funciones que habitualmente pueblan las teclas de nuestras calculadoras y que incorporan los principales lenguajes de programación. Sin ánimo de ser exhaustivo, se tienen `sqrt` (raíz cuadrada), `exp`, `log` (logaritmo neperiano), las funciones trigonométricas `sin`, `cos`, `tan` y sus inversas `asin`, `acos`, `atan`. Además tenemos el número π denotado por `pi`. Por ejemplo:

```
>> ( log(pi) + exp(1) )*( (sin(1))^2 + sqrt(6) )
ans = 12.1977
```

Los espacios son irrelevantes, solo se han introducido para dar legibilidad al código.

Si te sientes defraudado por este ejemplo, ya que tu calculadora te da mayor precisión, es el momento que conozcas que hay varios formatos posibles y los dos más usados son `format short` y `format long`. Por defecto, estamos en el primero que da solo precisión simple. Si tecleamos `format long` y después nuestro ejemplo anterior, el resultado pasa a ser `12.197703479831025` que posiblemente mejore a tu calculadora de bolsillo. Estos dos formatos admiten

algunas variantes que no examinaremos aquí. Solo mencionaré `format rat` que aproxima por una fracción y en el caso anterior daría `5306/435`.

Una constante que no está en las calculadoras más básicas es `i`, la unidad imaginaria. Sin nada especial, podemos hacer operaciones con números imaginarios, incluso con funciones trascendentes. Por ejemplo, la famosísima fórmula de Euler $e^{i\pi} + 1 = 0$, es coherente con que `exp(i*pi)` resulte `-1.0000+0.000i` y con que `log(-1)`, que no nos dejaban hacer en secundaria, resulte `0.0000+3.1416i`. Por cierto, dependiendo de la versión de `matlab/octave` puede que la parte imaginaria de `exp(i*pi)` sea un número del orden de 10^{-16} en vez de ver `0.000`. De hecho, en la versión actual de `matlab` (*online*), el resultado antes mencionado contrasta con que `exp(pi*i) + 1` dé `0.0000e+00 + 1.2246e-16i` (recuerda que la notación científica *a*e*b* abrevia $a \cdot 10^b$). Seguro que estás sospechando, acertadamente, que esto se debe a que `matlab` (a diferencia de `sagemath`) no está haciendo cálculo simbólico. Hay cierto error interno con números que no son "exactos". La falta de precisión se cuantifica con el llamado *épsilon máquina*. Este es el menor error relativo que es posible distinguir. En `matlab` el épsilon máquina se obtiene con el comando `eps`. Si lo tecleamos, obtendremos `2.2204e-16`. Parece un número que no tiene nada de particular pero si calculas `2^-52` quizá cambies de opinión. La inmensa mayoría de los instrumentos digitales de cálculo que hayas utilizado en tu vida están afectados exactamente por este épsilon máquina. La razón es que el formato empleado habitualmente para almacenar números con 64 bits no permite un error relativo menor.

Si todo esto del error relativo más pequeño te resulta incomprensible, el siguiente ejemplo te iluminará:

```
>> (0.1+eps)-0.1
ans = 2.2204e-16

>> (10+eps)-10
ans = 0
```

La moraleja es que aunque nuestra calculadora u ordenador sea capaz de tratar con números tan pequeños como 10^{-100} y distinguirlos de cero, eso no significa que sea capaz de discriminar cualquier par de números que difieran en 10^{-100}. El límite real es del orden de 10^{-16} y este problema es bastante universal, no una manía de `matlab`.

Volviendo a la comparación con la calculadora, algo en lo que `matlab` va más allá es que está permitido el uso de variables como en cualquier lenguaje de programación típico. Además, al escribir la variable recuperamos su valor. Por ejemplo, digamos que queremos saber cuántos minutos ha tenido 2020. Para ello creamos las variables `d` de días, `hd` de horas por día y `mh` de minutos

por hora. La cuenta sería

```
>> d = 366
d =  366
>> hd = 24
hd =  24
>> mh = 60
mh =  60
>> d*hd*mh
ans =  527040
```

Es un poco pesado ver el mismo resultado que tecleamos. Una solución, que a la hora de programar será casi necesidad, es añadir al final de la línea un punto y coma, lo cual omite la salida. Es decir, teclearíamos `d = 366;` etc. excepto en la última línea de la que queremos ver el resultado. Digamos que ahora queremos hacer el cálculo para 2021, entonces `d` se reduce en uno porque no es bisiesto y podríamos proceder con

```
>> d = d - 1;
>> d*hd*mh
ans =  525600
```

4. Organizando los cálculos en ficheros

En cuanto queramos hacer algo más que un cálculo inmediato, es mejor utilizar un fichero en vez de teclear separadamente cada cálculo. Esto nos permitirá hacer modificaciones de manera más eficiente y se convertirá en una necesidad para programar.

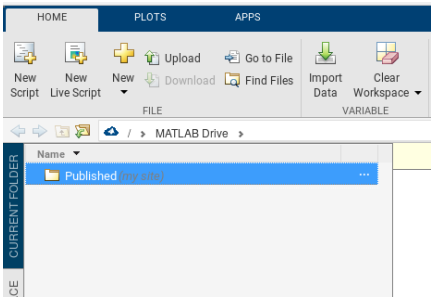
Los ficheros con instrucciones matlab tienen la extensión `.m` y esencialmente son de dos tipos: *scripts* y funciones. En realidad, la terminología varía según los autores. En un contexto informático general, los primeros son lo que llamaríamos programas y las segundas, subrutinas a las que llaman los programas.

Por ahora nos centramos en los *scripts*. Imaginemos que queremos crear uno llamado `ex01.m`. Distingamos dos situaciones, en la primera teniendo `matlab/octave` instalado y en la segunda con `matlab` *online*.

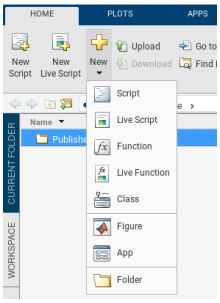
En el primer caso, las versiones modernas de `matlab/octave` llevan incorporado un editor pero si nos gusta más el nuestro (esto se aplica sobre todo a los maniáticos de Linux), nada impide que usemos el que queramos (que es lo que hago yo, sobre todo con `octave`). Así que basta crear con nuestro editor o con el editor incorporado, un fichero de texto `ex01.m` de la manera habitual (posiblemente File, New,...). Para que `matlab/octave` tenga acceso al fichero debe estar en el `path` correspondiente. Existen los comandos de Linux `cd`, `ls`, `pwd` que nos permitirán saber dónde estamos y

movernos donde esté el fichero (y también es posible hacerlo con la interfaz). Soy consciente de que la explicación es un poco críptica pero al buen entendedor... y prefiero centrarme en el la segunda posibilidad, que posiblemente sea la mayoritaria.

Veamos ahora con todo detalle cómo crear un *script* en `matlab` *online*, que quedará almacenado en la nube. Para hacerlo un poco más completo vamos a meterlo dentro de un directorio llamado `week01`. Si desplegamos la pestaña `Current folder` veremos que `MATLAB Drive` solo contiene una carpeta llamada `Published`. Para crear `week01` pulsamos en `New` y en el desplegable escogemos `Folder`. Nos pedirá el nombre, `week01` en nuestro caso.

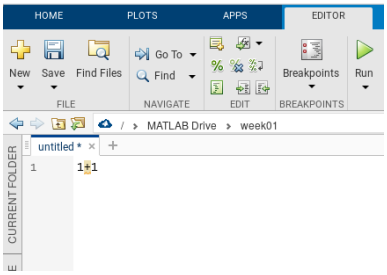


Pestaña Current folder

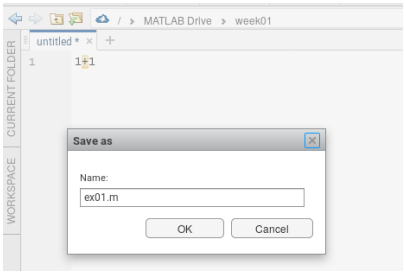


Desplegable de New

Con ello, en `Current folder` ya tenemos el nuevo directorio y nos metemos en él simplemente pinchando encima. Notemos que en la línea bajo el menú gráfico, `>MATLAB Drive` se ha completado con un `>week01` a la derecha. Podemos usar esa línea con el ratón para navegar entre los directorios. O bien pulsando de nuevo `New`, pero ahora escogiendo `script`, o bien haciendo lo mismo pulsando el botón derecho del ratón dentro de la pestaña `Current folder`, creamos un *script*. De la segunda forma podremos ajustar directamente el nombre, de la primera, cuando demos a `Save` nos pedirá un nuevo nombre (por defecto es `untitled.m`.



Nuevo script



Grabar el nuevo script

En las imágenes, `ex01.m` tiene un contenido bastante anodino, solo `1+1`. Si lo ejecutamos pulsando el botón `Run` (el triángulo verde) o escribiendo en la ventana de comandos `ex01` (sin el `.m`), obtendremos la consabida respuesta `ans = 2`.

Iremos viendo a lo largo del curso muchos comandos. Por ahora, juguemos a hacer nuestro uso tipo calculadora pero con más comodidad. Los comentarios se escriben precediendo las líneas con `%`.

```
1  % Aproximando el número e
2  n = 50;
3  (1+1/n)^n
4  % Con un n doble del anterior
5  n = 2*n;
6  (1+1/n)^n
7  % sin(1) y su aproximación de Taylor de orden 7
8  sin(1)
9  1-1/factorial(3)+1/factorial(5)-1/factorial(7)
10 % Diferencia de sin(1) y Taylor de orden 7
11 sin(1) - (1-1/factorial(3)+1/factorial(5)-1/factorial(7) )
```

Los puntos y comas 2 y 11 impiden que se vean los valores 50 y 100 de `n`. Por cierto, indico los números de línea para hacer referencia a ellos y facilitar la copia, por supuesto no hay que teclearlos.

5. Matrices, la gran fortaleza de `matlab/octave`

Cada lenguaje de programación tiene sus peculiaridades: en C los punteros, en C++ y Java las clases y en python las listas. Como indica su nombre, en `matlab` la estructura de datos estrella son las matrices. En muchas ocasiones, sustituirán a los bucles que utilizaríamos en los lenguajes mencionados. La manera de indicar una matriz es limitándola con corchetes e indicando los cambios de fila con punto y coma. Los elementos de una fila se pueden separar con comas pero esto no es obligatorio, con espacios es suficiente. La suma y el producto de matrices no requieren comandos especiales, bastan `+` y `*`. La traspuesta se indica con el apóstrofo. Para ser del todo rigurosos, lo que se consigue con él es la traspuesta conjugada pero para matrices reales no hay diferencia. La matriz nula `m` $\times$ `n` se indica con `zeros(m,n)` y la matriz identidad de dimensión `n` mediante `eye(n)`. También existe `ones(m,n)` que general una matriz `m` $\times$ `n` con todos sus elementos iguales a uno.

Como ejemplo, escribamos en un *script* algunas operaciones con las matrices

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 2 & -1 & 1 \end{pmatrix} \qquad \text{y} \qquad B = \begin{pmatrix} -2 & 1 & 0 \\ 3 & 1 & 1 \end{pmatrix}$$

que también involucren matrices especiales.

```
% Definiendo dos matrices
A = [1, 0, 1; 2, -1, 1];
B = [-2, 1, 0; 3, 1, 1];
% Calcula la suma de matrices
A + B
% La traspuesta de B es
B'
% El producto de A por la traspuesta de B es
A*B'
% Esto da A
A + zeros(2,3)
% y esto vuelve a dar A*B'
A*B'*eye(2)
```

Un comentario para los que uséis `octave` en vez de `matlab` *online*, es que cuando la salida ocupa más de una "página" debemos pasar la salida línea a línea. Para evitar esto, basta teclear en la ventana de comandos `more off`. Si queremos restablecer este comportamiento, usaremos `more on`.

El elemento en la fila `m` y columna `n` de una matriz `A` se indica con `A(m,n)`. Así, en el ejemplo anterior `A(2,1)` resultaría 2 y `B(1,1)` sería -2. A diferencia de lo que ocurre en `sagemath` y en los principales lenguajes de programación, en `matlab/octave` los índices no empiezan en cero sino en uno.

Las matrices admiten ser construidas por bloques conservando la misma sintaxis que tendrían si los bloques fueran números. Por ejemplo, en física la matriz de Dirac γ_2 es una matriz que se construye a partir de la matriz de Pauli σ_2 por medio de

$$\gamma_2 = \begin{pmatrix} O & -\sigma_2 \\ \sigma_2 & O \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \qquad \text{con} \qquad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

Esta matriz es unitaria, es decir, al multiplicarla por su traspuesta conjugada da la identidad. Escribamos su construcción y la comprobación de que es unitaria en un *script*.

```
% Matriz de Pauli
s_2 = [0 -i; i 0]
% Matriz de Dirac
g_2 = [zeros(2) -s_2; s_2 zeros(2) ]
% Comprobamos que es unitaria
g_2*g_2'
```

La construcción por bloques se produce en la línea 4 y de paso aprendemos que `zero(n)` es una abreviatura de `zero(n,n)` (lo mismo funciona para `ones`). La matriz γ_2 es simétrica, por tanto la traspuesta conjugada es lo mismo que la conjugada. La conjugación se indica mediante `conj`, entonces sustituyendo la línea 6 por `g_2*conj(g_2)`, seguimos obteniendo la identidad.

Los vectores (fila o columna) no son otra cosa que matrices que tienen una de sus dimensiones iguales a uno. Sin embargo hay dos constructores de vectores fila que merecen una consideración especial.

Generalizando ligeramente lo que ocurre en python, con `[`a`:`b`]`, o simplemente con a`:`b, obtenemos un vector fila cuyos elementos son a, $a+1$, etc. hasta el último valor que no supera a b. De esta forma $2:4$ es el vector `[2, 3, 4]` y $1{,}7:4$ es `[2.7, 3.7]`. Si necesitamos un incremento diferente de la unidad, basta indicarlo entre los dos puntos. Así podemos generar cualquier vector cuyos elementos estén en progresión aritmética. Por ejemplo:

```
[2:3:15]     →   [2, 5, 8, 11, 14]
```

donde los corchetes son superfluos.

Necesitaremos, a veces, definir estos vectores con elementos en progresión aritmética especificando el origen, el final y el número de elementos. Por supuesto lo podemos conseguir con los dos puntos haciendo un pequeño cálculo en sucio, pero es más directo emplear el comando `linspace(`a`,`b`,`n`)` donde n es el número de elementos. Por ejemplo, `linspace(2,5,8)` resultaría

```
2.0000 2.4286 2.8571 3.2857 3.7143 4.1429 4.5714 5.0000
```

(así, sin corchetes ni comas es como aparece en la salida). El incremento es de $3/7$ y por tanto equivale a `2:3/7:5`.

Por cierto, la longitud de un vector se obtiene con el comando `length` y las dimensiones de una matriz con `size`.

La notación de los dos puntos sirve también para establecer rangos de índices de vectores o matrices. Veámoslo con un ejemplo.

```
% Una matriz y un vector fila
A = [1, 0, 1; 2, -1, 1];
v = [2, 3, 5, 7];
% La matriz formada por las dos últimas columnas
A(1:2,2:3)
% Una alternativa
A(:,2:3)
% La sección central de v
v(2:3)
% Muestra las dimensiones de A y v
size(A)
size(v)
length(v)
% Anulamos las dos últimas columnas de A
A(:,2:3) = zeros(2)
```

De hecho la última instrucción se podría escribir como `A(:,2:3) = 0` porque cuando no hay ambigüedad, los números se identifican con matrices de las dimensiones adecuadas. Por ejemplo, `12+A` es válido y `12` se identifica

con la matriz 2×3 que tiene todas sus elementos iguales a 12. Por si sirve para que los matemáticos más estrictos no despotriquen de los informáticos, este convenio también se aplica en algunas partes de la física.

Veamos otro ejemplo para practicar. Se sabe que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \qquad \text{para} \quad n \geq 1,$$

donde F_n son los números de Fibonacci. Utilizando esta fórmula, vamos a hacer un programilla, poco eficiente, que dado n nos diga cuánto vale F_n. Utilizaremos los comandos de entrada/salida `input`, que pide un dato al usuario y `disp` que muestra el valor de una variable sin que aparezca `ans =`. En una sección posterior volveremos sobre los comandos de entrada/salida.

```
% Pedimos la n
n = input('Introduce n');
% Elevamos a n
F = [1,1;1,0]^n;
% El resultado es el elemento 1,2 (o el 2,1)
disp(F(1,2))
```

Como cabe esperar, elevar a un n natural una matriz en `matlab` es la abreviatura de multiplicarla por sí misma n veces. Esto funciona incluso con números enteros si la matriz es invertible.

Las funciones trascendentes se aplican elemento a elemento a una matriz. De esta forma `exp(A)` no tiene nada que ver con el e^A que te explica(rá)n en los cursos de ecuaciones diferenciales sino que es algo tan prosaico como calcular e elevado a cada elemento. De esta forma

```
exp([0,1]) → 1.0000 2.7183,     sin(ones(2)) → 0.84147 0.84147
                                               0.84147 0.84147
```

donde, recuérdese, que `ones(2)` abrevia `ones(2,2)`.

A veces nos gustaría tener un producto de matrices, o potencias naturales, término a término. En principio parece un antojo raro pero en breve veremos que es fundamental por ejemplo a la hora de representar gráficas. La manera de indicar que `*` o `^` se deben hacer término a término es precediéndolos de un punto. Por ejemplo, `[1:10].^2` es un vector fila con los 10 primeros cuadrados y `[1,2].*[6,3]` resulta `[6,6]`. Con la multiplicación usual `*` no tiene sentido multiplicar dos matrices 2×3 pero sí lo tiene empleando `.*` para obtener otra matriz de las mismas dimensiones. Análogamente, `./` define una división término a término sin problemas siempre que los denominadores no se anulen.

6. Gráficas en dos dimensiones

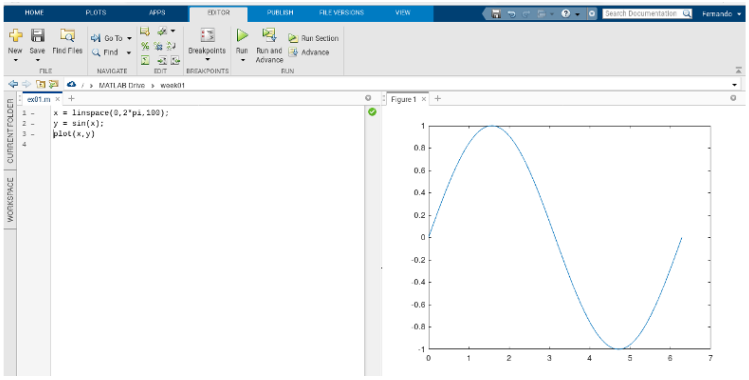
Posiblemente parte del éxito inicial de `matlab` en el mundo académico se debiera a que, en comparación con el poco *software* disponible entonces, hacía gráficos bonitos con poco esfuerzo.

El comando básico para hacer gráficas bidimensionales en `plot`. Su estructura más simple es `plot(`a`,`b`)` donde a y b son dos vectores de la misma longitud. Lo que hace es emparejar los elementos respectivos de a y b para formar puntos y después los une.

Si queremos dibujar la gráfica de $y = \sin(x)$ en el intervalo $x \in [0, 2\pi]$ debemos crear un vector que tenga muchos valores de x en este intervalo y construir consecuentemente y. Lo conseguimos con el *script*

```
1  x = linspace(0,2*pi,100);
2  y = sin(x);
3  plot(x,y)
```

Al ejecutarlo en `matlab` *online* veremos que aparece una figura a la derecha, con la etiqueta `Figure 1`, En `octave` y versiones instaladas antiguas de `matlab` la figura sale fuera de la interfaz (como una ventana *pop-up*).



El script y la figura

Si cambiamos 100 en la primera línea por un valor mucho más pequeño, apreciaremos la discretización.

Imaginemos que ahora queremos dibujar $y = e^{-x}\sin(x)$ en el mismo intervalo. Un error de principiante sería escribir en la segunda línea la expresión `exp(-x)*sin(x)`. Esto es incorrecto porque `exp(-x)` y `sin(x)` son vectores fila de 100 elementos, esto es, matrices 1×100 y es imposible multiplicarlas con `*`, lo correcto es `exp(-x).*sin(x)`.

Si ahora pensamos en cómo representar $y = (1+x)^{-1}\sin(x)$, veremos la razón de ser del convenio de interpretar los números como matrices de las

dimensiones adecuadas. Gracias a él, no hay que convertir `1` en un vector de unos para sumárselo a `x`. Dos maneras naturales válidas de introducir esta función en la segunda línea son:

```
(1+x).^-1.*sin(x)      y      sin(x)./(1+x)
```

A continuación de los vectores que sirven de argumento a `plot`, es posible incluir una cadena de caracteres, limitada por apóstrofos o comillas y separada mediante una coma, con información sobre el color, el estilo de la línea y el símbolo usado para los puntos, según las tablas:

Color →

cyan	magenta	yellow	red	green	blue	white	black
`c`	`m`	`y`	`r`	`g`	`b`	`w`	`k`

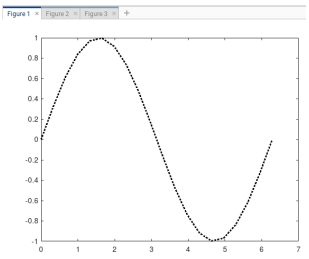
Estilo de línea →

solid	dashed	dotted	dash-dot
`-`	`--`	`:`	`-.`

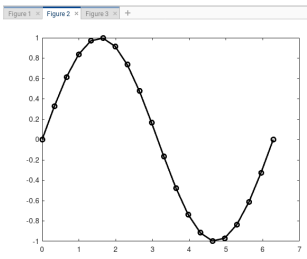
Punto →

plus	circle	asterisk	x	square	diamond
`+`	`o`	`*`	`x`	`s`	`d`

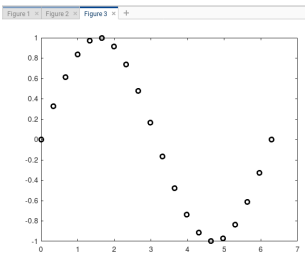
Por ejemplo, con `plot(x,y,'k:')` obtendremos la gráfica con línea de puntos en negro, mientras que con `'k-o'` la línea será sólida conectando puntos marcados con un círculo y con `'ko'` la línea desaparecería. Si comprobáramos estos ejemplos con nuestro código inicial, los puntos estarían demasiado juntos y se superpondrían. Estos serían los resultados cambiando el vector en la primera línea por `linspace(0,2*pi,20)`.



`plot(x,y,'k:')`



`plot(x,y,'k-o')`



`plot(x,y,'ko')`

Las pestañas que se han dejado encima de las figuras dan que pensar que estas figuras no fueron creadas modificando cada vez el `plot`. En realidad se utilizó solo el código común:

```
x = linspace(0,2*pi, 20);
y = sin(x);
figure(1)
plot(x,y, 'k:', 'linewidth',2)
figure(2)
```

```
6  plot(x,y, 'k-o', 'linewidth',2)
7  figure(3)
8  plot(x,y, 'ko', 'linewidth',2)
```

Si precedemos una gráfica con `figure(`n`)` los situará en la pestaña n-ésima (o en la venta n-ésima en `octave` o versiones instaladas antiguas de `matlab`). Por otro lado, `'linewidth',2` (o si se prefiere, `'LineWidth',2`) indica que el grosor de la línea pasa a ser 2. El grosor por defecto es 0.5 y la razón de aumentarlo es para favorecer aquí la visibilidad de las figuras.

Las imágenes anteriores son simples capturas de pantalla pero podemos guardar los ficheros directamente. Pinchando sobre cada figura pasaremos al menú `FIGURE` y uno de sus botones es `Save` (también hay un menú que aparece sobre las figuras al mover el ratón). Si usamos `matlab` *online*, con él podemos guardarlo en la nube y después descargarlo de allí o bien entrando en nuestra cuenta y eligiendo `Matlab Drive` en vez de `Matlab Online` o bien sin salirnos de la interfaz buscando el fichero en `Current Folder`, seleccionándolo y usando en el menú `Home` el botón `Download` (la flecha hacia abajo). Más adelante veremos cómo hacer esto también por *software*.

Las figuras admiten rótulos con `xlabel`, `ylabel` y `title` cuyo nombre se explica a sí mismos y cuyo uso queda claro con la gráfica inicial:

```
1  x = linspace(0,2*pi, 100);
2  y = sin(x);
3  plot(x,y)
4  xlabel('Eje X')
5  ylabel('Eje Y')
6  title('Mi primera gráfica')
```

Hay algunas consideraciones que destacar relativas a la superposición de gráficas. La primera es que usar un `plot` y después otro no produce superposición, sino que cada uno borra el resultado anterior. Para superponer gráficas una posibilidad es repetir los argumentos de `plot`. Por ejemplo,

```
1  x = linspace(0,2*pi, 100);
2  y = sin(x);
3  y2 = cos(x);
4  plot(x,y,'--',x,y2,':')
```

muestra en la misma figura la gráfica del seno en línea discontinua y la del coseno en línea de puntos. Nótese que automáticamente se asignan colores diferentes. Podemos cambiar este comportamiento especificando un color. Añadiendo `legend(`*texto*`1,`*texto*`2)` se mostrará la leyenda: un cuadro que explica cada gráfica. Cuando uno se canse ella, basta incluir `legend('off')` en el *script* o teclearlo en la ventana de comandos.

Otra manera de superponer gráficas es con `hold on`. El programa anterior equivale a

```
1  x = linspace(0,2*pi, 100);
2  y = sin(x);
3  y2 = cos(x);
4  plot(x,y,'k--')
5  hold on
6  plot(x,y2,'k:')
7  hold off
```

El `hold off` no es necesario pero es aconsejable porque lo necesitaremos si queremos utilizar la ventana para nuevas gráficas.

Un ejemplo iluminador (que agradezco al profesor del otro grupo). Es dibujar una circunferencia usando `plot`. Lo natural parece superponer dos gráficas de semicircunferencias. Por ejemplo:

```
1  x = linspace(-1,1, 100);
2  y = sqrt(1-x.^2);
3  plot(x,y,'k')
4  hold on
5  y = -sqrt(1-x.^2);
6  plot(x,y,'k')
7  axis('square')
8  hold off
```

La línea 7 es para que la ventana de la figura sea cuadrada y así no deforme la escala. Otra posibilidad, conservando la ventana rectangular habitual, sería `axis('equal')`. Sin embargo hay una solución más elegante. Conocemos una parametrización de la circunferencia con un solo trozo, $x = \cos t$, $y = \operatorname{sen} t$ y nadie ha dicho que los valores que aparecen como `x` tengan que ser crecientes. Con esta idea, utilizamos

```
1  t = linspace(0,2*pi, 100);
2  plot (cos(t),sin(t))
3  axis('square')
```

Hay muchas finuras para adornar nuestras gráficas, como cambiar el tamaño de las fuentes o incluir texto. Todas esas cosas exceden lo que nosotros vamos a usar en el curso. Si quieres "lucirte", lo mejor es que consultes la documentación.

7. Ejemplos de gráficas en tres dimensiones

En los cursos de análisis de primero es muy posible que sacaras la idea de que la representación de gráficas de funciones $y = f(x)$ siguen una mecánica (crecimiento, decrecimiento, asíntotas, puntos de inflexión ...) pero la de

superficies $z = f(x, y)$ es una especie de arte. Un problema que se refleja en el *software* es que cuando representamos un objeto tridimensional en una pantalla plana, el punto de vista es muy importante para darnos una idea certera. En `matlab/octave` hay muchos comandos para la representación de superficies pero aquí solo veremos los dos más simples y algunos ejemplos sin muchas explicaciones mostrando resultados más atrayentes.

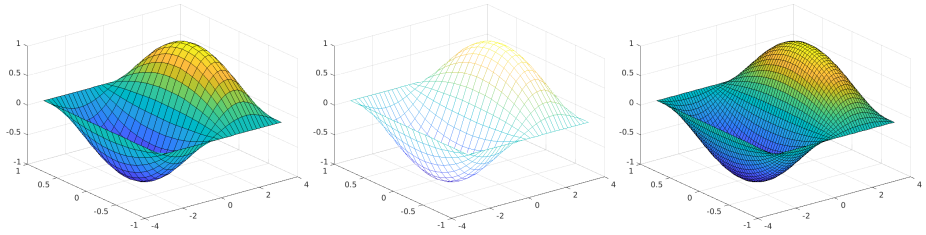
Los comandos más básicos para representar superficies son `mesh` y `surf`. Ambos tienen tres argumentos, la x, la y y la z, y dibujan una malla correspondiente a la discretización de los datos. Difieren en que el primero muestra la malla como si las "baldosas" que limitan fueran blancas mientras que el segundo las colorea.

Supongamos que queremos dibujar la superficie $z = (1 - y^2)\,\mathrm{sen}\, x$ con $(x, y) \in [-\pi, \pi] \times [-1, 1]$. Esta fórmula es `sin(x).*(1-y.^2)` en formato `matlab/octave`. Un error de principiante sería definir `x` e `y` como vectores, al igual que en la sección anterior. Si lo pensamos un instante, esto no puede funcionar porque con la expresión anterior estamos pidiendo a `matlab/octave` que haga productos elemento a elemento, por ejemplo $x = -\pi$ irá con $y = -1$. Lo que en realidad deseamos es que $x = -\pi$ se combine con *todos* los valores de y. La manera de conseguir este efecto es utilizar el comando `meshgrid` que repite el vector `x` tantas veces como valores de `y` haya y hace lo mismo con `y`, convirtiendo a `x` e `y` en matrices (bidimensionales). Si no quieres dedicar tiempo a entender este argumento, todo lo que tienes que saber es usar el siguiente ejemplo como plantilla:

```
1  [x,y] = meshgrid(linspace(-pi,pi,30), linspace(-1,1,20));
2  z = sin(x).*(1-y.^2);
3  surf(x,y,z)
```

Por supuesto, los nombres `x`, `y`, `z` son arbitrarios. Si todavía no has perdido la esperanza de entender lo que hace realmente `meshgrid` mira la salida que ofrece `[a,b] = meshgrid([1,2,3],[7,8])`.

Las siguientes figuras muestran el resultado de `surf`, `mesh` y el efecto de discretizar con `surf` de manera más fina:



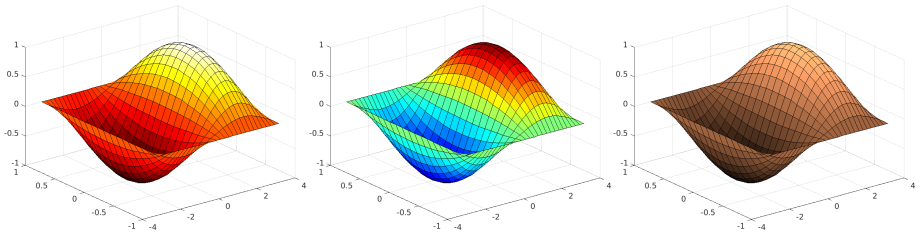
`surf` `mesh` `surf 30, 20` $\to$ `50`

Los comandos `xlabel`, `ylabel` y `title` funcionan como antes y además se añade un `zlabel`.

Los colores asignados dependen del valor de la altura siguiendo lo que se llama un *mapa de color* que se especifica mediante `colormap(`*nombre*`)` o `colormap` *nombre*. Algunos de los mapas de color predefinidos son:

`hsv, hot, cool, pink, gray, bone, jet, copper`

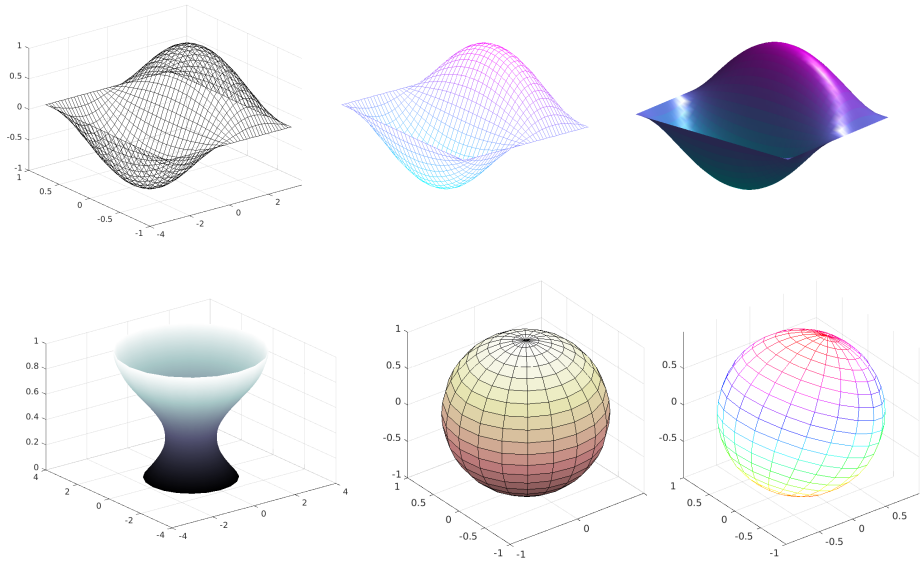
Aquí van tres ejemplos:



`colormap hot` `colormap jet` `colormap copper`

Mirando la documentación aprenderás a crear tus propios mapas de colores, aunque pocos usuarios lo encuentran necesario.

Terminamos con seis ejemplos que dan lugar a las siguientes figuras con el código de más abajo.



En la última figura se ha intentado simular la inclinación de unos 23° del eje de la Tierra (girar 180° alrededor de un eje inclinado 11,5°).

```
1  [x,y] = meshgrid(linspace(-pi,pi,40), linspace(-1,1,30));
2  z = sin(x).*(1-y.^2);
3
4  figure(1)
5  surf(x,y,z,'FaceColor','none')
6
7  figure(2)
8  mesh(x,y,z)
9  colormap cool
10 axis off
11
12 figure(3)
13 surf(x,y,z,'EdgeColor','none')
14 colormap cool
15 camlight right
16 lighting phong
17 view([-50,30])
18 axis off
19
20 figure(4)
21 th = linspace(-3,pi/2,100);
22 r = 2 + sin(th);
23 [x,y,z] = cylinder(r);
24 surf(x,y,z)
25 shading interp
26 colormap bone
27
28 figure(5)
29 sphere(20)
30 colormap pink
31 axis square
32
33 figure(6)
34 [x,y,z] = sphere(20);
35 eje_rot = [sin(11.5*pi/180) 0 cos(11.5*pi/180)];
36 rotate( mesh(x,y,z), eje_rot, 180)
37 colormap hsv
38 axis equal
```

En `octave` la línea 16, que especifica cierto algoritmo para la iluminación, no funciona. Sin ella se ven los parches que forman la superficie.

Quien tenga interés en los comandos y parámetros que aparecen en el código anterior, debería consultar la documentación.

8. Control de flujo

Al igual que en cualquier lenguaje de programación típico, es posible alterar el orden natural en la ejecución de las líneas de un programa con `for` y con `if`. La sintaxis de `matlab/octave` fuerza a que ambos deban acabar con `end`. La sangría (empezar los renglones más a la derecha) no es obligatoria pero favorece la legibilidad del código.

Lo único que puede resultar un poco singular de los bucles `for` es que la variable recorre típicamente un vector fila. Por ejemplo, si quisiéramos que se mostrasen los cuadrados de los enteros del 1 al 20, usaríamos:

```
1  for k = 1:20
```

```
2       disp(k^2)
3   end
```

Aunque no es tan habitual, también la variable puede recorrer matrices (bidimensionales) y como estas pueden considerarse un vector fila de columnas, tomará como valores cada una de las columnas.

Para utilizar `if` hay que decir algo de los operadores lógicos. Igual que en C/C++, las condiciones AND y OR se indican mediante `&&` y `||`. Por otro lado, $\sim$ es la negación. Por ejemplo, si en nuestra lista de cuadrados queremos saltarnos el 13 porque pensamos que da mala suerte, una manera de proceder es:

```
1   for k = 1:20
2       if k ~= 13
3           disp(k^2)
4       end
5   end
```

También existen `break` y `continue` con el significado habitual (salir totalmente de un bucle y pasar a la siguiente iteración). Así una alternativa para el código anterior es

```
1   for k = 1:20
2       if k == 13
3           continue
4       end
5       disp(k^2)
6   end
```

Cualquiera con experiencia programando sabe por qué aparece `k == 13` en la línea 2 en vez de `k = 13`. Un igual duplicado indica comparación y uno simple, asignación.

Se puede completar la estructura de `if` con `elseif`, para añadir nuevas condiciones, o con `else` para indicar la alternativa a todas ellas. En este ejemplo se pide un número del 1 al 10 y muestra diferentes mensajes dependiendo de su valor:

```
1   n = input('Introduce un número del 1 al 10');
2   if (n<1) || (n>10)
3       disp('No me has hecho caso: no está en el rango')
4   elseif mod(n,2)==0
5       disp('Es un número par')
6   elseif (n-5)^2==16
7       disp('Es un cuadrado impar')
8   elseif (n==3) || (n==5) || (n==7)
9       disp('Es primo')
10  else
11      disp('No es entero, ¿eres matemático?')
12  end
```

Seguro que conoces de antemano el uso del comando `while`. Ejecuta un

bloque mientras que la condición especificada se cumpla.

```
% Halla el primer k tal que sqrt(1)+...+sqrt(k) > n
n = 10;
k = 1;
s = 1;
while n > s
    k = k + 1;
    s = s + sqrt(k);
end
disp(k)
```

También existe, como en otros lenguajes de programación, un comando `switch`, que no veremos aquí.

9. Funciones

Vamos a crear una función (una subrutina) llamada `fibo` de forma que `fibo(n)` dé el `n`-ésimo número de Fibonacci aprovechando el *script* que habíamos creado antes.

En `matlab`, la pestaña de *Current Folder* con el botón derecho seleccionamos `function` y creamos el fichero `fibo.m`. También lo podemos hacer por medio del menú principal eligiendo `New>function`. En `octave`, seguiremos el menú `File > New > New function`.

Tanto en `matlab` como en `octave`, nos aparecerá un plantilla general para funciones. La de `matlab` es:

```
function [outputArg1,outputArg2] = fibo(inputArg1,inputArg2)
%FIBO Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

En realidad, podríamos no emplearla y crear el fichero por nuestra cuenta como cualquier otro *script*. Lo que es realmente importante, y distinto a los lenguajes de programación típicos, es que el nombre de la función debe coincidir con el nombre del fichero.

Nosotros solo queremos un argumento de salida, que llamaremos `Fn` y otro de entrada que llamaremos `n`. Por lo demás, copiamos el programa para calcular números de Fibonacci y asignamos el resultado final a la variable `Fn`, que será la que devuelve la función.

```
function Fn = fibo(n)
  % fibo(n) calcula el número de Fibonacci Fn
  % Elevamos a n
  F = [1,1;1,0]^n;
  % El resultado es el elemento 1,2 (o el 2,1)
  Fn = F(1,2);
end
```

La sangría, que no aparece en la plantilla, de nuevo solo es para favorecer la legibilidad del código. Es natural que en las funciones casi todas las líneas acaben con punto y coma porque no queremos que nos muestren resultados intermedios, sino que actúen como cajas negras que respondan a cada entrada devolviendo solo un resultado.

Ahora necesitamos un *script* que use la función. Creemos `lista_fibo.m` que dé los primeros 10 elementos de la sucesión de Fibonacci llamando a `fibo`. Para ello debe estar en el mismo directorio que el fichero `fibo.m` de la función. Un posible contenido de `lista_fibo.m` es, por hacer lo más simple,

```
1  for n = 1:10
2      disp(fibo(n))
3  end
```

Sí, seguro que a la mayoría nos parece que el formato del resultado es feo, en la siguiente sección veremos cómo mejorarlo.

Las funciones no están limitadas a argumentos y valores de salida escalares reales, pueden ser también matriciales, en particular vectoriales.

Con `return` se fuerza el retorno desde una función. Esto es útil cuando se utilizan condiciones.

10. Ficheros y entrada y salida

En primer lugar vamos a mejorar un poco la presentación de nuestra lista de números de Fibonacci con

```
1  for n = 1:10
2      disp(['El número de Fibonacci ' num2str(n) ' es ' num2str(fibo(n))])
3  end
```

que produce salidas del tipo `El número de Fibonacci ... es ...` con los puntos suspensivos sustituyendo el índice y el valor. La lógica para esto es que `num2str` transforma el número en cadena de caracteres (*number to string*) y las cadenas de caracteres se pueden yuxtaponer en `matlab`. Los corchetes son necesarios para no hacer un lío a `matlab/octave` con el fin de la cadena de caracteres.

Una manera alternativa con idéntico resultado que nos recordará más al `printf` de C, es:

```
1  for n = 1:10
2      fprintf('El número de Fibonacci %d es %d\n',n,fibo(n))
3  end
```

Como en C, el uso natural de `fprintf` no es imprimir en pantalla sino en ficheros. Para ello hay que que abrir un fichero, obteniendo su identificador y al final cerrarlo. Es prácticamente idéntico a C. Con el siguiente código:

```
1  fid = fopen('mifichero.dat','w');
2  fprintf(fid, 'Minitabla de números de Fibonacci\n');
3
4  for n = 1:5
5      fprintf(fid,'%d --> %d\n',n,fibo(n));
6  end
7  fclose(fid);
```

se genera un fichero conteniendo

```
Minitabla de números de Fibonacci
1 --> 1
2 --> 1
3 --> 2
4 --> 3
5 --> 5
```

Para los que tengan el C más oxidado, o lo desconozcan, aquí van algunas explicaciones. En la línea 1 se abre el fichero `mifichero.dat` en modo de escritura (la `w` es de *write*). Tras ello, `fid` identificará las posiciones de memoria correspondientes al fichero. Con `fprintf(fid,...)` se va escribiendo en ella. Cuando terminemos de escribir, en nuestro caso en la línea 7, hay que cerrar el fichero con `fclose`.

Para leer un fichero hay que usar `r` (la inicial de *read*) en vez de `w`. Esencialmente `fscanf` es el comando contrario que `fprintf`. En nuestro caso, queremos omitir la lectura del título, por lo que usaremos `fgetl`, que lee una línea. Una posibilidad para recuperar los datos de la lista a partir del fichero es:

```
1  fid = fopen('mifichero.dat','r');
2  titulo = fgetl(fid);
3  datos = fscanf(fid,'%d --> %d\n');
4  fclose(fid);
5  reshape(datos,2,5)'
```

A diferencia de lo que ocurriría en C, no es necesario un bucle, `fscanf` lee todos los datos como un vector. Tras ello, el contenido de `datos` es 1, 1, 2, 1, 3, 2, 4, 3, 5, 5, los diez datos en el orden leído. La instrucción `reshape` de la última línea es para darle la forma de matriz 5×2 que corresponde a la tabla. La trasposición que produce el apóstrofo es solo una cuestión estética, para que la tabla tenga la misma orientación que la del fichero de datos.

En realidad, `fscanf` admite un último argumento indicando las dimensiones de la salida y entonces el anterior código tiene la versión reducida equivalente:

```
fid = fopen('mifichero.dat','r');
titulo = fgetl(fid);
fscanf(fid,'%d --> %d\n',[2,5])'
fclose(fid);
```

Sabemos guardar una figura usando la interfaz. Con el comando `saveas` lo conseguiremos automáticamente por *software*. Por ejemplo, para guardar la superficie que habíamos usado en un fichero en formato `png` llamado `superficie.png`

```
[x,y] = meshgrid(linspace(-pi,pi,30), linspace(-1,1,20));
z = sin(x).*(1-y.^2);
surf(x,y,z)
saveas(gcf,'superficie.png')
```

Aquí `gcf` es la manera en la que `matlab/octave` indica la figura en curso. Supongo que son las iniciales de *get current figure* pero no estoy totalmente seguro.

Un ejemplo más avanzado es el siguiente, que crea tres imágenes consistentes en traslaciones de la función seno y asigna automáticamente los nombres `seno1.png`, `seno2.png` y `seno3.png`.

```
x = linspace(0,2*pi,100);
for k = 1:3
    plot(x, sin(x+k))
    nombre = sprintf('seno%d.png',k);
    saveas(gcf,nombre)
end
```

De nuevo, `sprintf` viene heredado de C.

Es raro que utilicemos en el curso programas como el anterior, e incluso la lectura y escritura en ficheros. No obstante, todo ello es muy útil si tenemos que tratar con cantidades relativamente grandes de figuras o datos.

Si tenemos un número reducido de datos pero te ha costado mucho generarlos, te resultará de interés guardar todas las variables que tienes activas en un momento dado (puedes curiosear cuáles son en la ventana del `workspace`). Por ejemplo, supongamos que `f` almacena el número `65537`, el mayor primo de Fermat conocido hasta la fecha, y quieres recordarlo en una futura sesión. Puedes guardar `f` en el fichero `mis_variables.mat`, junto con el resto de las variables mediante `save('mis_variables.mat')`. Si ahora te sales de la sesión y vuelves a entrar o escribes `clear`, que sirve para que se borren todas las variables, la variable `f` estará indefinida. Para volverla a la vida, basta con que ejecutes `load('mis_variables.mat')`.

11. Para saber más

Naturalmente, te surgirá más de una duda con los comandos que menos emplees o querrás aprender otros nuevos. Para lo primero, la ayuda de `matlab` es una buena fuente de información. Si escribes en la ventana de comandos (*command window*) `help` *comando*, te mostrará una ayuda en modo texto que terminará con un enlace a la página correspondiente de *Matlab Online Help*. Por ejemplo, con `help plot` obtenemos una larga parrafada conteniendo una descripción general, los usos más comunes y un ejemplo. Al final se incluyen dos enlaces: *Documentation for plot* y *Other functions named plot*. El primero es el más interesante y nos da la posibilidad de abrir los ejemplos como *live scripts*, una forma decorada de *scripts*, que podemos ejecutar y modificar.

La documentación está en el sitio web

`https://www.mathworks.com/help/matlab/`

A veces he encontrado alguna deficiencia en la traducción al español, por tanto suelo seleccionar el sitio web de Estados Unidos. Allí puedes explorar los temas que prefieras. Si te fijas, hay un enlace a la derecha en el anuncia la documentación en PDF. Si lo sigues, necesitarás entrar en tu cuenta. A este nivel, el fichero en el que estarás interesado es *MATLAB Primer*. Nota que es bastante largo, 164 páginas, por tanto no lo imprimas si no estás seguro de ello.

Si empleas `octave`, de nuevo `help` *comando* da acceso a la documentación en modo texto. Recuerda usar `more off` si prefieres que no te salga línea a línea cuando la salida es larga. La documentación está en el sitio web

`https://octave.org/doc/v6.1.0/`

Mi experiencia es que está menos cuidada que la de `matlab`, tanto es así que muchas veces he accedido a la ayuda de `matlab` para algunos comandos a pesar de estar empleando `octave`. Hay una versión PDF en `https://www.gnu.org/software/octave/octave.pdf` que seguro que no querrás imprimir porque consta de 1121 páginas.

Si quieres más información, en el siguiente apartado hay algunas referencias (no muy seleccionadas, a decir verdad). Una de ellas es un manual en PDF que, por la cantidad de años que lleva empleándose, podría tildarse de clásico. De todas formas, es muy fácil encontrar buen material de aprendizaje con búsquedas más o menos aleatorias en la red.

Referencias

[Gil06] A. Gilat. *Matlab: una introducción con ejemplos prácticos*. Reverté, 2006.

[Gri15] D. F. Griffiths. An Introduction to Matlab. `http://www.maths.dundee.ac.uk/ftp/na-reports/MatlabNotes.pdf`, 2015.

[HVZ12] E. Hernández, M. J. Vázquez, and M. A. Zurro. *Álgebra lineal y geometría*. Pearson, Madrid, 2012. 3a ed.

[Pra02] R. Pratap. *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers*. Oxford University Press, 2002.

[QS07] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[QSG14] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific computing with MATLAB and Octave*, volume 2 of *Texts in Computational Science and Engineering*. Springer, Heidelberg, 2014. Fourth edition [of MR2253397].