

APUNTES DE *MATLAB*
Fundamentos Matemáticos de la Ingeniería

Xabier Domínguez Pérez

A Coruña, 2006

Índice general

1. Primera sesión	3
1.1. Operaciones básicas	3
1.2. Las matrices en MATLAB	6
1.3. Ejercicios	13
2. Segunda sesión	15
2.1. Gráficas sencillas en MATLAB	15
2.2. Programación en MATLAB: <i>Scripts</i>	21
2.3. Ejercicios	25
3. Tercera sesión	27
3.1. Programación en MATLAB: las <i>functions</i>	27
3.2. Ejercicios	32
4. Cuarta sesión	34
4.1. Bucles <i>for... end</i>	34
4.2. Bucles <i>if... end</i> y <i>while... end</i>	37
4.3. Ejercicios	39
A. Soluciones a los ejercicios	40
A.1. Primera sesión	40
A.2. Segunda sesión	42
A.3. Tercera sesión	45
A.4. Cuarta sesión	50

Prólogo

Presentamos aquí un guión detallado de las prácticas de MATLAB que han formado parte, desde su puesta en marcha en el curso 2003/2004, de la asignatura *Fundamentos Matemáticos de la Ingeniería* de primer curso de Ingeniería Técnica en Obras Públicas, esp. Construcciones Civiles, de la Universidad de A Coruña.

Las prácticas se han venido estructurando en cuatro sesiones de 100 minutos. La mitad de ese tiempo, al menos, se dedica a la resolución de ejercicios relacionados con los aspectos de MATLAB que hayan sido tratados en cada sesión. Se incluyen, además de las explicaciones “teóricas”, los enunciados de los ejercicios, y en un apéndice las soluciones a todos ellos.

Es importante tener en cuenta que este material ha sido elaborado a partir de la realización de las mencionadas prácticas, y no al revés. En particular no hemos incluido más contenidos que los que ha dado tiempo a explicar y ejercitar razonablemente en el escaso tiempo disponible. Por supuesto, existe un gran número de fuentes que el lector puede consultar para continuar su aprendizaje o resolver una duda concreta, empezando por la propia ayuda de MATLAB.

Por otra parte, el carácter informal de estas notas y la introducción gradual y detallada de los contenidos pueden resultar convenientes para alguien que nunca ha usado el programa y quiere aprender, por su cuenta y rápidamente, sus características básicas.

Capítulo 1

Primera sesión

1.1. Operaciones básicas

MATLAB es una utilidad matemática, originalmente concebida para realizar cálculos numéricos con vectores y matrices (de ahí el nombre, *MATrix LABORatory*), aunque en las sucesivas versiones ha ido incorporando multitud de aplicaciones nuevas. En estas sesiones sólo podremos ver unas cuantas, pero se trata sobre todo de familiarizarse con el entorno del programa y ponerse en situación de ir aprendiendo cosas nuevas conforme se vayan necesitando.

Al abrir el programa nos encontramos una especie de sub-escritorio, es decir, una ventana en la que viven varias ventanas más pequeñas. Por ahora vamos a fijarnos en la ventana más a la derecha en la configuración estándar, que es la ventana de comandos. En ella introduciremos los comandos en modo directo, es decir, las instrucciones para las que queramos una respuesta inmediata.

Los dos ángulos que aparecen en la ventana de comandos

```
>>
```

se conocen como el *prompt* de MATLAB y nos indican que el programa está esperando nuestras instrucciones.

Para empezar, MATLAB se puede utilizar, por supuesto, como una calculadora. Si escribís lo siguiente

```
>> 234*485
```

y pulsáis Entrar, el programa os devuelve

```
ans =  
    113490
```

Ahora fijaos en que en la ventana de Workspace (“espacio de trabajo”) aparece la variable **ans** (de answer). MATLAB va guardando el resultado de

la última operación en esta variable. Si hacéis doble click sobre el icono que aparece al lado del nombre, aparece una ventana con el valor de la variable `ans`. Esta ventana es un editor, así que el valor se puede modificar.

Vemos que el asterisco `*` se utiliza para multiplicar. Si queremos calcular una potencia, por ejemplo 5^7 , lo haremos con el acento circunflejo `^`:

```
>> 5^7
ans =
    78125
```

Si repetís la operación de editar la variable `ans`, veréis que aparece almacenado este otro valor. El resultado de la última operación lo hemos perdido, o al menos ya no está almacenado en ninguna variable, aunque se podría recuperar copiando y pegando dentro de la propia ventana de comandos.

En las expresiones compuestas de varias operaciones, hay que tener en cuenta las reglas de prioridad, que nos indican qué operaciones se efectúan antes y cuáles después. Son las habituales: lo que primero se ejecuta es lo que hemos puesto entre paréntesis, en su caso, y en caso de tener varios paréntesis anidados, se van evaluando de dentro hacia fuera. Dentro de cada paréntesis (si es que los hay), lo primero que se evalúa son las potencias, después las multiplicaciones y divisiones, y finalmente las sumas y restas. Si hay varias operaciones del mismo nivel seguidas, se efectúan de izquierda a derecha. Por ejemplo, para obtener el valor de la expresión

$$\frac{2^{12} + \frac{1}{7}}{0.25 - 3(1 - \sqrt{3})}$$

podríamos teclear

```
>> (2^12+1/7)/(0.25-3*(1-3^0.5))
ans =
    1.6745e+003
```

Al igual que ocurre con las calculadoras científicas, la notación `1.6745e+003` significa $1'6745 \cdot 10^3$, es decir, $1674'5$.

MATLAB admite aritmética compleja. Por ejemplo si tecléis

```
>> (3-2i)*(4+5i)
```

el resultado es

```
ans =
    22.0000 + 7.0000i
```

Por supuesto podemos guardar el resultado de una operación en una variable nueva:

```
>> x=tan(pi/3)
x =
    1.7321
```

En una línea hemos hecho dos cosas: pedirle a MATLAB que evalúe esa expresión y guardar el resultado en la variable `x`, que aparece en el Workspace junto a `ans`.

Fijaos en que `pi` es una constante interna de MATLAB, es decir, tiene un valor asignado.

Aunque los resultados que vamos obteniendo aparezcan sólo con cuatro cifras decimales, MATLAB opera realmente con una precisión mucho mayor. Para que los resultados aparezcan con más cifras significativas basta teclear

```
>> format long
```

Si volvemos a pedirle el valor de `x`

```
>> x
nos devuelve ahora
x =
    1.73205080756888
```

MATLAB opera siempre con doble precisión, independientemente de cómo nos dé los resultados. Es importante tener en cuenta que la instrucción `format` no cambia la precisión de la máquina sino sólo el formato de salida de resultados.

Cuando MATLAB hace un cálculo, o simplemente se da por enterado de que hemos asignado un valor a una variable, nos responde con ese resultado en pantalla, como hemos podido ver hasta ahora. Para pedirle que no lo haga, escribimos punto y coma al final de la expresión y antes de pulsar enter

```
>> y=exp(i*pi);
```

(`exp` es la exponencial de base e .) MATLAB ha hecho este cálculo y ha guardado el resultado en la variable `y`, pero no nos contesta con el resultado. Sin embargo la variable `y` aparece en el Workspace, y podemos recuperar su valor editándola desde allí o bien simplemente tecleando

```
>> y
y =
-1.000000000000000 + 0.000000000000000i
```

Como veis a veces el formato largo es un poco incómodo. Para recuperar el formato por defecto escribimos

```
>> format short
>> y
y =
-1.0000 + 0.0000i
```

Hemos visto que, como es habitual en las ventanas de edición de texto, una vez se ha llenado la Command Window con nuestros comandos y las respuestas del programa, las líneas van desapareciendo por la parte superior de la ventana, desplazadas por las nuevas líneas de la parte inferior. Las líneas de la sesión actual que van quedando ocultas se pueden mostrar utilizando la barra móvil vertical a la derecha de la ventana. Si lo que queremos hacer es borrar todas las líneas de la Command Window, el comando que debemos utilizar es

```
>> clc
```

Vamos a fijarnos ahora en la ventana que aparece abajo a la izquierda, llamada Command History (Historia de comandos). Como su nombre indica, recoge todos los comandos que hemos ido introduciendo en la presente sesión (y en las últimas sesiones). El comando `clc` no tiene efecto sobre la Command History. Desde esta ventana se puede directamente arrastrar con el ratón una línea completa hasta la ventana de comandos y ejecutarla o modificarla una vez allí; también, si hacemos click con el botón derecho del ratón sobre un comando de la Command History, se abre un menú local que nos permite copiarla, ejecutarla, borrarla y otras opciones. Otra forma de recuperar comandos anteriores y en general, moverse por la historia reciente de comandos, es utilizar las teclas de cursor desde la Command Window.

1.2. Las matrices en MATLAB

Como antes comentábamos, una de las características de MATLAB es que está especialmente diseñado para trabajar con variables vectoriales y matriciales. Podemos hacer esta asignación

```
>> a=[2 3 0 1];
```

sin haberle indicado previamente al programa que `a` no es una variable escalar (es decir, una variable en la que almacenamos un solo número) sino una variable vectorial. De hecho en MATLAB no hay propiamente variables numéricas escalares ni vectoriales, sino matriciales (*arrays*): si miráis el Workspace en cualquier sesión de trabajo veréis que los números se van almacenando como matrices 1×1 . Análogamente, nuestra variable `a` es para MATLAB una matriz 1×4 .

Las matrices se introducen entre corchetes, separando las filas por `;` y los elementos de cada fila por comas o simplemente espacios.

```
>> A=[0 -1 3 2; 2 1 7 2; 3 0 6 3; 5 0 10 6]
```

```
A =
```

```
    0    -1     3     2
    2     1     7     2
    3     0     6     3
    5     0    10     6
```

Como no hemos puesto ; al final de la introducción de datos, MATLAB nos contesta con el valor de la variable. Tanto en la ventana de comandos como en la de variables, ya aparece colocada en forma matricial.

Las variables `a` y `A` no se interfieren (las podéis ver conviviendo en el Workspace) porque MATLAB distingue mayúsculas de minúsculas. Las variables pueden estar formadas por varios caracteres (como ya hemos visto con los ejemplos de `ans` y `pi`), pero el primero de ellos siempre ha de ser una letra.

Vamos a crear dos variables matriciales más (fijaos en que todas van apareciendo en la ventana del Workspace):

```
>> D=[2 -1 3 0 ; 0 0 1 5]
```

```
D =
```

```
    2    -1     3     0
    0     0     1     5
```

```
>> E=rand(4,4)
```

```
E =
```

```
    0.9501    0.8913    0.8214    0.9218
    0.2311    0.7621    0.4447    0.7382
    0.6068    0.4565    0.6154    0.1763
    0.4860    0.0185    0.7919    0.4057
```

(El comando `rand` crea una matriz del tamaño especificado, en este caso 4×4 , formada por números aleatorios entre 0 y 1.)

Las operaciones de suma de matrices y producto de un escalar por una matriz se realizan directamente, sin necesidad de ir componente a componente:

```
>> A+E
```

```
ans =
```

```
    0.9501   -0.1087    3.8214    2.9218
    2.2311    1.7621    7.4447    2.7382
    3.6068    0.4565    6.6154    3.1763
    5.4860    0.0185   10.7919    6.4057
```

```
>> 3.5*E
ans =
    3.3255    3.1195    2.8749    3.2263
    0.8090    2.6673    1.5565    2.5837
    2.1239    1.5976    2.1540    0.6169
    1.7009    0.0648    2.7718    1.4200
```

Por supuesto, si intentamos sumar dos matrices de tamaños distintos obtendremos un mensaje de error

```
>> A+D
??? Error using ==> +
Matrix dimensions must agree.
```

Igual de fácil resulta multiplicar matrices

```
>> D*E
ans =
    3.4896    2.3899    3.0444    1.6342
    3.0368    0.5490    4.5751    2.2048
```

D*E es el producto ordinario de las matrices D y E. Para que tenga sentido, como sabéis, el número de columnas del primer factor tiene que coincidir con el número de filas del segundo

```
>> E*D
??? Error using ==> *
Inner matrix dimensions must agree.
```

Tiene sentido definir otro “producto” de matrices, el que se hace componente a componente, como la suma. Para multiplicar en este sentido dos matrices es necesario que tengan el mismo tamaño: cada elemento de la matriz resultado se obtiene multiplicando los elementos que ocupan esa misma posición en las dos matrices. Vamos a crear por ejemplo la matriz

```
>> F=10*rand(2,4)
F =
    9.3547    4.1027    0.5789    8.1317
    9.1690    8.9365    3.5287    0.0986
```

y multiplicar elemento a elemento las matrices D y F, que tienen las mismas dimensiones. Las operaciones “elemento a elemento” se indican anteponiendo un punto al símbolo correspondiente. Por ejemplo

```
>> D.*F
ans =
    18.7094   -4.1027    1.7367         0
         0         0    3.5287    0.4931
```

La potencia n -ésima de una matriz cuadrada es el producto matricial de la matriz por sí misma n veces:

```
>> A^4
ans =

    2419    -204    5342    3030
    5343   -452   11838    6702
    5457   -465   12093    6852
    9870   -840   21870   12391
```

También se puede plantear la potencia n -ésima elemento a elemento:

```
>> F.^4
ans =
  1.0e+003 *
    7.6581    0.2833    0.0001    4.3724
    7.0680    6.3778    0.1550    0.0000
```

o elevar una matriz a otra, elemento a elemento

```
>> F.^D
ans =
    87.5104    0.2437    0.1940    1.0000
    1.0000    1.0000    3.5287    0.0000
```

o la división

```
>> F./D
Warning: Divide by zero.
ans =
    4.6773   -4.1027    0.1930     Inf
     Inf      Inf    3.5287    0.0197
```

Aquí veis que cuando dividimos por cero MATLAB no da error sino que devuelve `Inf` (infinito).

Las funciones elementales (trigonométricas, exponencial, logaritmo, etc.) se pueden aplicar a las matrices, componente a componente, sin necesidad de anteponer un punto:

```
>> sin(F)
ans =
    0.0700   -0.8198    0.5471    0.9617
    0.2530    0.4691   -0.3775    0.0985
```

```
>> exp(D)
ans =
    7.3891    0.3679   20.0855    1.0000
    1.0000    1.0000    2.7183  148.4132
```

Las operaciones “elemento a elemento” resultan útiles en muchas ocasiones en las que queremos hacer el mismo cálculo simultáneamente sobre diversos valores numéricos. Por ejemplo, para evaluar la función $f(x) = \tan^2(\ln x)$ en los valores $x = 1, 1.5, 2, 3, 5$ basta hacer

```
>> x=[1 1.5 2 3 5];
>> y=tan(log(x)).^2
y =
           0    0.1843    0.6900    3.8339   669.0486
```

Tanto los cinco valores de la x como las cinco evaluaciones de la función los hemos almacenado en sendas variables vectoriales.

Para trasponer matrices utilizamos el apóstrofe. Por ejemplo:

```
>> B=A'
B =
     0     2     3     5
    -1     1     0     0
     3     7     6    10
     2     2     3     6
```

Hay que hacer una observación aquí: Si la matriz con la que trabajamos es de números complejos, por ejemplo la matriz 4×1 siguiente

```
>> C= [ 1-i ; -i; 0; 4-i];
```

al teclear C' no nos da exactamente la traspuesta

```
>> C'
ans =
1.0000 + 1.0000i    0 + 1.0000i    0    4.0000 + 1.0000i
```

sino la traspuesta conjugada: se traspone la matriz y se calculan los conjugados de todos sus elementos. Esto es debido a que cuando se trabaja con matrices complejas, la operación combinada trasposición-conjugación es muy común. Si queremos, en el caso complejo, simplemente trasponer, tenemos que escribir

```
>> C.'
ans =
1.0000 - 1.0000i    0 - 1.0000i    0    4.0000 - 1.0000i
```

Se puede “extraer” un elemento de una matriz. Por ejemplo, el elemento de la fila 2 y columna 4 de A lo recuperamos tecleando

```
>> A(2,4)
ans =
     2
```

Un rango de filas, o de columnas, se indica utilizando los dos puntos :
Por ejemplo, los elementos de la matriz A que están dentro de la fila 2, entre las columnas 1 y 3 inclusive, se extraen así de la matriz:

```
>> A(2,1:3)
ans =
     2     1     7
```

Fijaos en que `ans` es una variable 1×3 . Los elementos de A que están dentro de la columna 3, entre las filas 2 y 4 inclusive se extraen así:

```
>> A(2:4,3)
ans =
     7
     6
    10
```

y ahora el resultado es 3×1 (lógico...). Si queremos sacar de A una fila o columna entera podemos poner delimitadores `1:4` (porque la matriz es 4×4) o no poner ninguno:

```
>> A(:,4)
ans =
     2
     2
     3
     6
```

También podemos sacar de una matriz elementos no adyacentes. El segundo y cuarto elementos de la fila 3 de A:

```
>> A(3,[2 4])
ans =
     0     3
```

Si definimos delimitadores antes y después de la coma, lo que obtenemos son submatrices. Por ejemplo, la submatriz 3×3 de A obtenida al intersecar las filas $\{2,3\}$ con las columnas $\{2,3,4\}$ sería

```
>> A(2:3,2:4)
ans =
     1     7     2
     0     6     3
```

Las submatrices pueden estar formadas por elementos no adyacentes. La submatriz de los elementos de A que están en las filas 1 ó 4 y en las columnas 2 ó 4 sería

```
>> A([1 4],[2 4])
ans =
    -1     2
     0     6
```

Se le puede añadir una fila a una matriz

```
>> u=[3 4 1 5];
>> G=[A;u]
G =
     0    -1     3     2
     2     1     7     2
     3     0     6     3
     5     0    10     6
     3     4     1     5
```

o bien una columna, de esta otra forma:

```
>> v=[1; 0; 2; -1];
>> H=[A v]
H =
     0    -1     3     2     1
     2     1     7     2     0
     3     0     6     3     2
     5     0    10     6    -1
```

Existe una cosa un poco extraña en MATLAB que es la matriz vacía []. Para quitarle a H la fila 3, por ejemplo, la igualo a la matriz vacía:

```
>> H(3,:)=[]
H =
     0    -1     3     2     1
     2     1     7     2     0
     5     0    10     6    -1
```

Para quitarle al resultado las columnas 3 y 5, escribo

```
>> H(:,[3 5])=[]
H =
     0    -1     2
     2     1     2
     5     0     6
```

MATLAB tiene comandos para crear matrices predeterminadas. Por ejemplo, la matriz identidad $n \times n$ se genera con `eye(n)`

```
>> eye(5)
ans =
```

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

Una matriz toda de unos se genera con `ones(m,n)`; por ejemplo

```
>> ones(4,3)
ans =
```

```
1 1 1
1 1 1
1 1 1
1 1 1
```

y una matriz toda de ceros, con `zeros(m,n)`; por ejemplo

```
>> zeros(1,7)
ans =
```

```
0 0 0 0 0 0 0
```

1.3. Ejercicios

1. Calcular módulo y argumento del número complejo

$$\frac{(3i - 1)^5}{5 + i}$$

Nota: el comando `abs` da el valor absoluto de un número real, o bien el módulo de un número complejo. El comando `angle` da el argumento en radianes de un número complejo. Como siempre, se pueden aplicar a matrices.

2. Comprobar que

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = e$$

de la siguiente forma: Crear una variable vectorial `n` que contenga los elementos

```
1 10 100 500 1000 2000 4000 8000
```

Seguidamente crear un nuevo vector y cuyas componentes sean los valores correlativos de la sucesión en los índices de `n`. Comparar los valores de las componentes de `y` con el auténtico valor de `e`.

3. Definir las siguientes matrices:

$$\mathbf{A} = \begin{pmatrix} 2 & 6 \\ 3 & 9 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} -5 & 5 \\ 5 & 3 \end{pmatrix}$$

Crear la siguiente matriz (que tiene sobre la diagonal las matrices \mathbf{A} , \mathbf{B} , \mathbf{C}) sin introducir elemento a elemento:

$$\mathbf{G} = \begin{pmatrix} 2 & 6 & 0 & 0 & 0 & 0 \\ 3 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 5 \\ 0 & 0 & 0 & 0 & 5 & 3 \end{pmatrix}$$

Realizar sobre \mathbf{G} las siguientes operaciones, guardando todos los resultados en variables distintas:

- (a) Borrar la última fila y la última columna de \mathbf{G} .
 - (b) Extraer la primera submatriz 4×4 de \mathbf{G} .
 - (c) Extraer la submatriz $\{1, 3, 6\} \times \{2, 5\}$ de \mathbf{G} .
 - (d) Reemplazar $\mathbf{G}(5, 5)$ por 4.
4. (Resolución de sistemas de ecuaciones lineales.) El comando `inv` calcula la matriz inversa de una matriz regular. Por lo tanto, el sistema de ecuaciones lineales $\mathbf{Ax} = \mathbf{b}$ puede resolverse simplemente mediante

```
>> inv(A)*b
```

Sin embargo, hay una forma de hacer que MATLAB calcule la solución de $\mathbf{Ax} = \mathbf{b}$ utilizando el método de Gauss (reducción del sistema mediante operaciones elementales de fila). Este método es preferible al anterior ya que el cálculo de la inversa involucra más operaciones y es más sensible a errores numéricos. Se utiliza la llamada división matricial izquierda `\`

```
>> A\b
```

Probar los dos métodos con el sistema siguiente:

$$\begin{cases} 2x - y + 3z = 4 \\ x + 4y + z = 2 \\ 6x + 10y + 3z = 0 \end{cases}$$

Capítulo 2

Segunda sesión

En esta sesión vamos a aprender a producir algunas gráficas con MATLAB y también empezaremos a escribir y ejecutar programas.

2.1. Gráficas sencillas en MATLAB

La forma más “artesanal” de generar gráficas 2D en MATLAB es usando el comando `plot`. Vamos a representar, por ejemplo, la función $f(x) = \sin x - \cos^2 x$ en el intervalo $[-5, 5]$. Primero tenemos que crear dos variables vectoriales: una, que llamaremos por ejemplo `x`, y que almacenará los valores de $x \in [-5, 5]$ en los que evaluaremos la función f , y otra, que podemos llamar `y`, en el que se almacenarán las evaluaciones de f en esos puntos. En definitiva, se trata simplemente de crear una tabla de valores.

Habitualmente los valores de x se escogen equiespaciados entre los dos extremos del intervalo. Hay dos formas de hacer esto: indicando el número de puntos o indicando la distancia entre dos puntos consecutivos. Por ejemplo, tecleando

```
>> x=linspace(-5,5,20);
```

almacenamos en la variable `x` 20 valores distribuidos regularmente entre -5 y 5 . (Comprobadlo editando la variable en el Workspace.) Si hacemos en cambio

```
>> x=-5:0.5:5;
```

la variable `x` almacenará valores entre -5 y 5 , cada uno a una distancia 0.5 del siguiente. (Si queremos que el paso sea de 1 en vez de 0.5 , en lugar de `x=-5:0.5:5`; podríamos poner simplemente `x=-5:1:5`; de forma similar a cuando determinábamos un rango de filas o columnas en una matriz.)

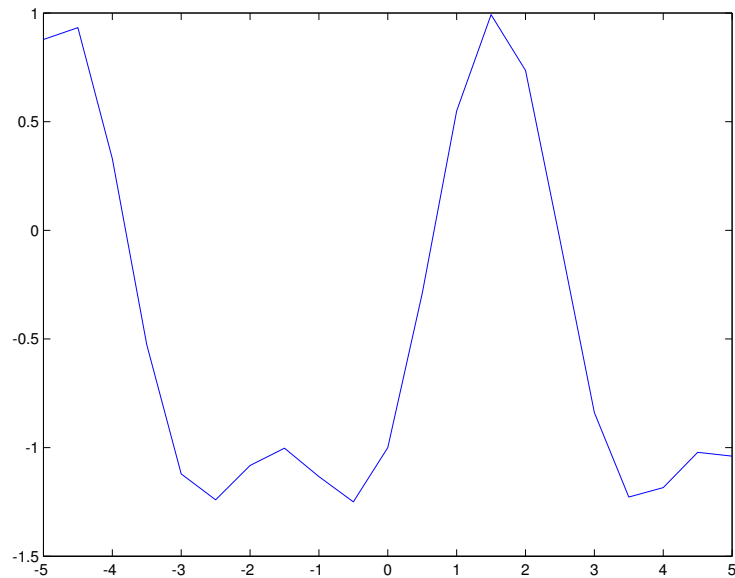
Nos quedamos por ejemplo con este último valor de `x`, y evaluamos la función en esos puntos:

```
>> x=-5:0.5:5;
>> y=sin(x)-cos(x).^2;
```

Notar que $\cos(x)$ es una matriz fila y queremos elevarla al cuadrado en el único sentido posible, es decir, elemento a elemento; de ahí que antepongamos un punto al carácter \wedge . Ahora sólo queda pedirle al programa que represente los puntos (x,y) en un sistema de ejes coordenados. Esto se hace simplemente escribiendo

```
>> plot(x,y)
```

Se abre una ventana gráfica con la representación de la función.



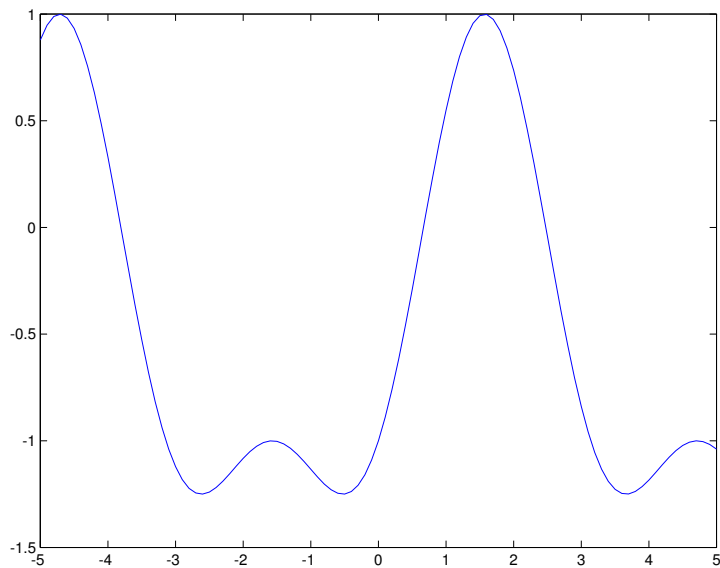
Observamos que la gráfica no es muy satisfactoria: es una línea poligonal. Lo que hace el comando `plot` es pintar los puntos (x,y) que hemos creado y unirlos con segmentos de línea recta. Para que la gráfica aparezca más suave, por lo tanto, hay que tomar los puntos de x más cercanos unos de otros. Por ejemplo

```
>> x=-5:0.1:5;
```

crea un *array* con puntos desde -5 hasta 5 espaciados 0.1 (fijaos en el *Workspace*). Evaluando de nuevo la función en los puntos de x

```
>> y=sin(x)-cos(x).^2;
>> plot(x,y)
```

se crea una gráfica más suave.



Esta nueva curva sustituye a la anterior en la ventana gráfica. Si queremos conservarla, podemos guardarla de la forma habitual, desde la propia ventana gráfica (*File>>Save* o *Save as...*), o haciendo click sobre el icono del diskette). Como en otras aplicaciones, hay una carpeta donde el programa guardará por defecto todos los archivos a menos que le indiquemos otra cosa. Esa carpeta se llama “Current Directory” y su contenido es accesible desde la vista normal del escritorio de MATLAB, haciendo click en la pestaña correspondiente. Al iniciar el programa el Current Directory se sitúa en una carpeta llamada *work*, que cuelga de la carpeta donde está instalado MATLAB, pero se puede cambiar utilizando los botones de la parte superior de la ventana. Las gráficas generadas por MATLAB se guardan como archivos *.fig*, un formato propio del programa, aunque también se pueden convertir a *.jpg*, a *.eps* y otros (*File>>Export...*).

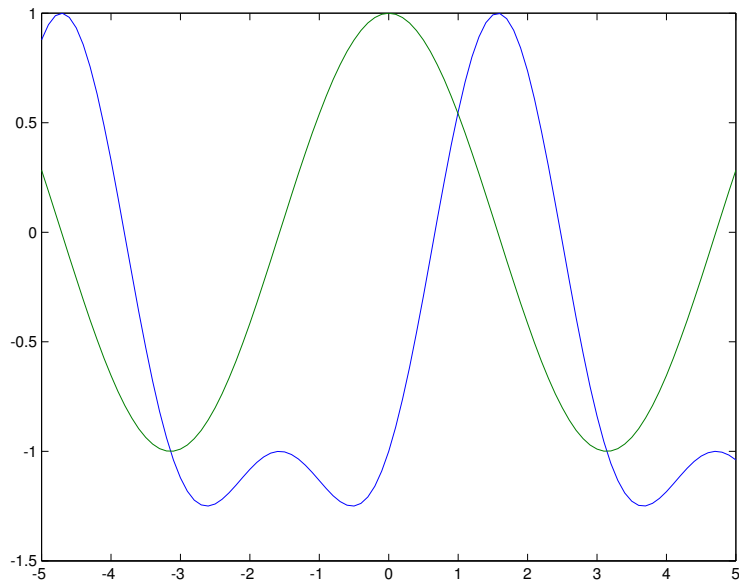
Se pueden pintar varias gráficas superpuestas. Por ejemplo, definimos los valores de la función coseno sobre la malla de puntos *x* ya creada:

```
>> z=cos(x);
```

y pintamos las dos gráficas a la vez (ver gráfica en página siguiente), simplemente escribiendo

```
>> plot(x,y,x,z)
```

Puede ser que queramos pintar sólo una serie de puntos. Por ejemplo, si nos interesa representar gráficamente los elementos de la sucesión $1/n$ desde $n = 1$ hasta 10, la secuencia de comandos



```
>> n=1:10;
>> m=1./n;
>> plot(n,m)
```

produce una gráfica continua que seguramente no nos viene bien. En este caso basta añadirle la opción `'.'` como un argumento más del comando `plot`:

```
>> plot(n,m, '.')
```

(ver gráfica en página siguiente). Hay multitud de opciones que controlan la apariencia de la gráfica. Por ejemplo,

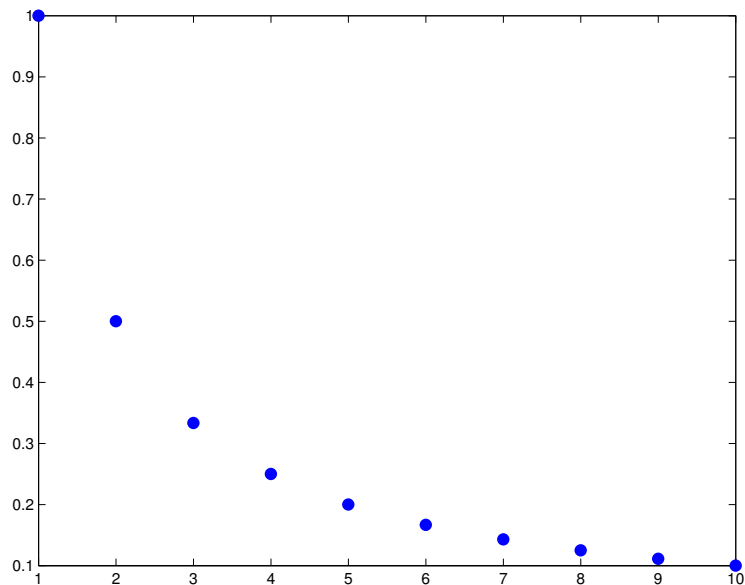
```
>> plot(n,m, 'o')
```

sustituye los puntos por pequeños círculos. Si tecleáis

```
>> help plot
```

os aparecerá en pantalla una lista de opciones disponibles para este comando. `help` se puede usar para obtener información sobre cualquier comando.

También podéis mejorar o modificar la gráfica desde la propia ventana gráfica, sin introducir comandos desde la Command Window. Desde los menús *Edit* e *Insert*, y haciendo click sobre los elementos de la gráfica que nos interesen, se puede modificar el color de la línea, su grosor, el aspecto de los ejes, ponerle etiquetas a los ejes *X* e *Y*, darle un título a la gráfica, insertar líneas, flechas, texto...

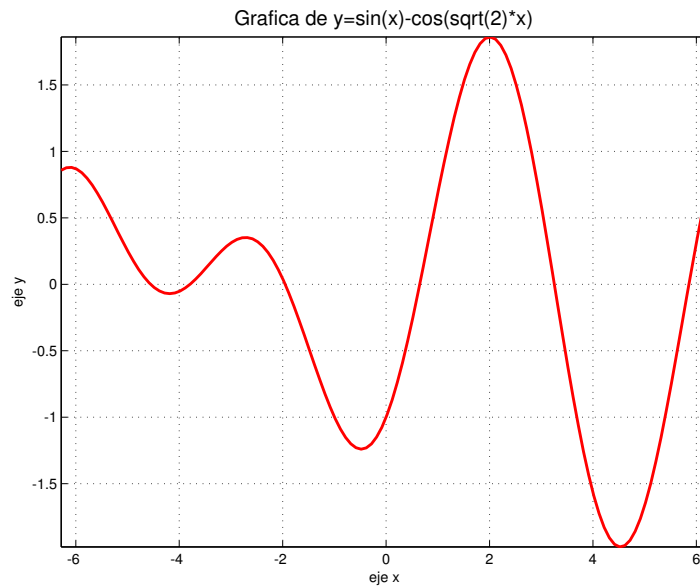


Por supuesto, todas estas operaciones se pueden hacer desde la Command Window, pero esto es más complicado porque necesitamos acordarnos del comando que hace cada cosa. Por ejemplo los siguientes comandos

```
>> x=-2*pi:.1:2*pi;
>> y=sin(x)-cos(sqrt(2)*x);
>> plot(x,y,'r','linewidth',2)
>> axis tight
>> grid on
>> xlabel('eje x')
>> ylabel('eje y')
>> title('Grafica de y=sin(x)-cos(sqrt(2)*x)','FontSize',14)
```

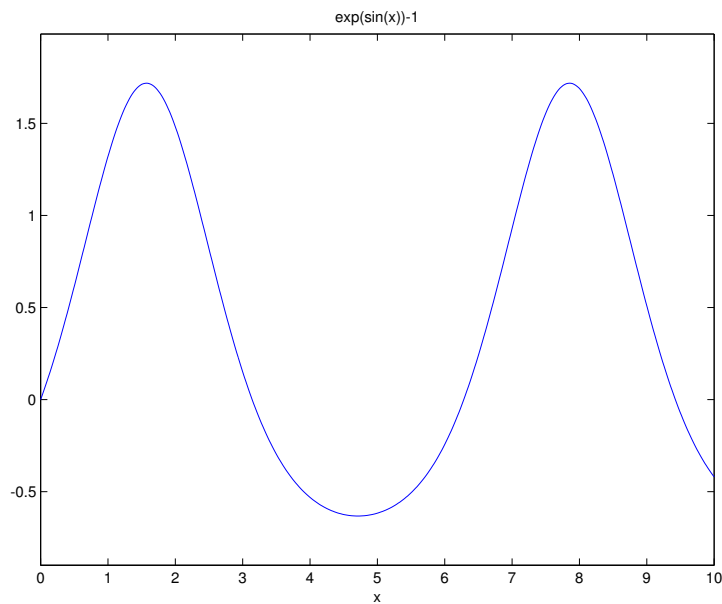
dan lugar a la gráfica reproducida en la página siguiente.

La ventaja de aprender a editar una gráfica con comandos en vez de desde la ventana gráfica es que los comandos se pueden programar. (Veremos enseguida cómo hacerlo.) La edición de una gráfica a golpe de ratón es mucho más intuitiva pero en muchos casos resulta cómodo almacenar el proceso de edición en una secuencia de comandos, para no tener que guardar la gráfica, o si tenemos que producir varias gráficas parecidas.



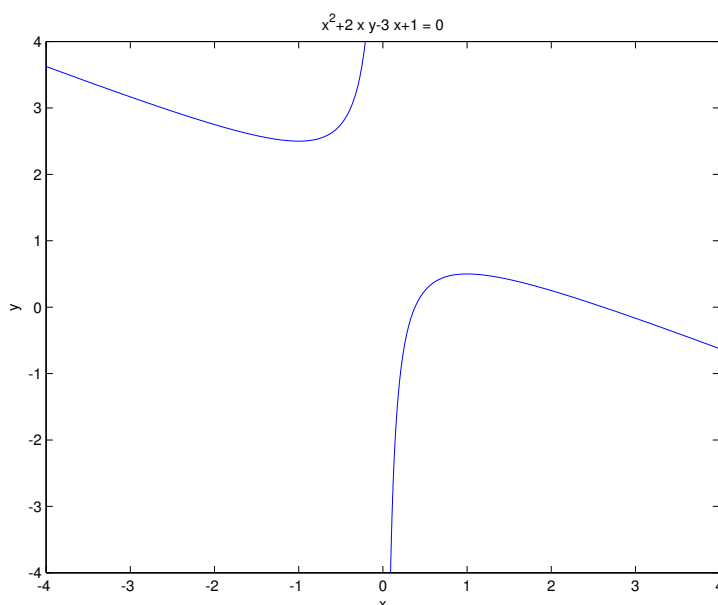
La instrucción `plot` es muy versátil, pero si queremos producir una gráfica estándar que represente una sola curva sin complicarnos generando una tabla de valores, disponemos del comando `ezplot`, que traza la curva correspondiente a una expresión funcional que se introduce como una cadena de caracteres. Por ejemplo: para dibujar la función $f(x) = \exp(\sin(x)) - 1$ en el intervalo $[0, 10]$ basta teclear

```
>> ezplot('exp(sin(x))-1',[0,10])
```



'exp(sin(x))-1' es una cadena de caracteres que MATLAB ha de interpretar como la expresión analítica de una función. Las cadenas de caracteres (*strings*) han de introducirse entre apóstrofes. Una de las ventajas de `ezplot` es que también puede utilizarse para dibujar gráficas de curvas definidas implícitamente (curvas en el plano). Por ejemplo, representamos la cónica de ecuación $x^2 + 2xy - 3x + 1 = 0$ (el conjunto de puntos (x, y) del plano que satisfacen esa ecuación):

```
>> ezplot('x^2+2*x*y-3*x+1', [-4 4 -4 4])
```



donde los cuatro números indican el recuadro del plano donde está el trozo de gráfica que nos interesa, en este caso $-4 \leq x \leq 4$, $-4 \leq y \leq 4$.

2.2. Programación en MATLAB: *Scripts*

Un *script* no es más que un conjunto de comandos concatenados que podemos ejecutar siempre que nos apetezca, sin teclearlos cada vez.

Vamos a introducir en un *script* la secuencia de comandos que producía la gráfica de la función $\sin x - \cos(\sqrt{2}x)$ de arriba. En el menú *File* del escritorio de MATLAB escogemos el comando *New* y el subcomando *M-file*. Se abre una ventana en la que podemos teclear o copiar los comandos que queremos que formen el programa. Vamos copiando sucesivamente, desde la *Command Window* o la *Command History*, las diferentes líneas que antes tecleamos y ejecutamos una a una:

```

x=-2*pi:.1:2*pi;
y=sin(x)-cos(sqrt(2)*x);
plot(x,y,'r','linewidth',2)
axis tight
grid on
xlabel('eje x')
ylabel('eje y')
title('Grafica de y=sin(x)-cos(sqrt(2)*x)','FontSize',14)

```

Una vez hecho esto guardamos el programa (menú *File*, comando *Save as...*) dándole un nombre, por ejemplo *grafica*. MATLAB le añade automáticamente una extensión *.m* (los programas se guardan como *M-files*, un tipo de archivo propio de MATLAB). En la ventana del Current Directory aparece el nuevo archivo *grafica.m*. Ahora, si en la ventana de comandos tecleamos

```
>> grafica
```

los comandos del programa se ejecutan sucesivamente, y se genera la gráfica. Es como si hubiésemos creado un nuevo comando de MATLAB, el comando *grafica*.

Por supuesto los programas se pueden modificar. Por ejemplo, vamos a introducir una línea de comentario al principio del programa para explicar lo que hace. (Si ya no tenéis activa la ventana de *grafica.m*, podéis acceder a ella en el menú *File>>Open*, como hacemos habitualmente en las aplicaciones para Windows.) Un comentario se introduce siempre detrás del símbolo *%*. MATLAB simplemente ignora lo que haya detrás de este símbolo. Así que hacemos click al principio de la línea 1 y escribimos como en un procesador de textos (el texto ya escrito se va desplazando)

```
% Dibuja la grafica de una funcion
```

le damos a Entrar y guardamos los cambios. Esta explicación aparece en el Current Directory (no inmediatamente sino la próxima vez que MATLAB tenga que reconstruir esta ventana) al lado del nombre del programa, lo que nos facilita identificarlo entre otros muchos que podemos haber guardado.

Veamos otro ejemplo. Consideramos la siguiente sucesión de números reales:

$$\frac{(-1)^{k-1}}{k} \quad (k = 1, 2, 3, \dots)$$

Resulta que la suma de los primeros términos de la sucesión, es decir,

$$\frac{(-1)^{1-1}}{1} + \frac{(-1)^{2-1}}{2} + \frac{(-1)^{3-1}}{3} + \dots + \frac{(-1)^{n-1}}{n}$$

es una aproximación de $\ln 2$, tanto mejor cuantos más términos tomemos. Vamos a preparar un *script* que calcule la suma de los 1000 primeros términos

de la sucesión, es decir,

$$\frac{(-1)^{1-1}}{1} + \frac{(-1)^{2-1}}{2} + \frac{(-1)^{3-1}}{3} + \dots + \frac{(-1)^{999}}{1000}$$

y que además compare esa suma con el “verdadero” valor de $\ln 2$.

Necesitaremos usar el comando `sum`, que calcula la suma de todos los elementos de una variable vectorial, por ejemplo

```
>> a=[2 3.5 0 -1];
>> sum(a)
ans =
    4.5000
```

Antes de seguir, teclearemos

```
>> format long
```

porque nos van a venir bien los resultados en doble precisión.

Siguiendo los pasos que ya conocemos abrimos un nuevo *M-file* y escribimos en él las líneas de comando

```
% Calcula la suma de 1000 terminos de la serie de ln(2)
k=1:1000;
s=(-1).^ (k-1) ./k;
suma=sum(s)
vreal=log(2)
difa=abs(suma-vreal)
```

Si ahora guardamos este programa como `sumaln` y a continuación tecleamos en la Command Window

```
>> sumaln
```

el resultado debería ser

```
suma =
    0.69264743055982
vreal =
    0.69314718055995
difa =
    4.997500001230337e-004
```

Vamos a hacer un poco más interactivo este *script*, adaptándolo para que calcule un número variable de sumandos de la expresión de arriba. Abrimos de nuevo `sumaln.m` (*File>>Open...*) y lo modificamos así:

```

% Calcula la suma de n terminos de la serie de ln(2)
k=1:n;
s=(-1).^(k-1)./k;
suma=sum(s)
vreal=log(2)
difa=abs(suma-vreal)

```

Lo guardamos de nuevo, y lo ejecutamos, teniendo en cuenta que antes de llamarlo hay que darle un valor a *n*, la cantidad de términos que queremos sumar. Por ejemplo

```

>> n=100;
>> sumaln
suma =
    0.68817217931020
vreal =
    0.69314718055995
difa =
    0.00497500124975
>> n=10000;
>> sumaln
suma =
    0.69309718305996
vreal =
    0.69314718055995
difa =
    4.999749998702008e-005

```

Ahora le echaremos un vistazo al Workspace. Todas las variables que intervienen en nuestro programa están allí, con el último valor que hayan tomado al ejecutar `sumaln`. (Entre ellas están las “monstruosas” variables *k* y *s*, ocupando un buen trozo de memoria). Hay varios tipos de variables: unas cuyo valor hemos introducido desde la ventana de comandos (en este caso sólo *n*), otras cuyo valor se nos devuelve como resultado de la ejecución (*suma*, *vreal*, *difa*), y otras que se han generado dentro del programa simplemente para hacer cálculos (*k* y *s*). Si el programa es un *script*, como es el caso, MATLAB no distingue entre unas y otras: independientemente de que su valor nos interese o no, todas se incorporan al Workspace, porque ejecutar el *script* es equivalente a teclear y ejecutar sucesivamente cada una de sus líneas desde la Command Window. Esto no es bueno, sobre todo si nuestro programa es un poco complicado e involucra muchas variables: el Workspace se convertiría en algo inmanejable.

Las variables que aparecen en el Workspace se denominan *variables del espacio de trabajo base*. Para que no se nos llene el Workspace de variables inútiles, tendremos que sustituir nuestro *script* por una *function*, que es un

tipo de programa que tiene su propio espacio de trabajo. Al trabajar con *functions*, distinguiremos entre variables de entrada, variables de salida y variables internas al programa; esto se corresponde con el hecho de que en casi cualquier programa interesante, unos datos de entrada se procesan para obtener datos de salida, y no nos importa prescindir de los datos intermedios que genera el propio proceso. Veremos cómo se programan *functions* en la próxima sesión.

2.3. Ejercicios

1. Hay toda una gama de comandos `ez...` que permiten hacer rápidamente gráficas en dos y tres dimensiones, en coordenadas polares, de curvas en el plano y el espacio... Los principales son : `ezplot`, `ezpolar` (gráficas en coordenadas polares), `ezplot3` (curvas en el espacio), `ezcontour` (dibuja líneas de nivel de superficies), `ezsurf` (superficies). Probad por ejemplo:

```
>> ezpolar('1 + cos(t)')
>> ezplot3('cos(t)', 'sin(t)', 't', [0,6*pi])
>> ezcontour('x*exp(-x^2- y^2)')
>> ezcontourf('x*exp(-x^2-y^2)')
>> ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)', [-6*pi,6*pi])
```

Podéis obtener una explicación del funcionamiento de estos comandos, con algunos ejemplos, tecleando `help` seguido del nombre del comando.

2. Representar gráficamente la función

$$f(x) = \begin{cases} 2 + \operatorname{sen} x & (-10 \leq x \leq -5) \\ e^x & (-5 < x < 2) \\ \ln(x^2 + 1) & (2 \leq x \leq 10) \end{cases}$$

Investigar el uso de los comandos de edición directa en la ventana gráfica para intentar darle un mejor aspecto al resultado.

3. Preparar un *script* `solucion.m` que resuelva el siguiente sistema de ecuaciones:

$$\begin{cases} 5x + 2ry + rz = 2 \\ 3x + 6y + (2r - 1)z = 3 \\ 2x + (r - 1)y + 3rz = 5 \end{cases}$$

para un valor arbitrario del parámetro r que introduciremos antes de ejecutar el programa, de esta forma:

```
>> r=10;
>> solucion
```

4. (Un bucle `for`.) Es posible (y recomendable) hacer el cálculo de la suma de, pongamos, 10000 términos de la sucesión $(-1)^{n-1}/n$ sin necesidad de crear variables vectoriales de 10000 componentes. La forma habitual de hacerlo es mediante un bucle `for`.

Un bucle `for` es un conjunto de líneas de programa comprendidas entre dos líneas parecidas a éstas:

```
for k=1:10
:
end
```

Las líneas de programa comprendidas entre estas dos se ejecutarán sucesivamente 10 veces seguidas, y en cada una de ellas la variable `k` tomará el valor correspondiente, desde 1 hasta 10, en este caso.

- Para entender cómo funciona, crear y ejecutar un *script* con las siguientes líneas

```
for a=1:5
    a^2
end
```

y razonar la respuesta que se obtiene.

- Desde la Command Window ejecutar el comando

```
>> clear
```

que borrará las variables del Workspace.

- A continuación crear el siguiente *script*

```
% Calcula la suma de 10000 terminos de la serie de ln(2)
suma=0;
for k=1:10000
    suma=suma+(-1)^(k-1)/k;
end
suma
vreal=log(2)
difa=abs(suma-vreal)
```

Guardarlo por ejemplo como `sumaln2` y ejecutarlo. Intentar razonar, línea a línea, cómo funciona el programa. Comprobar que en el Workspace no aparece ahora ninguna variable vectorial. Adaptar el *script* como antes, para un número arbitrario `n` de sumandos.

Capítulo 3

Tercera sesión

3.1. Programación en MATLAB: las *functions*

En la sesión anterior aprendimos a almacenar una secuencia de comandos en un *script* para ejecutarlos sucesivamente siempre que lo necesitáramos, sin necesidad de teclearlos todos cada vez. Ejecutar un *script* es totalmente equivalente a ejecutar desde la Command Window cada una de sus líneas de comando. En particular, todas las variables que se creen dentro del *script* se incorporarán al Workspace y permanecerán almacenadas por si necesitamos usarlas más adelante.

Las variables que aparecen en el Workspace se denominan *variables globales* o *variables del espacio de trabajo base*. Hasta ahora son las únicas variables que nos hemos encontrado. Si trabajamos siempre desde la Command Window, o mediante *scripts*, MATLAB no puede averiguar qué variables nos conviene conservar y cuáles usamos simplemente como variables auxiliares.

Sin embargo, la mayor parte de las tareas que vayamos a programar se podrán describir como el procesamiento de unos datos de entrada para obtener datos de salida, y no nos importará prescindir de los datos intermedios que genere el proceso. Si distinguimos entre estos tres tipos de variables, y si además conseguimos que se “limpien” automáticamente del Workspace las variables que no nos interese conservar, programaremos de forma más sistemática, ahorraremos memoria y evitaremos la acumulación de información inútil.

Para ello disponemos de un tipo de programa diferente a los *scripts*, que se denomina *function*. Las *functions* se caracterizan por admitir argumentos de entrada y salida y por disponer de su propio espacio de trabajo.

Vamos a generar un *script* y convertirlo en una *function* para entender mejor estas nuevas posibilidades. Programaremos una operación elemental de fila sobre una matriz; por ejemplo, la operación $\mathcal{H}_{31(-2)}$ consistente en sumarle a la fila 3 de una matriz la fila 1 multiplicada por -2 .

Creemos y guardamos el siguiente *script* con el nombre de `msumf`:

```
% suma a la fila 3 de la matriz A, la fila 1
% multiplicada por -2
A(3,:)=A(3:)-2*A(1,:);
A
```

Vamos a probar este programa sobre la matriz

$$A = \begin{pmatrix} 1 & 0 & -1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 0 \\ 2 & 0 & -2 & 2 & 1 \end{pmatrix}$$

Para ello tecleamos desde la Command Window lo siguiente:

```
>> A=[1 0 -1 1 1 ; 1 1 1 2 0 ; 2 0 -2 2 1];
>> copia=A;
>> msumf
A =
     1     0     -1     1     1
     1     1     1     2     0
     0     0     0     0    -1
```

Vamos a analizar un poco lo que hace el programa `msumf`, y cómo lo hemos usado. La línea

```
A(3,:)=A(3:)-2*A(1,:);
```

que es la que realiza la operación, actúa de la siguiente forma: como siempre, el valor situado a la derecha del signo `=` se le asigna a la variable situada a la izquierda. `A(3,:)` es la tercera fila de la matriz `A` y `A(1,:)` la primera; por lo tanto `A(3:)-2*A(1,:)` es la nueva fila resultado de la operación, la que resulta de restarle a la tercera el doble de la primera. Estos nuevos valores pasan a ocupar la fila 3 de la matriz `A`, sustituyendo a los anteriores, ya que los almacenamos en `A(3,:)`. La línea de programa siguiente se limita a sacar en pantalla la nueva matriz `A`, ya con la operación incorporada.

Desde la Command Window, después de definir la matriz `A` y antes de ejecutar el programa, hemos creado una copia (llamada `copia`) de `A`

```
>> copia=A;
```

ya que, una vez ejecutado `msumf`, la variable `A` almacenará la matriz *transformada*, sobrescribiendo a la de partida, que se perdería si no la guardáramos en algún sitio. Si examináis el Workspace veréis que la variable que almacena la matriz inicial es ahora `copia`.

Por supuesto, tal como está el programa es poco útil; deberíamos poder adaptarlo para que realizara cualquier operación del tipo $\mathcal{H}_{ij(\lambda)}$, para filas i y j y números λ arbitrarios. Para ello basta modificarlo así:

```

% suma a la fila i de la matriz A, la fila j
% multiplicada por lambda
A(i,:)=A(i,:)+lambda*A(j,:);
A

```

y ahora, cada vez que lo queramos ejecutar, debemos indicar los valores de i , j y λ . Por ejemplo, en la matriz A (que ya ha sufrido la primera transformación), vamos a sumarle a la segunda fila la primera multiplicada por -1 .

```

>>i=2;j=1;lambda=-1;
>> msumf
A =
     1     0    -1     1     1
     0     1     2     1    -1
     0     0     0     0    -1

```

(Nota: Como vemos, se pueden introducir varios comandos en la misma línea de la Command Window o en una línea de programa, separados por puntos y comas si queremos que no salgan los resultados por pantalla, o por comas si queremos que salgan.) En el Workspace vemos aparecer las variables i , j , λ , además de A y copia .

Si convertimos este *script* en una *function* podremos controlar qué variables permanecen en el Workspace, y además no necesitaremos acordarnos cada vez que ejecutamos el programa de que la matriz que queremos transformar ha de llamarse A . Lo primero que tenemos que hacer es determinar cuáles son las variables de entrada (los datos sobre los que va a trabajar el programa) y cuáles las variables de salida (el resultado de ejecutar el programa). En este caso, las variables de entrada son claramente la matriz A a la que queremos aplicar la operación, y las filas i y j y el número λ que intervienen en la misma, y la variable de salida es la matriz transformada, que vamos a llamar de otra forma (B) para evitar confusiones.

La primera línea de una *function* tiene siempre la misma estructura, que tenemos que respetar: primero la palabra **function**, después un espacio en blanco, después las variables de salida, después un signo $=$, después el nombre del programa (que ha de ser necesariamente el mismo nombre con el que lo guardemos), y finalmente, entre paréntesis y separadas por comas, las variables de entrada. Vamos ya a editar nuestro archivo `msumf` y convertirlo en una *function*:

```

function B=msumf(A,i,j,lambda)
% suma a la fila i de la matriz A, la fila j
% multiplicada por lambda
B=A;
B(i,:)=B(i,:)+lambda*B(j,:);

```

En este caso realizamos la operación sobre una matriz B

```
B(i,:)=B(i,:)+lambda*B(j,:);
```

que previamente hemos inicializado como una copia de A,

```
B=A;
```

y que designamos como variable de salida, ya que al finalizar la ejecución del programa almacena la matriz transformada.

Si las variables de salida son más de una (es decir, si los resultados del programa salen almacenados en varias variables, cosa que sucede frecuentemente), han de ir entre corchetes y separados por comas.

Vamos a ejecutar este programa, pero antes, para empezar otra vez desde el principio, borraremos todas las variables del Workspace. Eso se hace con el comando

```
>> clear
```

La llamada a una *function* incluye necesariamente la asignación de valores a las variables de entrada. Si intentamos ejecutar `msumf` tecleando sin más el nombre del programa, como cuando era un *script*,

```
>> msumf
```

obtendremos como respuesta un mensaje de error, parecido al que recibimos al ejecutar el comando

```
>> cos
```

sin indicar de qué ángulo es el coseno que queremos calcular: en ambos casos hace falta indicar el o los argumentos. Al llamar una *function* hay que introducir los valores de las variables en el mismo orden en que aparecen en la primera línea del programa: en nuestro caso, primero la matriz que queremos modificar, después las dos filas que intervienen en la operación (en el orden adecuado), y después el número:

```
>>>> msumf([1 0 -1 1 1 ; 1 1 1 2 0 ; 2 0 -2 2 1],3,1,-2)
```

```
ans =
```

```
1     0    -1     1     1
1     1     1     2     0
0     0     0     0    -1
```

Si ahora miramos el Workspace veremos que la única variable que se ha creado es `ans`. Todas las variables que aparecen en una *function*, tanto las de entrada, como las de salida, como las que en su caso utilice internamente el programa, son variables *locales*, es decir, pertenecen al espacio de trabajo de éste, se borran al acabar la ejecución del mismo y por lo tanto no aparecerán en el Workspace.

También podríamos haber llamado a nuestra *function* asignándole de paso un nombre al resultado

```
>> matriz=msumf([1 0 -1 1 1 ; 1 1 1 2 0 ; 2 0 -2 2 1],3,1,-2)
```

Al entrar esta línea en la Command Window, MATLAB sigue los siguientes pasos: busca y localiza la función `msumf`; como la primera línea de ésta es `function B=msumf(A,i,j,lambda)`, introduce respectivamente

en las variables locales de entrada `A`, `i`, `j`, `lambda`

los valores $\begin{pmatrix} 1 & 0 & -1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 0 \\ 2 & 0 & -2 & 2 & 1 \end{pmatrix}$, 3, 1 y -2,

ejecuta el programa con esos datos, obteniendo $\begin{pmatrix} 1 & 0 & -1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix}$ como resultado de esa ejecución, guardado en la variable local de salida `B`; a continuación, asigna ese resultado a la nueva variable del Workspace `matriz` (o a `ans` si no hubiéramos especificado nosotros una), borra cualquier otro resto de la ejecución del programa y como la línea que hemos introducido para llamar al programa no acaba en `;` nos devuelve el resultado por pantalla

```
matriz =
     1     0    -1     1     1
     1     1     1     2     0
     0     0     0     0    -1
```

En vez de introducir los valores numéricos concretos de los argumentos al llamar a una *function*, podemos asignar todos o parte de ellos a través de variables del Workspace, por ejemplo

```
>> C=[1 -1; 3 -2 ; 4 6; 1 1];
>> resultado=msumf(C,2,1,-3)
resultado =
     1    -1
     0     1
     4     6
     1     1
```

Otro ejemplo:

```
>> i=[1 2 ; 3 4]; numero=-3;
>> j=msumf(i,2,1,numero)
j =
     1     2
     0    -2
```

Las variables globales i y j , que ahora preferimos utilizar para almacenar las matrices, no interfieren con las variables i y j de la *function*, ya que pertenecen a espacios de trabajo distintos. Ahora mismo, para nosotros, `msumf` es una “caja negra” que realiza determinado cálculo, sin importarnos cómo: sólo nos importa el resultado que obtendremos con determinados datos de entrada, exactamente igual que con la función `sin` o `cos`. Preferimos en general las *functions* a los *scripts* porque no queremos que el programa nos devuelva información que no nos interesa, ni tampoco preocuparnos porque dentro de las “tripas” de ese programa haya variables que puedan interferir con las que tengamos definidas en el momento de ejecutarlo. Las *functions* permiten programar en varios módulos o etapas, descomponiendo una tarea que puede ser muy complicada en diversos subprogramas que se escriben, corrigen o comprueban de una forma sencilla.

3.2. Ejercicios

1. Preparar una *function* `solucion.m` que resuelva el siguiente sistema de ecuaciones

$$\begin{cases} 5x + 2ry + rz = 2 \\ 3x + 6y + (2r - 1)z = 3 \\ 2x + (r - 1)y + 3rz = 5 \end{cases}$$

para un valor arbitrario del parámetro r . (La variable de entrada será el parámetro r ; la de salida, el vector solución del sistema. Recordar que `A\b` proporciona la solución del sistema de ecuaciones con matriz de coeficientes A y vector de términos independientes b .)

2. Preparar tres *functions* que efectúen cada una de las tres operaciones elementales de fila sobre una matriz dada. Las *functions* tendrán los siguientes encabezamientos (la segunda de ellas ya la tenemos):

```
function B=mprodf(A,i,lambda)
% multiplica la fila i de la matriz A por lambda
```

```
function B=msumf(A,i,j,lambda)
% suma a la fila i de la matriz A, la fila j
% multiplicada por lambda
```

```
function B=minterf(A,i,j)
% intercambia las filas i y j de la matriz A
```

Teclear `format rat` para obtener los resultados numéricos que siguen en forma de fracciones.

Utilizando las *functions* recién programadas, calcular la forma escalonada por filas y la forma escalonada reducida por filas de la matriz

$$A = \begin{pmatrix} 1 & 1 & -1 & 1 & -2 & -1 \\ 4 & -5 & 7 & -2 & -4 & -6 \\ 2 & 5 & -8 & 4 & -3 & 1 \\ 3 & -3 & 4 & -1 & -3 & -4 \end{pmatrix}$$

Comprobar que el segundo resultado es el mismo que el obtenido ejecutando el comando

```
>> rref(A)
```

3. Teclear `format long` para obtener los resultados numéricos que siguen en el formato de muchas cifras decimales.

Se considera la función $f(x) = xe^x - 1$. Preparar una *function*

```
function y=valores(a,b)
```

que calcule los valores de f en once puntos equiespaciados entre a y b (incluidos estos dos); dicho de otra forma, que evalúe f sobre los puntos que marcan la división de $[a, b]$ en diez subintervalos iguales (estos puntos se obtienen mediante `linspace(a,b,11)`). La salida de la *function* será una matriz 2×11 , llamada y , cuya primera fila almacenará los once puntos de la partición del intervalo, y la segunda los once valores correspondientes de la función.

Utilizar sucesivamente la función `valores` para aproximar hasta la quinta cifra decimal el único cero de la función f en el intervalo $[0, 1]$. (Al ejecutar `valores` sobre el intervalo $[0, 1]$ observamos que la función cambia de signo en el subintervalo $[0'5, 0'6]$, que por lo tanto contendrá la raíz. Aplicamos de nuevo `valores` sobre este subintervalo, para obtener la segunda cifra decimal, y así sucesivamente.)

Capítulo 4

Cuarta sesión

Vamos a aprender a trabajar con algunos bucles (*loops*), básicos en programación. Se utilizan cuando queremos repetir un proceso un determinado número de veces.

4.1. Bucles *for... end*

Empecemos con un ejemplo sencillo. Supongamos que queremos imprimir en la Command Window las potencias quintas de los primeros 10 números naturales. Una forma de hacer esto es crear un *script* con las líneas

```
k=1:10;  
k.^5
```

(Por supuesto también se pueden ejecutar sucesivamente estos comandos desde la Command Window.) Guardamos el *script* como *potencias.m*. Al ejecutarlo obtenemos

```
>> potencias  
ans =  
Columns 1 through 5  
      1      32      243      1024      3125  
Columns 6 through 10  
    7776    16807    32768    59049    100000
```

Como era de esperar la variable *k* aparece en el Workspace (ya que hemos programado un *script* y no una *function*). Esta variable y *ans* son vectores de 10 componentes: el primero contiene los números del 1 al 10 y el segundo, las potencias quintas de estos números.

Hay otra forma de hacer lo mismo: mediante un bucle *for*. Tecleamos *clear* para limpiar el Workspace y modificamos así el programa *potencias*:

```
for k=1:10
    k^5
end
```

Se trata de pedirle a MATLAB que ejecute el comando o comandos situados desde la línea `for...` hasta la línea `end` tantas veces como indique el contador situado en la línea `for...`: en este caso, desde que `k` es igual a 1 hasta que es igual a 10. La respuesta que obtenemos al ejecutar esta nueva versión de potencias es

```
>> potencias
ans =
     1
ans =
    32
ans =
   243
ans =
  1024
ans =
  3125
ans =
  7776
ans =
 16807
ans =
 32768
ans =
 59049
ans =
100000
```

Lo que hemos hecho es ejecutar sucesivamente el comando `k^5` desde que `k` es 1 hasta que `k` es 10, pasando por todos los valores intermedios. El bucle empieza con `k` igual a 1. Nos encontramos con la línea `k^5` que nos pide evaluar esa expresión para el valor actual de `k`, que es 1, imprimir el resultado en pantalla (ya que la línea no acaba con punto y coma) y guardar el resultado en la variable `ans` (ya que no indicamos otra variable en la que guardarlo). Después viene la línea `end` que nos dice que el paso `k=1` está terminado; entramos de nuevo en el bucle con `k=2` y hacemos la misma operación; de los valores anteriores de `k` y `ans` no queda ni rastro... y así sucesivamente hasta alcanzar el valor `k=10`.

Ahora en el Workspace sólo aparecen las variables `k` y `ans`, pero no son vectoriales sino escalares: guardan los últimos valores de `k` y de `ans`, correspondientes a la ejecución `k=10` del bucle `for`.

Se puede utilizar un paso distinto de 1 para el bucle `for`. Por ejemplo, modificando potencias así

```
for k=1:2:10
    k^5
end
```

aparecen las potencias quintas de los números del 1 al 10 pero con un salto de 2, es decir, 1, 3, 5, 7, 9.

Vamos a ver un ejemplo un poco más elaborado. Supongamos que queremos calcular la suma de los cubos de los 100 primeros números naturales,

$$1^3 + 2^3 + 3^3 + 4^3 + \dots + 100^3$$

Podemos hacerlo con un *script* como el que sigue:

```
k=1:100;
s=k.^3;
sum(s)
```

Este programa genera dos variables vectoriales: una, `k`, con los números naturales del 1 al 100 y otra, `s`, con los valores de la sucesión k^3 en cada uno de esos números. (Comprobadlo ejecutando el programa.) Si lo único que nos interesa es el valor final de la suma, no tiene mucho sentido generar esas variables. De hecho este tipo de sumas de términos consecutivos de una sucesión se suelen calcular haciendo uso de un bucle `for`. Modificamos el *script* así:

```
suma=0;
for k=1:100;
    suma=suma+k^3;
end
suma
```

Analicemos lo que hace este programa, antes de ejecutarlo. Al empezar la ejecución inicializamos `suma` a cero. Después entramos en el bucle por primera vez, con el valor de `k=1`; la línea `suma=suma+k^3`; tiene el siguiente efecto: asigna el valor $\text{suma} + k^3 = 0 + 1^3 = 1^3$ a la variable `suma` (el primer término de la suma que queremos calcular). Con ese valor de `suma=1^3` se ejecuta el segundo paso del bucle, correspondiente a `k=2`. La línea `suma=suma+k^3`; en este caso se ejecuta así: se asigna el valor $\text{suma} + k^3 = 1^3 + 2^3 = 9$ a la variable `suma` (que ahora almacena la suma de los dos primeros términos). Con ese valor de `suma=1^3 + 2^3` se entra en el bucle por tercera vez (`k=3`) ... como vemos, `suma` almacena en cada paso las sumas parciales de la expresión de partida, hasta llegar al paso `k=100`, en el que guardará la suma de los cien términos. Al acabar el bucle la línea `suma` hace que salga el resultado en pantalla.

Como en el caso anterior, ahora las variables `k` y `s` ya no almacenan un vector sino un único valor numérico, distinto en cada ejecución del bucle, y por eso podemos prescindir de las operaciones componente a componente en el programa (es decir, de anteponer un punto a los operadores de potenciación o cociente).

Los bucles se pueden *anidar*, es decir, meter unos dentro de otros. Por ejemplo, la ejecución de este *script*

```
for i=1:3
    disp('Hola')
    for j=1:2
        disp('Adios')
    end
end
```

produce la siguiente estupidez:

```
Hola
Adios
Adios
Hola
Adios
Adios
Hola
Adios
Adios
```

Dentro de cada una de las tres ejecuciones del bucle en `i` se realizan dos ejecuciones del bucle en `j`. El comando `disp` (de *display*) se utiliza para mostrar valores de variables o cadenas de caracteres en pantalla.

4.2. Bucles *if... end* y *while... end*

Ambos tipos de bucle son de ejecución condicional, es decir, los comandos que engloban se ejecutan sólo si se verifica determinada condición. En el caso de los bucles `if`, los comandos se ejecutarán, si la condición se cumple, una sola vez. Por ejemplo, ejecutad este *script*

```
i=input('Escribe un numero ')
if i>10
    disp('Es mayor que 10')
end
```

(fijaos de paso en el uso del comando `input` para introducir datos durante la ejecución). El comando `disp('Es mayor que 10')` se ejecutará sólo si se cumple la condición `i>10`.

Es posible introducir varias condiciones dentro del bucle `if`, por ejemplo

```

i=input('Escribe un numero ')
if i>10
    disp('Es mayor que 10')
elseif i==10
    disp('Es igual a 10')
else
    disp('Es menor que 10')
end
end
end

```

`elseif` significa “si en cambio se cumple que...”, mientras que `else`, que aparece (si es el caso) al final de la lista de condiciones, significa “en cualquier otro caso...” Notar que el signo igual de la línea 4 representa una identidad, no (como los que nos hemos encontrado hasta ahora) una asignación. En este caso se utiliza el doble signo igual.

En el caso de los bucles `while...end`, los comandos del bucle se ejecutarán un número indefinido de veces, hasta que la condición deje de cumplirse. Por ejemplo, vamos a calcular mediante un *script* el menor número natural cuyo factorial es mayor o igual que 10^5 .

```

k=1;
while factorial(k)<10000
    k=k+1;
end
k

```

En la primera línea inicializamos `k` a 1, para ir probando con todos los factoriales a partir de $1!$. Ahora nos encontramos el bucle `while`, en el que estamos condenados a entrar hasta que la condición de entrada (`factorial(k)<10000`) deje de cumplirse. En este primer momento `k` es 1 y por lo tanto la condición se cumple (el factorial de 1 es menor que 10^5); por lo tanto se ejecuta la línea de dentro del bucle, que añade 1 al contador. Luego la vez siguiente que se comprueba si se cumple o no la condición de entrada, `k` ya vale 2; como todavía $2! < 10^5$, seguimos entrando y por lo tanto añadiendo una unidad a `k`, y así sucesivamente hasta que `k` haya crecido lo suficiente como para superar 10^5 , momento en el que el bucle deja de ejecutarse y `k` se queda con ese primer valor que no cumple la condición.

Al ejecutar este *script* obtenemos la respuesta

```

k =
    8

```

Podemos comprobar que el programa ha funcionado, es decir, que 8 es el primer número cuyo factorial supera 10^4

```

>> factorial(7)
ans =
    5040
>> factorial(8)
ans =
    40320

```

4.3. Ejercicios

1. La *sucesión de Fibonacci* se define por recurrencia de la siguiente forma: los primeros dos términos son iguales a 1, y a partir del tercero, cada término es la suma de los dos anteriores.
 - (a) Preparar un programa que calcule y almacene en una variable los 50 primeros términos de la sucesión. (Empezar creando una matriz fila de 50 ceros, que se irá rellenando con los sucesivos valores de la sucesión, mediante un bucle `for` adecuado.)
 - (b) Si dividimos cada término de la sucesión por el anterior, obtenemos otra sucesión que resulta ser convergente. Modificar el programa para ir calculando y almacenando estos cocientes a medida que se calculan los términos de la sucesión de partida. Aproximar el valor del límite. (El límite de estos cocientes es la razón áurea, $\Phi = (1 + \sqrt{5})/2$.)
2. Crear una *function* que, introducida por el usuario una matriz arbitraria, devuelva una matriz del mismo tamaño en la que se ha sumado 1 a los elementos de la primera fila de la matriz original, 2 a los elementos de la segunda, 3 a los de la tercera, y así sucesivamente. La *function* tendrá un único argumento de entrada (la matriz inicial) y un único argumento de salida (la matriz resultado). `size(A,1)` da el número de filas, y `size(A,2)` el de columnas, de la matriz `A`.
3. Crear un *script* en el que, mediante el uso de bucles y de condicionales, se genere una matriz 5×8 con los siguientes elementos:
 - si el elemento está en una columna par o bien en una fila par, la raíz cuadrada de la suma de los dos índices (de fila y de columna).
 - en otro caso, la suma de los dos índices elevados al cuadrado.

Nota: El resto de la división de `x` entre `y` se puede calcular en MATLAB mediante `rem(x,y)`. El “o” lógico se escribe con una barra vertical, `|`. De esta forma, la condición “`i` es par o `j` es par” se podría escribir así: `(rem(i,2)==0) | (rem(j,2)==0)`

Apéndice A

Soluciones a los ejercicios

A.1. Primera sesión

- ```
>> w=(3i-1)^5/(5+i)
w =
-61.2308 + 9.8462i
>>abs(w)
ans =
62.0174
>> angle(w)
ans =
2.9822
```
- ```
>> n=[1 10 100 500 1000 2000 4000 8000];
>> y=(1+1./n).^n
y =
Columns 1 through 6
2.0000    2.5937    2.7048    2.7156    2.7169    2.7176
Columns 7 through 8
2.7179    2.7181
>> exp(1)
ans =
2.7183
```
- ```
>> A=[2 6; 3 9]; B=[1 2; 3 4]; C=[-5 5; 5 3];
```

Pueden ir varios comandos en la misma línea, separados por `,` o bien por `;`. Si utilizamos comas MATLAB nos devuelve el resultado en pantalla.

Primero inicializo la matriz a ceros

```
>> G=zeros(6,6);
```

después meto las tres matrices como submatrices de **G**

```
>> G(1:2,1:2)=A; G(3:4,3:4)=B; G(5:6,5:6)=C
```

```
G =
```

```
 2 6 0 0 0 0
 3 9 0 0 0 0
 0 0 1 2 0 0
 0 0 3 4 0 0
 0 0 0 0 -5 5
 0 0 0 0 5 3
```

Eliminar la última fila y la última columna: Como quiero conservar la matriz **G**, primero le asigno el mismo valor a una nueva variable **F** sobre la que haré los cambios:

```
>> F=G;
```

y ahora hago la eliminación sobre **F**

```
>> F(6,:)=[]
```

```
F =
```

```
 2 6 0 0 0 0
 3 9 0 0 0 0
 0 0 1 2 0 0
 0 0 3 4 0 0
 0 0 0 0 -5 5
```

```
>> F(:,6)=[]
```

```
F =
```

```
 2 6 0 0 0
 3 9 0 0 0
 0 0 1 2 0
 0 0 3 4 0
 0 0 0 0 -5
```

Extraer la submatriz  $4 \times 4$  de la esquina superior izquierda de **G**:

```
>> H=G(1:4,1:4)
```

```
H =
```

```
 2 6 0 0
 3 9 0 0
 0 0 1 2
 0 0 3 4
```

Extraer la submatriz  $\{1, 3, 6\} \times \{2, 5\}$  de **G**:

```
>> K=G([1 3 6],[2 5])
K =
 6 0
 0 0
 0 5
```

Para cambiar el valor de un elemento basta con asignarle el nuevo:  
Como quiero conservar la matriz  $G$ , los cambios los haré sobre  $J$

```
>> J=G;
>> J(5,5)=4
J =
 2 6 0 0 0 0
 3 9 0 0 0 0
 0 0 1 2 0 0
 0 0 3 4 0 0
 0 0 0 0 4 5
 0 0 0 0 5 3
```

Nota: La mayor parte de estos manejos (eliminación de filas, cambio de valor de elementos, etc.) se pueden hacer desde la ventana del Workspace, editando la variable. Pero necesitamos saber hacerlo también con comandos.

```
4. >> A=[2 -1 3; 1 4 1; 6 10 3]; b=[4;2;0];
>> inv(A)*b
ans =
 -1.8049
 0.2927
 2.6341
>> A\b
ans =
 -1.8049
 0.2927
 2.6341
```

La solución es  $x = -1'8049$ ,  $y = 0'2927$ ,  $z = 2'6341$ .

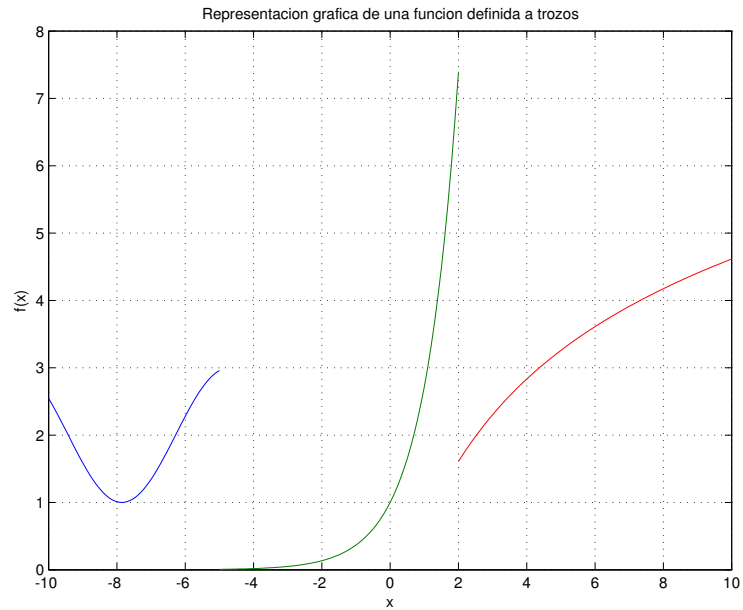
## A.2. Segunda sesión

```
2. >> x=-10:.1:-5;
>> y=2+sin(x);
>> z=-5:.1:2;
>> t=exp(z);
```

```

>> u=2:.1:10;
>> v=log(u.^2+1);
>> plot(x,y,z,t,u,v)
>> grid on
>> xlabel('x'), ylabel('f(x)')
>> title('Representacion grafica de una funcion definida a trozos')

```



Notar que hemos generado tres tablas de valores:  $(x,y)$ ,  $(z,t)$ ,  $(u,v)$ .

3. El programa podría ser

```
% Resuelve un sistema de ecuaciones en funcion de un parametro
```

```

A=[5, 2, r; 3, 6, 2*r-1; 2, r-1, 3*r];
b=[2; 3 ; 5];
s=A\b

```

Lo guardamos como solucion y probamos si funciona

```

>> r=10;
>> solucion
s=
-0.0220
-0.4286
0.2967
>> r=5;

```

```
>> solucion
s=
 0.0833
 -0.0417
 0.3333
```

4. `suma` es la variable en la que se van almacenando las sumas parciales. El programa inicializa su valor a cero; después entra en el bucle. En la primera ejecución del bucle `k` vale 1. La línea

```
suma=suma+(-1)^(k-1)/k;
```

como cualquier igualdad dentro de un programa o una secuencia de comandos, es en realidad una asignación: se asigna el valor a la derecha del signo `=` a la variable indicada a la izquierda. En este caso, el valor  $\text{suma}+(-1)^{(k-1)}/k = 0 + (-1)^{1-1}/1$  se le asigna a la variable `suma`, sustituyendo el valor anterior, que era 0. `suma` pasa a almacenar, por lo tanto, el primer sumando. Acaba el bucle en `end` y vuelve a ejecutarse para `k` igual a 2 y ese nuevo valor de `suma`. Luego en esta ejecución la línea

```
suma=suma+(-1)^(k-1)/k;
```

asigna el valor  $\text{suma}+(-1)^{(k-1)}/k = (-1)^{1-1}/1 + (-1)^{2-1}/2$  a la variable `suma`, sustituyendo el valor anterior. `suma` pasa a almacenar la suma de los dos primeros términos de la sucesión. El bucle se ejecuta de nuevo para `k=3`, y al acabar esa ejecución `suma` almacenará la suma de los tres primeros términos, y así sucesivamente hasta `k=10000`.

Al terminar las 10000 ejecuciones del bucle el programa sale del mismo y ejecuta las líneas de comando

```
suma
vreal=log(2)
difa=abs(suma-vreal)
```

Ninguna de las tres líneas acaba en `;` así que las tres producirán una salida por pantalla, la de cada una de las tres variables `suma` (que a estas alturas almacena la suma de los 10000 sumandos), `vreal` que es el valor auténtico de  $\ln 2$  y `difa` que es el error cometido en la aproximación.

### A.3. Tercera sesión

1. El programa podría ser

```
function s=solucion(r)
% Resuelve un sistema de ecuaciones en funcion de un parametro
A=[5, 2, r; 3, 6, 2*r-1; 2, r-1, 3*r];
b=[2; 3 ; 5];
s=A\b;
```

La llamada al programa incluirá la asignación de un valor al parámetro

```
>> solucion(5)
ans =
 0.0833
 -0.0417
 0.3333
>> solucion(100)
ans =
 0.1437
 0.0093
 0.0126
```

2. Las *functions* pedidas podrían ser

```
function B=mprodf(A,i,lambda)
% multiplica la fila i de la matriz A por lambda
B=A;
B(i,:)=lambda*B(i,:);
```

.....

```
function B=msumf(A,i,j,lambda)
% suma a la fila i de la matriz A,
% la fila j multiplicada por lambda
B=A;
B(i,:)=B(i,:)+lambda*B(j,:);
```

.....

```
function B=minterf(A,i,j)
% intercambia las filas i y j de la matriz A
B=A;
B([j i],:)=B([i j],:);
```

Vamos a explicar un poco más la línea  $B([i \ j], :)=B([j \ i], :)$ . Lee-  
mos de derecha a izquierda: asignarle el valor  $B([j \ i], :)$  a  $B([i \ j], :)$ .  
Es decir: la submatriz de B formada por las filas j e i (en ese orden)  
la metemos en B como submatriz  $B([i \ j], :)$ , sustituyendo el antiguo  
valor de esa submatriz.

Una vez guardadas las *functions*, ya las podemos usar como comandos.  
Partimos de

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

Vamos a llamarle p. ej. X a la matriz que almacenará todos los resul-  
tados parciales, hasta la forma reducida final.

```
>> X=msumf(A,2,1,-4)
X =
 1 1 -1 1 -2 -1
 0 -9 11 -6 4 -2
 2 5 -8 4 -3 1
 3 -3 4 -1 -3 -4
```

La siguiente operación elemental la haré sobre el resultado X de haber  
aplicado la primera. La matriz resultante la vuelvo a almacenar en X  
porque no me interesa guardar estos resultados intermedios.

```
>> X=msumf(X,3,1,-2); X=msumf(X,4,1,-3)
X =
 1 1 -1 1 -2 -1
 0 -9 11 -6 4 -2
 0 3 -6 2 1 3
 0 -6 7 -4 3 -1
>> X=minterf(X,2,3)
X =
 1 1 -1 1 -2 -1
 0 3 -6 2 1 3
 0 -9 11 -6 4 -2
 0 -6 7 -4 3 -1
```

```
>> X=msumf(X,3,2,3); X=msumf(X,4,2,2)
```

```
X =
 1 1 -1 1 -2 -1
 0 3 -6 2 1 3
 0 0 -7 0 7 7
 0 0 -5 0 5 5
```

```
>> X=msumf(X,4,3,-5/7)
```

```
X =
 1 1 -1 1 -2 -1
 0 3 -6 2 1 3
 0 0 -7 0 7 7
 0 0 0 0 0 0
```

Ya tengo una forma escalonada por filas. Guardo este resultado en una nueva variable F1

```
>> F1=X
```

```
F1 =
 1 1 -1 1 -2 -1
 0 3 -6 2 1 3
 0 0 -7 0 7 7
 0 0 0 0 0 0
```

y sigo haciendo operaciones elementales hasta llegar a la reducida

```
>> X=mprodf(X,2,1/3); X=mprodf(X,3,-1/7)
```

```
X =
Columns 1 through 5
 1.0000 1.0000 -1.0000 1.0000 -2.0000
 0 1.0000 -2.0000 0.6667 0.3333
 0 0 1.0000 0 -1.0000
 0 0 0 0 0
Column 6
 -1.0000
 1.0000
 -1.0000
 0
```

Si quiero puedo hacer que los resultados salgan en forma fraccionaria, tecleando

```
>> format rat
```

Sigo con las operaciones elementales de fila

```
>> X=msumf(X,1,2,-1)
X =
Columns 1 through 4
 1 0 1 1/3
 0 1 -2 2/3
 0 0 1 0
 0 0 0 0
Columns 5 through 6
 -7/3 -2
 1/3 1
 -1 -1
 0 0
```

```
>> X=msumf(X,1,3,-1); X=msumf(X,2,3,2)
X =
Columns 1 through 4
 1 0 0 1/3
 0 1 0 2/3
 0 0 1 0
 0 0 0 0
Columns 5 through 6
 -4/3 -1
 -5/3 -1
 -1 -1
 0 0
```

Esta última ya es la forma escalonada reducida por filas. Tecleando `rref(A)` compruebo que da el mismo resultado.

Salgo del formato racional p. ej. al formato con muchos decimales, para hacer el siguiente ejercicio:

```
>> format long
```

```
3. function y=valores(a,b)
% ejercicio 3, tercera sesion de
% MATLAB curso 2006/07
y=zeros(2,11);
y(1,:)=linspace(a,b,11);
y(2,:)=y(1,:).*exp(y(1,:))-1;
```

La línea `y=zeros(2,11)` inicializa la matriz y a ceros, reservando el espacio necesario en memoria. La línea `y(1,:)=linspace(a,b,11);`

coloca en la primera fila  $y(1,:)$  de la matriz  $y$  once valores entre  $a$  y  $b$  a distancias iguales. La línea  $y(2,:)=y(1,:).*\exp(y(1,:))-1$ ; evalúa la función  $f(x) = xe^x - 1$  en cada uno de esos once valores y coloca los once resultados en la segunda fila  $y(2,:)$  de la matriz  $y$ .

Ejecutamos el programa en el intervalo  $[0, 1]$

```
>> valores(0,1)
ans =
 Columns 1 through 3
 0 0.100000000000000 0.200000000000000
-1.000000000000000 -0.88948290819244 -0.75571944836797
 Columns 4 through 6
 0.300000000000000 0.400000000000000 0.500000000000000
-0.59504235772720 -0.40327012094349 -0.17563936464994

 Columns 7 through 9
 0.600000000000000 0.700000000000000 0.800000000000000
 0.09327128023431 0.40962689522933 0.78043274279397
 Columns 10 through 11
 0.900000000000000 1.000000000000000
 1.21364280004126 1.71828182845905
```

Debajo de cada valor de la  $x$  encontramos la evaluación de la función en ese punto. Vemos que el signo de la función cambia entre  $0'5$  y  $0'6$ , luego la raíz está en el intervalo  $[0'5, 0'6]$ .

```
>> valores(0.5,0.6)
ans =
 Columns 1 through 3
 0.500000000000000 0.510000000000000 0.520000000000000
-0.17563936464994 -0.15070149057760 -0.12534562215658
 Columns 4 through 6
 0.530000000000000 0.540000000000000 0.550000000000000
-0.09956587643217 -0.07335629442018 -0.04671084017293
 Columns 7 through 9
 0.560000000000000 0.570000000000000 0.580000000000000
-0.01962339983418 0.00791221931723 0.03590228983504
 Columns 10 through 11
 0.590000000000000 0.600000000000000
 0.06435316508474 0.09327128023431
```

La raíz está en el intervalo  $[0'56, 0'57]$  así que hacemos

```

>> valores(0.56,0.57)
ans =
Columns 1 through 3
 0.560000000000000 0.561000000000000 0.562000000000000
 -0.01962339983418 -0.01689010883386 -0.01415232987487
Columns 4 through 6
 0.563000000000000 0.564000000000000 0.565000000000000
 -0.01141005671286 -0.00866328309542 -0.00591200276217
Columns 7 through 9
 0.566000000000000 0.567000000000000 0.568000000000000
 -0.00315620944469 -0.00039589686653 0.00236894125680
Columns 10 through 11
 0.569000000000000 0.570000000000000
 0.00513831121786 0.00791221931723

```

Así seguiríamos hasta obtener la precisión pedida.

#### A.4. Cuarta sesión

1. (a) `% sucesion de fibonacci`  
`f=zeros(1,50);`  
`f(1)=1;f(2)=1;`  
`for k=3:50`  
`f(k)=f(k-2)+f(k-1);`  
`end`
- (b) `% sucesion de fibonacci con calculo de cocientes`  
`f=zeros(1,50);q=zeros(1,50);`  
`f(1)=1;f(2)=1;q(1)=1;q(2)=1;`  
`for k=3:50`  
`f(k)=f(k-2)+f(k-1);`  
`q(k)=f(k)/f(k-1);`  
`end`

Las variables `f` y `q` se pueden recuperar desde el Workspace.

Inicializamos las variables con `zeros` simplemente para reservar espacio en memoria; cuando una matriz se va “rellenando” mediante la ejecución de un bucle es más eficiente inicializarla previamente como una matriz de ceros del mismo tamaño.

```

2. function B=transformada(A)
% ejercicio 2, cuarta sesión de MATLAB
m=size(A,1);n=size(A,2);
B=zeros(m,n);
for i=1:m
 B(i,:)=A(i,)+i;
end

3. A=zeros(5,8);
for i=1:5
 for j=1:8
 if (rem(i,2)==0)|(rem(j,2)==0)
 A(i,j)=sqrt(i+j);
 else
 A(i,j)=i^2+j^2;
 end
 end
end
A

```