Práctica I: Programación en Matlab/Octave.

Para la realización de esta práctica, el alumno leerá detenidamente las secciones 1 a 4, ejecutando en su ordenador todos los ejemplos incluidos en dichas secciones y comprobando que dichos ejemplos se ejecutan correctamente. A continuación deberá realizar los ejercicios indicados en la sección 5, consistentes en la elaboración de programas utilizando todo lo aprendido en las cuatro primeras secciones. Cada ejercicio tendrá como resultado un archivo .m. Todos los archivos serán comprimidos en un archivo zip que deberá ser subido al campus virtual en el enlace dispuesto para ello.

1. Programación en Matlab/Octave.

Un *programa* (*script*) es simplemente una secuencia ordenada de comandos que lleva a cabo alguna acción. En Matlab / Octave, un programa habitualmente ocupa un archivo de texto. A continuación desarrollaremos nuestro primer programa en Matlab.

Importante: Podemos guardar nuestro archivo con el programa en cualquier directorio del disco duro del PC, o en un pendrive. Si ejecutamos el comando:

```
>>> pwd
```

matlab nos muestra el directorio de búsqueda actual; si vamos a guardar nuestro programa en un directorio distinto, debemos ejecutar:

```
>>> cd ruta_del_directorio_donde_vamos_a_trabajar
```

ya que de no hacerlo así, Matlab será incapaz de encontrar y ejecutar nuestro programa. *Es recomendable crear un directorio específico para guardar los programas matlab*. Asimismo, si introducimos el comando:

```
>>> dir
```

(o, de modo equivalente, el comando ls), matlab nos muestra todos los archivos contenidos en ese directorio.

1.1. Nuestro primer programa.

Vamos a confeccionar un programa que resuelva la ecuación de segundo grado $ax^2 + bx + c = 0$. Para ello abrimos un archivo nuevo y tecleamos las siguientes lineas: (Pinchar aquí para descargar el archivo .m)

```
disp('Resolución de una ecuación de segundo grado')
disp('===========================================')
disp(' ')
disp('Introducir el valor de los parámetros ax2+bx+c=0')
a=input('a => ');
b=input('b => ');
c=input('c => ');
disp(' ')
disp('Soluciones:')
disp('===========')
discri=sqrt(b^2-4*a*c);
x1=(-b+discri)/(2*a)
x2=(-b-discri)/(2*a)
```

En este fragmento de código hemos utilizado dos funciones propias de Matlab/Octave:

- x=input('mensaje'): nos permite solicitar un valor de entrada, mostrando un mensaje y asignando el resultado a la variable x.
- disp('mensaje de salida'): muestra un mensaje de salida.

Asimismo, dado que las lineas en que se calculan x1 y x2 no terminan en ';', sus valores se muestran directamente por pantalla.

Al terminar guardamos el archivo con el nombrte EcuacionSegundoGrado.m y en la terminal de matlab introducimos el nombre de este archivo (sin la extensión .m):

```
>>> EcuacionSegundoGrado
```

Podemos comprobar como matlab nos pide los valores de a, b y c, y nos devuelve las soluciones de la ecuación.

2. Funciones en matlab

Una función en Matlab/Octave es conceptualmente muy similar a un programa. De hecho, todos los programas pueden entenderse como funciones. Concretamente, una función es un *script* (secuencia de

comandos) particular que se caracteriza porque acepta argumentos. La estructura general de una función es la siguiente:

```
function [resultado] = nombreFun(argumentos)
% Texto de ayuda 1;
% Texto de ayuda 2;
    Cuerpo de la función;
end %function
```

siendo:

- resultado: el nombre o nombres de las variables que se van a producir como resultado de la función.
- nombreFun: nombre con el que se va a llamar la función.
- argumentos: valor o valores de entrada que necesita la función para realizar los cálculos para los que está diseñada.
- Texto de ayuda: Texto de ayuda que se mostrará al ejecutar help nombreFuncion en la línea de comandos de Matlab.
- Cuerpo: líneas de código en las que se implementam los cálculos. Habitualmente deben terminar en ';' para que no se muestren los resultados intermedios.
- end: indica el final de la función. En Octave puede utilizarse también endfunction. En Matlab resulta recomendable escribir end %function; aquí el comentario %function permite identificar el end como el final de la función (especialmente útil si el código incluye otros comandos end, evitando posibles confusiones)

Las funciones se guardan también en archivos cuyo nombre **debe coincidir** con el nombre de la función y deben tener la extensión .m.

2.1. Nuestra primera función.

Vamos a construir una función que, dado un valor r, calcula la longitud de la circunferencia de radio r y el área del círculo correspondiente; en este caso, la función recibe un único argumento, r, y devuelve como resultado un vector [long,area] que contiene los valores calculados. Para ello abrimos un archivo de texto y tecleamos el siguiente código: (Pinchar aquí para descargar el archivo .m)

```
function [long,area]=circ(r)
% Cálculo de la longitud y area de una circunferencia
```

```
    diametro=2*r
    long=pi*diametro;
    area=pi*r^2;
end % function
```

Guardamos el archivo con el nombre circ.m. Si ahora, desde la línea de comandos de matlab ejecutamos:

```
>>> [a,b]=circ(3)
```

veremos que matlab nos devuelve un vector con la longitud de la circunferencia (que se almacena en a), y el área del círculo (que se almacena en b). Por cierto, si ejecutamos simplemente:

```
>>> [a,b]=circ(3)
```

matlab muestra sólo el último de los valores que calcula la función, en este caso, el área del círculo.

Por último, no debe olvidarse que los argumentos de una función van siempre entre paréntesis y los resultados de salida entre corchetes. Si el resultado es un único valor, no son necesarios los corchetes (si bien no está de más ponerlos siempre, por claridad). A veces hemos de programar funciones que no reciben argumentos, en cuyo caso bastará con dejar un espacio vacío entre los paréntesis; asimismo, si no se producen resultados dejaremos un espacio en blanco entre los corchetes. Obsérvese como de esta forma es sencillo convertir un *script* en una función; por ejemplo nuestro script anterior para el cálculo de las soluciones de la ecuación de segundo grado podría convertirse en función del siguiente modo: (Pinchar aquí para descargar el archivo .m)

```
function []=resuelve2G()
    disp('Resolución de una ecuación de segundo grado')
    disp('===========================================')
    disp(' ')
    disp('Introducir el valor de los parámetros ax2+bx+c=0')
    a=input('a => ');
    b=input('b => ');
    c=input('c => ');
    disp(' ')
    disp('Soluciones:')
    disp('===========')
    discri=sqrt(b^2-4*a*c);
    x1=(-b+discri)/(2*a)
    x2=(-b-discri)/(2*a)
end % function
```

aunque, desde luego, para ser realmente una función que acepte argumentos y devuelva resultados, su código debería programarse más bien del siguiente modo: (Pinchar aquí para descargar el archivo .m)

```
function [x1,x2]=resolverE2G(a,b,c)
% Función que resuelve la ecuación de segundo grado
% a*x^2+b*x+c=0
% dados los valores de a, b y c.
    discri=sqrt(b^2-4*a*c);
    x1=(-b+discri)/(2*a);
    x2=(-b-discri)/(2*a);
end % function
```

y podríamos llamarla desde la consola de matlab mediante:

```
>>> [soluc1,soluc2]=resolverE2G(1,2,1)
```

Señalemos, por último, que una función puede recibir como argumentos tanto vectores como matrices, y también devolver como resultado vectores y/o matrices. Así, por ejemplo, podemos construir una función que multiplique una matriz A por un vector (columna) b, y devuelva el vector resultante: (Pinchar aquí para descargar el archivo .m)

```
function [M]=producto(A,b)
% Multiplica    una matriz A por un vector b
    M=A*b;
end %function
```

Podemos ver el efecto de esta función ejecutando:

```
>>> A=[1 2; 3 4]
>>> b=[1; 1]
>>> producto(A,b)
```

2.2. Scripts vs. funciones.

Como hemos visto en los ejemplos anteriores, scripts y funciones son conceptos muy parecidos, pero con diferencias importantes:

- Al llamar a un *script* por su nombre, no es posible pasarle argumentos; así el *script* que hemos desarrollado para resolver una ecuación de segundo grado no puede ser llamado desde un programa más complejo que le pase los valores de a, b y c.

- Las variables generadas en un *script* son variables *globales:* una vez creadas, permanecen en el *espacio de trabajo* del usuario; por contra, las variables generadas en el cuerpo de una función son *locales*, solo "existen" dentro de la función, y por tanto puede haber otras variables con el mismo nombre en el *workspace* global. Puede comprobarse ejecutando el script resuelveE2G; si a continuación escribimos discri en la linea de comandos de matlab, veremos que nos muestra el valor que ha resultado del cálculo de dicha variable dentro del *script.* Ahora bien si ejecutamos, por ejemplo, circ(7) y a continuación escribimos diametro en la linea de comandos, matlab nos informa de que dicha variable es desconocida.

- Un *script* en definitiva, resulta útil para automatizar una serie de comandos que deben repetirse de la misma forma muchas veces; una función, sin embargo, nos permite extender el lenguaje matlab con nuevas capacidades.

Tanto *scripts* como funciones se conocen genéricamente como m-files (ficheros m) ya que tanto unos como otras se archivan siempre en archivos con esta extensión. Si queremos conocer los m-files de nuestro directorio de trabajo basta con ejecutar el comando:

```
>>> what
```

El comando

```
>>> type nombreFuncion
```

nos muestra en la consola el contenido de la función nombreFuncion.

3. Ejecución condicional de comandos.

El comando if permite la ejecución de un comando condicionado al valor de una variable. Así, por ejemplo, la siguiente función permite determinar el menor de dos números: (Pinchar aquí para descargar el archivo .m)

```
function [m]=menor(a,b)
% Calcula el menor de dos números
    if a<b
        m=a
    else
        m=b
    end %if
end %function
```

(nótese el empleo de end %if o end %function con el texto tras el símbolo % simplemente indicando qué estructura es la que se cierra con el end). Si se van a realizar varias comparaciones, se utiliza elseif. Por ejemplo, para construir una función que, aplicada a un vector v, devuelva el valor de v si éste tiene dimensión 1, la suma de los cuadrados de sus componentes si tiene dimensión 2 y la suma de los cubos de sus componentes si tiene dimensión 3 o más, podemos utilizar el siguiente código: (Pinchar aquí para descargar el archivo .m)

```
function [m]=transforma(v)
    if length(v)==1
        m=v;
    elseif length(v)==2
        m=sum(v.^2);
    else
        m=sum(v.^3);
    end %if
end %function
```

Obsérvese en este código que:

- length(v) nos devuelve la dimensión del vector.
- La comparación se realiza con el doble signo '=='
- Para elevar a una potencia los diferentes componentes de un vector en una sóla operación utilizamos el operador .^
- elseif se escribe todo junto.

4. Bucles.

En matlab hay varios comandos que permiten implementar bucles. Un bucle es una secuencia de comandos que se repite hasta que se cumple determinada condición. El bucle más sencillo es el bucle for. Así, por ejemplo, para calcular la suma de todos los números enteros de 1 a n podríamos utilizar la siguiente función: (Pinchar aquí para descargar el archivo .m)

```
function [s] = sumaEnterosF(n)
% Suma los enteros de 1 a n
    s=0;
    for i=1:n
```

```
        s=s+i;
    end % for
end % function
```

Podemos utilizar también un bucle while para realizar la misma tarea: (Pinchar aquí para descargar el archivo .m)

```
function [s] = sumaEnterosW(n)
% Suma los enteros de 1 a n
    s=0;
    i=1;
    while i<=n
        s=s+i;
        i=i+1;
    end %while
end %function
```

Dependiendo del problema a resolver será más sencilla la implementación de uno u otro tipo de bucle. En general, cuando se sabe *a priori* para qué valores del índice debe ejecutarse el bucle, es preferible un bucle for. En caso contrario deberemos optar por un bucle while.

5. Ejercicios a realizar en esta práctica.

En esta práctica estudiaremos cómo podemos aumentar la funcionalidad del lenguaje MATLAB para nuestras aplicaciones definiendo funciones. Matlab dispone de su propio editor (*medit*), que utilizaremos para escribir los m-files; una vez generado este archivo, lo guardamos y podemos llamar al *script* o función que contiene, directamente desde la línea de comandos de MATLAB como si fuera cualquier otro comando de este lenguaje (una vez establecido el directorio).

1. La siguiente función calcula la parábola $y = x^2$ entre n y $-n$ a intervalos de $0{,}1$ (Nota: el comando grid superpone una malla rectangular a la gráfica, lo que permite identificar mejor sus valores): (Pinchar aquí para descargar el archivo .m)

```
function y = parabola(n)
% Calcula la función y=x^2
% La función se representa entre los valores -n y n
    x=-n:0.1:n;
    y=x.^2;
```

```
    plot(x,y);
    grid;
end % function
```

a) Comprobar el funcionamiento para varios valores de n.

b) Modificar el código anterior para que calcule y represente la función $y = x^p$ entre 0 y n, donde p puede ser cualquier valor positivo.

Solución:

Bastará añadir *p* como argumento, y modificar la línea en que se calcula el valor de y : (Pinchar aquí para descargar el archivo .m)

```
function y = potencia(n,p)
  % Calcula la función y=x^p
  % La función se representa entre los valores 0 y n
  x=0:0.1:n;
  y=x.^p;
  plot(x,y);
  grid;
end % function
```

2. La presión atmosférica (p) varía en función de la altura (h) según la siguiente expresión: $p = 1035e^{-0,12h}$, donde la altura se mide en kilómetros y la presión en milibares.

   a) Escribir una función presion(h) que calcule la presión para una altura h dada (utilizar la función de MATLAB exp()).

Solución:

La función podría ser, simplemente: (Pinchar aquí para descargar el archivo .m)

```
function p = presion(h)
```

```
% Calcula la presión (en milibares) a una altura h  (en kilómetros)
% segun la expresión p=1035exp(-0.12h)
    p=1035*exp(-0.12*h);
end % function
```

Para llamar a esta función y obtener la presión a una altura de 2 km, por ejemplo, podemos ejecutar en la linea de comandos:

```
>>> presion(2)
ans =  814.16
```

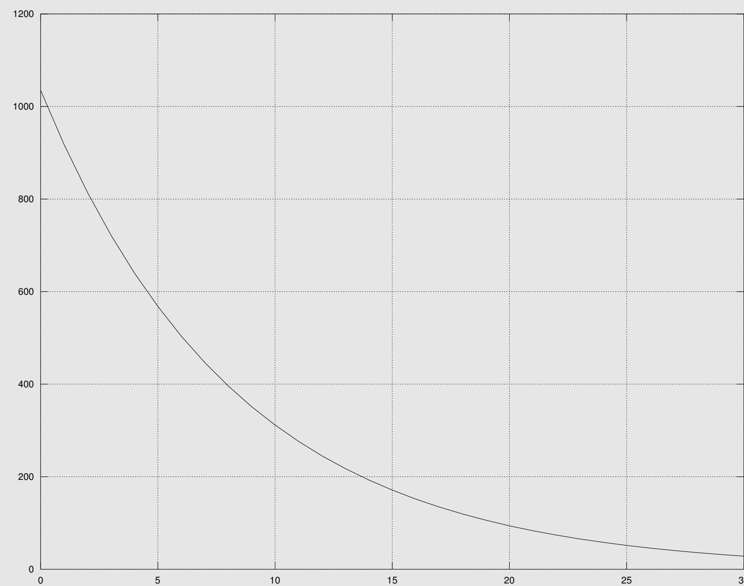
a) Queremos ahora observar en una gráfica cómo varía la presión en función de la altura. Escribir los comandos MATLAB necesarios para:

- Definir a_vec como un vector de alturas que toma los valores enteros entre 0 km y 30 km;
- Aplicar la función que hemos escrito para obtener los valores del vector de presiones p_vec;
- Dibujar la gráfica con la altura en las abscisas y la presión en las ordenadas.

Solución:
El siguiente *script* llevaría a cabo las tareas señaladas:(Pinchar aquí para descargar el archivo .m)

```
a_vec=0:30
p_vec=presion(a_vec)
plot(a_vec,p_vec)
grid
print -deps presiones.eps
```

El último comando (print -deps presiones.eps) genera un archivo postscript con la gráfica, que podemos incluir en nuestros documentos. La siguiente figura muestra la gráfica obtenida.



3. Diseñar una función en MATLAB que calcule la superficie y volumen de un cilindro dados su radio (r) y altura (h) $(V = \pi^2 rh,\ S = 2\pi r^2 + 2\pi rh)$.

Solución:

(Pinchar aquí para descargar el archivo .m)

```
function [sup,vol] = supvol(r,h)
% Calcula la superficie y volumen de un cilindro de radio r y ↩
   altura h
     sup=2*pi*r^2+2*pi*r*h;
     vol=r*h*pi^2;
end % function
```

Para llamar a esta función y obtener los valores numéricos para un cilindro de radio 2 y altura 10 por ejemplo, podemos ejecutar en la linea de comandos:

    >>> [s,v]=supvol(2,10)

```
s =  150.80
v =  197.39
```

4. La resistencia total ofrecida por 3 resistencias es:

   - $R_T = R_1 + R_2 + R_3$ si se conectan en serie, y
   - $1/R_T = 1/R_1 + 1/R_2 + 1/R_3$ si se conectan en paralelo.

   Escribir una función tal que dadas 3 resistencias calcule la resistencia total cuando se conectan en paralelo y en serie.

Solución:
La función podría ser de la forma:(Pinchar aquí para descargar el archivo .m)

```
function [serie,paralelo] = resistencia(R1,R2,R3)
% Calcula la resistencia total de tres resistencias en serie
%  y en paralelo.
     serie=R1+R2+R3;
     paralelo=1/R1+1/R2+1/R3;
end % function
```

Para llamar a esta función y obtener los valores numéricos para tres resistencias de 10,20 y 30 Ω por ejemplo, podemos ejecutar en la linea de comandos:

```
>>> [ser,paral]=resistencia(10,20,30)
ser =  60
paral =  0.18333
```

5. La función mod(a,b) devuelve el resto de dividir a por b. Escribir una función que, utilizando mod() nos diga si un entero arbitrario es par o impar.

Solución:

La función podría ser de la forma: (Pinchar aquí para descargar el archivo .m)

```
function [tipo] = es_par(valor)
    if valor-floor(valor)>0
        disp('El número no es entero.')
    elseif mod(valor,2)==0
        tipo='SI';
    else
        tipo='NO';
    end % if
end % function
```

La función determina primero si el valor que se le pasa como argumento es o no entero (ya que si tuviera dígitos decimales no sería par ni impar). Para ello simplemente resta al número valor su parte entera floor(valor). Si la diferencia es mayor que cero entonces se muestra el mensaje de que el número no es entero. En caso contrario se calcula el resto de dividir por 2 mediante el comando mod(valor,2): si el resto es cero, el número es par y en caso contrario es impar. Vemos algunos ejemplos de llamada a esta función:

```
>>> es_par(4)
ans=SI
>>> es_par(127)
ans=NO
>>> es_par(8.52)
El número no es entero.
```

6. Utilizando la estructura if-elseif construir un programa que determine el mayor de 3 números a, b y c.

Solución:

Una forma sencilla de implementar esta tarea es la siguiente: (Pinchar aquí para descargar el archivo .m)

```
function [M] = mayor_de_tres(a,b,c)
```

```
% Esta funcion identifica al mayor de tres numeros a, b y c
    if a>b & a>c
        M=a;
    elseif b>a & b>c
        M=b;
    else
        M=c;
    end %if
end % function
```

El operador & significa “y”. De esta forma la función comprueba si a es mayor que b y a es mayor que c, en cuyo caso el máximo es a. En cambio si b es mayor que a y b es mayor que c entonces el máximo es b; y si no ocurre ninguna de las condiciones anteriores, el máximo es c. Algunos ejemplos de llamada a esta función:

```
>>> mayor_de_tres(1,5,7)
ans =  7
>>> mayor_de_tres(10,5,9)
ans =  10
```