Capítulo 1

Introducción a Matlab

1.1. Resumen de la práctica

El objetivo de esta práctica es familiarizarse con Matlab, una herramienta de cálculo asistido por ordenador, y especialmente con el subconjunto de rutinas específicas de control automático (la *Control toolbox*). Matlab proporciona un entorno al usuario que facilita enormemente el análisis, diseño y simulación de sistemas de control, al incluir una serie de rutinas que resuelven los cálculos matemáticos de fondo, junto con una sencilla interfase para su uso.

1.2. El programa Matlab de Mathworks Inc.

`Matlab`, `Mathemática` y `Maple` son los programas de matemáticas asistidaa por ordenador más difundidos. Son sencillos, versátiles y fáciles de usarñ. Por ello se emplean en disciplinas tan diversas como Estadística, Visión Artificial, Robótica, Redes Neuronales, Análisis de Elementos Finitos, Finanzas, Ingeniería de Control y un largo etcétera.

Matlab incorpora gran número de funciones de caracter general, y otras más especializadas agrupadas en *toolboxes*. El nombre Matlab viene de laboratorio de matrices (*MATrix LABoratory*). En Matlab podemos trabajar de manera natural con tipos de datos tales como escalares, booleanos, vectores, matrices y polinomios, manejando indistintamente elementos reales o complejos.

Matlab permite además, de manera muy simple, la creación de funciones definidas por el usuario, incluyendo operadores condicionales, iterativas y secuenciales (Matlab dispone de palabras clave del tipo de `if`, `while`, `else` o `for`).

Además Matlab incluye una amplia gama de operadores y funciones predefinidas entre las que se incluyen:

- Funciones matemáticas elementales (trigonométricas, exponenciales, logarítmicas, etc.)
- Funciones elementales de manipulación de matrices
- Funciones para análisis de datos y transformadas de Fourier (medias de vectores, covarianzas, ordenación de datos, etc.)
- Funciones polinomiales y de interpolación.
- Funciones para gráficos 2-D y 3-D

- ➢ Funciones de Entrada/Salida para almecenar y recuperar datos en disco
- ➢ Un extenso juego de operadores (lógicos, escalares, matriciales, etc.)
- ➢ Existen, además, muchas otras funciones especializadas para trabajar con matrices dispersas, resolución de ecuaciones diferenciales etc.

Además de las funciones predefinidas, existen paquetes de herramientas matemáticas, denominados *toolboxes* que contienen colecciones de funciones de Matlab –incluyendo a menudo demostraciones y tutoriales– destinadas a temas específicos: Control, Procesamiento Digital de Señal, Algoritmos Genéticos, Redes Neuronales, etc.

Nuestro objetivo será apoyarnos en Matlab para facilitar el analizar y simular señales y sistemas, y el diseñar para estos últimos algoritmos de control.

1.3. Guión de la práctica

1.3.1. Instrucciones

Lanzar el Matlab. Aparece el siguiente mensaje:

```
Commands to get started: intro, demo, help help
Commands for more information: help, whatsnew, info, subscribe

>>
```

La mejor forma de usar esta introducción es recorrerla desde el principio hasta el final. Esta se escribió con abundantes ejemplos que el lector puede teclear directamente para comprobar por sí mismo su funcionamiento. Es muy recomendable –de hecho ello forma parte de la filosofía de esta introducción– jugar con los ejemplos que aquí se muestran, modificándolos o adaptándolos a un problema concreto, para ir adquiriendo habilidades mediante la experiencia.

La práctica consta de 4 partes:

- ➢ Tipos de datos básicos
- ➢ Programación
- ➢ Gráficos 2-D
- ➢ Una sesión de trabajo en Matlab

1.3.2. Elementos básicos en Matlab

Escalares: Uno de los elementos más simples que Matlab es capaz de manejar son los escalares. Estos pueden ser reales o complejos. Para crear un escalar podemos proceder de la siguiente manera

```
>> a = 1

a =

     1
```

Para introducir complejos, Matlab lleva definidos los elementos `i` y `j`, que denotan ambos el valor $\sqrt{-1}$. Así, es posible hacer, por ejemplo,

```
>> b = 1 + 2*j

b =

   1.0000 + 2.0000i
```

Introducir expresiones como $ae^{(a+b)(1+2*i)}$

```
>> a*exp((a + b)*(1 + 2*i))

ans =

   0.1299 - 0.0378i
```

Vectores: Para crear un vector fila, basta con introducir sus elementos separados con espacios entre corchetes

```
>> a = [1 2 1 5 6 5 7 4 5]

a =

     1     2     1     5     6     5     7     4     5
```

Un vector columna se crea de la misma manera pero separando cada elemento mediante un punto y coma ";"

```
>> c = [1; 2; 4; 3; -2]

c =

     1
     2
     4
     3
    -2
```

o bien creando un vector fila y trasponiéndolo mediante el operador transposición "'"

```
>> c = [1 2 4 3 -2]'

c =

     1
     2
     4
     3
    -2
```

Existen otras formas de crear vectores. Supongamos que pretendemos crear un vector con elementos espaciados en incrementos de 2 entre 0 y 20 (ambos inclusive)

```
>> t = 0:2:20

t =

     0     2     4     6     8    10    12    14    16    18    20
```

Por supuesto, es posible utilizar `a:p:b` donde `a` es el límite inferior, `b` el límite superior y `p` el "paso" o incremento, que puede tomar cualquier valor: entero positivo, entero negativo, flotante positivo o flotante negativo (no admite incrementos complejos). En el siguiente ejemplo, se ha utilizado un incremento flotante negativo:

```
>> t = 4.6:-0.1:3.3

t =

  Columns 1 through 7

    4.6000   4.5000   4.4000   4.3000   4.2000   4.1000   4.0000

  Columns 8 through 14

    3.9000   3.8000   3.7000   3.6000   3.5000   3.4000   3.3000
```

Otra posibilidad sería utilizar la función `linspace(a,b,n)` para crear, por ejemplo, un vector con 15 elementos equiespaciados desde el 0 hasta el 20 (ambos inclusive)

```
>> t = linspace(0,20,15)

t =

  Columns 1 through 7

         0   1.4286   2.8571   4.2857   5.7143   7.1429   8.5714

  Columns 8 through 14

   10.0000   11.4286  12.8571  14.2857  15.7143  17.1429  18.5714

  Column 15

   20.0000
```

Estos tres últimos ejemplos son útiles para crear bases de tiempos o abscisas en graficos X-Y.

La función `linspace(a,b,n)` permite todo tipo de valores para los límites `a` y `b`, incluso complejos. En éste último caso, genera una trayectoria compleja lineal, formada por `n` puntos entre `a` y `b`, (como antes, ambos inclusive).

También resulta sencillo generar vectores a partir de otros mediante el uso de expresiones:

```
>> b = 2*a + 5

b =

     7     9     7    15    17    15    19    13    15
```

Existe también la posibilidad de realizar operaciones elemento a elemento añadiendo un punto "." delante del operador, con lo que tendríamos operadores ".+", ".*", ".^", etc. Así, por ejemplo,

```
>> a = [1 2 3 4]

a =

     1     2     3     4

>> b = [2 2 3 3]

b =

     2     2     3     3

>> a.*b

ans =

     2     4     9    12
```

Matrices: Definir matrices es tan sencillo como definir vectores. El elemento ";" equivale a decir "nueva fila". Así

```
>> a = [1 2 3; 1 1 1; 2 3 4; 5 5 5]

a =

     1     2     3
     1     1     1
     2     3     4
     5     5     5
```

Operar con matrices es tan simple como hacerlo con escalares – siempre que las dimensiones de los operandos sean coherentes, es decir, no está permitido multiplicar una matriz de 3×2 por una de 5×5–

```
>> a = [1 2 3 4]

a =

     1     2     3     4

>> b = [2 3 2 1; 2 2 1 2; 1 1 1 1; 3 4 2 1]
```

```
b =

     2     3     2     1
     2     2     1     2
     1     1     1     1
     3     4     2     1

>> c = inv(a*a' + 2*b)

c =

   -1.5000   -0.5000    1.0313    1.0000
    1.0000    0.5000   -1.0313   -0.5000
    0.0000   -0.5000    0.5313    0.0000
    0.5000    0.5000   -0.5000   -0.5000
```

En el caso de matrices cuadradas, es posible utilizar el operador exponencial "^"

```
>> A = [1 0.5; 0.5 2]

A =

    1.0000    0.5000
    0.5000    2.0000

>> A^3

ans =

    2.0000    3.6250
    3.6250    9.2500
```

Puede comprobarse que cuando introducimos un exponente negativo, se produce la correspondiente inversión, es decir

$$\mathbf{A}^{-n} = (\mathbf{A}^{-1})^n \tag{1.1}$$

Como caso particular, podemos invertir la matriz sin utilizar la función `inv()` hallando $\mathbf{A}^{-1}$.

Resulta de gran utilidad la función `eig()`, que obtiene los valores y vectores propios de una matriz.

```
>> [v,d]=eig(A)

v =

    0.9239    0.3827
   -0.3827    0.9239


d =
```

```
    0.7929         0
         0    2.2071
```

donde las columnas de `v` son los vectores propios y los elementos de la diagonal de `d` son los valores propios correspondientes. También es posible llamar a `eig()` utilizando un único argumento de salida. En este caso devuelve sólamente los valores propios.

```
>> eig(A)

ans =

    0.7929
    2.2071
```

Al igual que con los vectores existe la posibilidad de aplicar los operadores `.+` y `.*`, etc. , que trabajan elemento a elemento de la matriz.

Matlab dispone de algunas funciones para crear matrices de uso común que nos pueden ser muy útiles. En la tabla 1.1 están expuestas las más frecuentes.

función	resultado
`eye(n)`	matriz identidad de tamaño $n \times n$
`diag(vector)`	matriz diagonal con los elementos de `vector`
`zeros(m,n)`	matriz de ceros de tamaño $m \times n$
`ones(m,n)`	matriz de unos de tamaño $m \times n$
`rand(m,n)`	matriz de aleatorios (dist. $\mathbf{U}(0,1)$) de tamaño $m \times n$
`randn(m,n)`	matriz de aleatorios (dist. $\mathbf{N}(0,1)$) de tamaño $m \times n$

Cuadro 1.1: Tabla de funciones para generar matrices.

No obstante, Matlab (en especial la versión 5.0) dispone de otras muchas funciones para generar matrices típicas, tales como

```
triu, tril, hilb, magic, toeplitz
```

capaces de crear matrices triangulares (superior e inferior), de Hilbert, mágicas y de Toeplitz, respectivamente. El usuario interesado puede utilizar `help` para más información.

Tanto vectores como polinomios son casos particulares de las matrices: considerados como objetos, Matlab los trata por igual. Todo lo que aquí se diga para matrices es, por tanto, válido para vectores, considerados éstos como matrices. Supóngase la siguiente matriz

```
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]

A =
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Para realizar un acceso de lectura, por ejemplo, al elemento de la fila 2 y la columna 3, (operación $a \leftarrow a_{23}$) basta con introducir

```
>> a = A(2,3)

a =

     7
```

Para realizar un acceso de escritura al mismo elemento ($a_{23} \leftarrow 4$) basta con teclear

```
>> A(2,3) = 4

A =

     1     2     3     4
     5     6     4     8
     9    10    11    12
    13    14    15    16
```

Los vectores pueden considerarse como matrices de tamaño $1 \times n$ si son vectores fila, o de tamaño $n \times 1$ si se trata de vectores columna, por lo que la forma de proceder sería análoga. Es posible, sin embargo acceder a ellos de una forma más corta

```
>> v = [1 2 3 4 5]

v =

     1     2     3     4     5

>> v(4)

ans =

     4
```

De una manera totalmente natural, podemos acceder a submatrices (en particular, a subvectores) empleando rangos. Un rango no es sino un vector cuyos elementos indican posiciones de elementos; dicho de otra forma, se trata de un conjunto de índices.

```
>> A([1 2],[1 3 4])

ans =

     1     3     4
     5     7     8
```

como puede comprobarse, se trata de la matriz

$$(a_{ij}) \qquad i \in \{1,2\}, \quad j \in \{1,3,4\} \tag{1.2}$$

o sea,

$$\begin{pmatrix} a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} \end{pmatrix} \tag{1.3}$$

es decir, el rango `[1 2]` señala las filas, mientras que el rango `[1 3 4]` señala las columnas. Dicho de otra forma, se crea una matriz de 2×3 cuyos elementos son aquellos cuyos índices son el producto cartesiano de ambos rangos.

La forma más frecuente de utilizar los rangos es empleando vectores con elementos consecutivos (el ejemplo anterior, más genérico, utiliza rangos en los que los elementos están salteados):

```
>> A(2:4,2:4)

ans =

     6     7     8
    10    11    12
    14    15    16
```

lo cual generaría la matriz adjunta del elemento a_{11}.

También es posible asignar valores a submatrices de una manera sencilla (siempre cuidando de que a la submatriz se le asigne una matriz de la misma dimensión), como se muestra en el siguiente ejemplo:

```
>> A = ones(7,7)

A =

     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1

>> A(3:5,2:6) = randn(3,5)

A =

    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.1650    0.3516    0.0591    0.8717    1.2460    1.0000
    1.0000    0.6268   -0.6965    1.7971   -1.4462   -0.6390    1.0000
    1.0000    0.0751    1.6961    0.2641   -0.7012    0.5774    1.0000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
```

Polinomios: Los polinomios en Matlab se expresan mediante vectores. Cada elemento del vector representa un coeficiente del polinomio y éstos se almacenan en orden descendente dentro del mismo. Así, si queremos representar el polinomio

$$s^5 + 2s^4 - 3s^3 + 3s^2 + s - 2 \tag{1.4}$$

deberemos crear el siguiente vector (Nota: en Matlab el carácter ";" situado al final evita que salga repetido lo que hemos escrito).

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

Matlab interpreta un vector de longitud $n+1$ como un polinomio de orden n. Por ello, si el polinomio tiene coeficientes nulos, deben añadirse ceros en los lugares correspondientes. Por ejemplo, el polinomio

$$s^2 + 1 \tag{1.5}$$

debe introducirse en Matlab como

```
>> b = [1 0 1];
```

Puede hallarse el valor de un polinomio utilizando la función `polyval()`. Supongamos que queremos hallar el valor del polinomio anterior en $s = 1{,}5$

```
>> z = polyval(b,1.5)

z =

    3.2500
```

Obtener las raíces de un polinomio es inmediato. Por ejemplo, para obtener las raíces del polinomio (1.4)

```
>> r = roots(a)

r =

  -3.1866
   0.5661 + 0.9536i
   0.5661 - 0.9536i
   0.7421
  -0.6877
```

La operación inversa a `roots()` es `poly()`. Esta función toma los ceros de un polinomio devolviendo sus coeficientes

```
>> poly(r)

ans =

  Columns 1 through 4

   1.0000 2.0000  -3.0000+0.0000i 3.0000-0.0000i

  Columns 5 through 6

   1.0000-0.0000i -2.0000+0.0000i
```

Para multiplicar dos polinomios, podemos utilizar la función `conv()`. El vector de coeficientes del polinomio producto $A(s)B(s)$ es el resultado de la convolución de los vectores de coeficientes de $A(s)$ y $B(s)$. De ahí que utilicemos dicha función, como se muestra en el siguiente ejemplo:

$$A(s) = s^2 + 1 \tag{1.6}$$
$$B(s) = s^2 + 2s + 2 \tag{1.7}$$

```
>> a = [1 0 1];
>> b = [1 2 2];
>> c = conv(a,b)

c =

     1     2     3     2     2
```

lo que significa que

$$C(s) = A(s)B(s) = s^4 + 2s^3 + 3s^2 + 2s + 2 \tag{1.8}$$

La función `deconv` permite dividir dos polinomios. Esta función devuelve el cociente y el resto de la división. Utilizando los polinomios del ejemplo anterior podemos hacer

```
>> [Q,R]=deconv(c,a)

Q =

     1     2     2


R =

     0     0     0     0     0
```

como se puede comprobar, $A(s)$ divide a $C(s)$, como era de esperar.

Para sumar y restar polinomios, dado que estos se representan en realidad como vectores deben ser del mismo grado. Si esto no

fuese así deberán añadirse ceros a la izquierda en el polinomio de orden más bajo hasta igualar el grado con el de orden superior. En el siguiente ejemplo se suman $A(s)$ y $C(s)$, polinomios que tienen distinto grado:

```
>> a = [0 0 a]

a =

     0     0     1     0     1

>> d = a + c

d =

     1     2     4     2     3
```

Cadenas alfanuméricas: Matlab permite también el uso de cadenas alfanuméricas. Una cadena alfanumérica se declara mediante el uso de comillas simples:

```
>> a = 'Hola'

a =

Hola
```

Realmente, una cadena alfanumérica es considerada por Matlab como un caso particular de matriz, como se ve en el siguiente ejemplo

```
>> a = 'Hola'

a =

Hola

>> b = 'Pepe'

b =

Pepe

>> c = [a b]

c =

HolaPepe

>> d = [a;b]

d =

Hola
Pepe
```

```
>> size(a), size(b), size(c), size(d)

ans =

     1     4


ans =

     1     4


ans =

     1     8


ans =

     2     4
```

Además, internamente, cada elemento (cada carácter de la cadena) es almacenado como un entero con su número ascii:

```
>> a .* [1 1 1 1]

ans =

    72   111   108    97
```

Nótese que hemos multiplicado cada carácter por 1 (operador "miembro a miembro" `.*`), lo que nos da los códigos ascii de los caracteres `H, o, l, a`.

1.3.3. Programación en Matlab

Funciones y scripts: Hasta ahora hemos manejado Matlab como un intérprete, donde las órdenes se introducen a través de una línea de comandos. Matlab permite la creación de funciones y scripts. Ambos son archivos de tipo texto con extensión `*.m` que contienen código de Matlab. Para ser ejecutados deben estar en el directorio actual de trabajo de Matlab, o bien estar en uno de los directorios especificados por la variable interna `matlabpath` (Nota: Matlab admite los comandos `pwd` y `cd` de UNIX, para ver y cambiar el directorio de trabajo por defecto).

Las funciones en Matlab, al igual que las de otros lenguajes como Pascal o C, son trozos de código que admiten una serie de parámetros (entradas) y devuelven una serie de resultados (salidas). Todas las variables utilizadas por las funciones son locales, es decir, una vez finalizada la ejecución de la función desaparecen. La ejecución de las funciones no afecta, pues, al espacio de trabajo que utilizamos al trabajar desde la línea de comandos.

Los scripts, contienen código de Matlab, pero ni reciben ni devuelven parámetros. Todos los objetos que se utilicen en los scripts

pertenecen al espacio de trabajo de Matlab. El script trabaja, por tanto, con variables globales. Realmente, no existe diferencia alguna entre trabajar desde la línea de comandos o hacerlo mediante scripts.

Funciones: Una función debe almacenarse en un archivo de tipo texto con extensión `*.m`. Para crear un archivo `*.m` cómodamente desde Matlab podemos seleccionar en la barra de menú `File\New\m-file`. Automáticamente lanzará el editor que hayamos elegido para Matlab en el menú de opciones. Por defecto lanza el `notepad.exe` abriendo un archivo con la extensión `*.m`

Una función de Matlab tiene la siguiente apariencia

```
function [a,b] = ejemplo (c, d, e)

% [a,b] = ejemplo (c, d, e)
%
% PARAMETROS DE ENTRADA:
% c: Parametro 1
% d: Parametro 2
% e: Parametro 3
%
% PARAMETROS DE SALIDA:
% a: Parametro 1
% b: Parametro 2

a = c + d + e;
b = c * d * e;
```

Cabecera. Como puede apreciarse, esta función recibe tres parámetros (`c,d,e`) y devuelve dos (`a,b`). Otros casos posibles serían los siguientes:

si la función no recibiese ningún parámetro,

```
function [a,b] = ejemplo ()
```

si la función devuelve sólo un parámetro, pueden omitirse los corchetes, tal y como se muestra a continuación

```
function a = ejemplo (c, d, e)
```

Comentarios. Los comentarios en Matlab son siempre de línea: todo lo que vaya desde el símbolo % hasta el final de la línea es ignorado.

En Matlab el primer bloque de comentario (entendiendo por "bloque" varias líneas de comentario que no estén separadas por una línea que no sea de comentario) se muestra cuando desde la línea de comandos tecleamos `help` seguido del nombre de la función. Así, en nuestro ejemplo veríamos esto:

```
>> help ejemplo

  [a,b] = ejemplo (c, d, e)

  PARAMETROS DE ENTRADA:
  c: Parametro 1
  d: Parametro 2
  e: Parametro 3

  PARAMETROS DE SALIDA:
  a: Parametro 1
  b: Parametro 2
```

Es muy recomendable incluir un bloque de comentarios parecido a éste, explicando qué hace cada parámetro y describiendo después qué hace la función, para que el usuario sepa cómo llamar a la función.

Número de argumentos variable. Matlab incluye, asimismo, la posibilidad de utilizar un número variable de argumentos en sus funciones mediante la variable `nargin` que nos indica el número de argumentos. Mediante el simple chequeo del número de argumentos podemos hacer que la función se comporte de una u otra manera según el número de éstos. Lo más típico es hacer que la función asigne valores por defecto a los argumentos omitidos, tal y como se muestra en el siguiente ejemplo, que consiste en una función (`variarg.m`) con número de argumentos variable:

```
function y = variarg(a,b,c,d,e)

% y = variarg(a,b,c,d,e)
%
% PARAMETROS DE ENTRADA:
% a: (...descripcion del parametro a)
% b: (...descripcion del parametro b)
% c: (...descripcion del parametro c)
% d: (...descripcion del parametro d)
% e: (...descripcion del parametro e)
%
% PARAMETROS DE SALIDA:
% y: (...descripcion de la salida y)


if nargin < 5
   e = 10000;
end

if nargin < 4
   d = 1000;
end

if nargin < 3
   c = 100;
```

```
end

y = a + b + c + d + e;
```

Al llamarla desde la línea de comandos se comporta de la siguiente manera:

```
>> variarg(1,2)

ans =

       11103

>> variarg(1,2,3)

ans =

       11006

>> variarg(1,2,3,4)

ans =

       10010

>> variarg(1,2,3,4,5)

ans =

    15
```

Nótese que la función del ejemplo contempla el caso de 2, 3, 4 y 5 argumentos, pero si el número de argumentos al llamarla es menor que 2 dará el siguiente mensaje de error:

```
>> variarg(1)
??? Input argument b is undefined.

Error in ==> c:\misdoc~1\tesis\docencia\ri\matlab\tex\variarg.m
On line 28  ==> y = a + b + c + d + e;
```

Como puede apreciarse, el motivo del error es que el argumento b no ha sido definido de ninguna manera al llegar a la línea 28: ni fue definido mediante su introducción en los parámetros al llamar la función, ni lo fue en el código de la función, en el que no se da un valor a dicho parámetro.

Scripts: El procedimiento para crear un script es idéntico al descrito para las funciones. La única diferencia con las funciones es que el script no lleva la línea de parámetros. La simple ausencia de ésta en un archivo `*.m` hace que Matlab lo trate como un script y asuma, por tanto, que todas las variables son globales. Los scripts suelen utilizarse para realizar demostraciones o "películas" del proceso de resolución de algún problema, para la ejecución de varias órdenes seguidas que no nos quepan en una línea de comandos de Matlab, etc.

Estructuras de control en Matlab: Matlab, como lenguaje estructurado, emplea tres tipos de estructuras de control: secuencial, iterativa y condicional.

El código de Matlab se asemeja mucho a otros como el C o el Pascal en su estructura aunque, evidentemente, de mucho más alto nivel. El flujo de control en el código de Matlab puede definirse de las tres maneras.

Secuencial La secuencial es, quizás, la más sencilla. El simple hecho de llamar a una función despues de otra implica una estructura secuencial. Para las estructuras iterativas y condicionales, Matlab trabaja de forma muy similar a los lenguajes típicamente estructurados como C o Pascal.

Iterativa Para repetir un bloque de código, Matlab utiliza la siguiente estructura

```
for variable = expresion, bloque_de_codigo, end
```

Veamos su funcionamiento con un ejemplo. Supongamos que pretendemos hallar la siguiente expresión

$$\mathbf{B} = \sum_{i=1}^{i=10} \mathbf{A}^i \tag{1.9}$$

Podríamos meter el siguiente trozo de código en un script y luego ejecutarlo:

```
B = zeros(2,2);
A = [1 .5;.5 2];
for i=1:10
    B = B + A^i;
end;
```

O bien meterlo todo en una línea y darle a `return`:

```
B = zeros(2,2); A = [1 .5;.5 2]; for i=1:10 B = B + A^i; end;
```

En ambos casos, el resultado sería idéntico. Lo más normal es que, como en el ejemplo anterior, el vector contenga enteros consecutivos, aunque esto no tiene por qué ser así. La siguiente pieza de código

```
B = zeros(2,2);
A = [1 .5;.5 2];
for i=[1 2 2.5 10]
    B = B + A^i;
end;
```

Operador	Símbolo
OR	|
AND	&
XOR	xor
igual	==
no igual	~=
menor que	<
menor o igual que	<=
mayor que	>
mayor o igual que	>=

Cuadro 1.2: Tabla de operadores booleanos de Matlab

ejecutaría

$$\mathbf{B} = \sum_{i=\{1,2,2,5,10\}} \mathbf{A}^i \tag{1.10}$$

Nótese que en todos los ejemplos la variable índice va tomando los valores de las columnas de la expresión que se le asigna. Al ser dicha expresión un vector fila, sus columnas son escalares con lo que la variable `i` va a ser siempre un escalar para el código que se ejecuta en cada iteración.

De una manera más general –aunque mucho más infrecuente y casi nunca necesaria–, la expresión asignada a `i` puede ser una matriz de $n \times m$, con lo que tendríamos m iteraciones en cada una de las cuales la variable `i` sería un vector columna de n elementos.

Condicional Existen dos estructuras de control condicionales en Matlab: `if` y `while`. Ambas utilizan expresiones booleanas. Estas se consideran falsas si su valor es 0 y ciertas en caso contrario[1]. Los operadores (lógicos y relacionales) utilizados en este tipo de expresiones son similares a los utilizados en los lenguajes de programación típicos:

La primera es muy similar a la de la mayoría de los lenguajes estructurados. Veamos con un ejemplo cómo se ejecuta:

```
if i == j
   a(i,j) = 2;
   b(i,j) = 3;
elseif abs(i-j) == 1
   a(i,j) = -1;
```

[1] Aunque cualquier valor distinto de 0 es interpretado como “cierto”, el valor que Matlab asigna a una expresión verdadera es el 1

```
   b(i,j) = -5;
else
   a(i,j) = 0;
   b(i,j) = 0;
   c      = 1;
end
```

En el siguiente ejemplo, relativo a la estructura `while`, se ejecuta el bloque de código mientras se cumpla la condición `d<10`

```
i = 1.1;
c = 0;
d = 1
while d < 10
   c = c + i;
   d = d * i;
end
```

1.3.4. Gráficos 2-D en Matlab

Función **plot()**: Sin duda, la que más nos interesa para la mayoría de las aplicaciones es la función `plot(x,y)`. Esta función es capaz de representar de múltiples formas en un gráfico 2-D los datos que se introducen por medio de los vectores `x` e `y`. Veámoslo con un ejemplo:

```
t = linspace(0,3,200);  % Generamos un vector de tiempos [0,3s]
s = sin(2*pi*t + pi/6); % Hallamos la funcion seno
plot(t,s);              % Trazamos el grafico mediante plot()
```

Esta función representará en abscisas un vector de tiempos formado por 200 puntos equidistantes entre 0 y 3 (ambos inclusive), mientras que en ordenadas una senoide de pulsación $w = 2\pi$, es decir, de frecuencia $f = 1$Hz, para los tiempos dados.

Existe también la posibilidad de llamarla con un vector como único argumento

```
plot(s);
```

lo que equivale a poner

```
plot(1:length(s),s);
```

es decir, asume una escala lineal en abscisas que representa el orden que ocupa en el vector `s` cada punto mostrado.

Por defecto, Matlab une con líneas los puntos trazados, dando continuidad al gráfico; sin embargo, otras veces nos interesará dibujar los puntos de manera aislada, sin unir. En el siguiente ejemplo, se muestra uno de estos casos:

```
t = linspace(0,3,200);   % Generamos un vector de tiempos [0,3s]
s = sin(2*pi*t + pi/6);  % Hallamos la funcion seno
sr= s + 0.2*randn(1,200) % Superponemos algo de ruido
plot(t,s);               % Trazamos 's'  (grafico continuo)
hold on;                 % lo mantenemos (evitamos que se borre)
plot(t,sr,'.');          % Trazamos 'sr' (grafico con puntos)
hold off;                % Trazamos el grafico mediante plot()
```

En este ejemplo, se ha generado la misma senoide que en el ejemplo anterior, pero se le ha añadido ruido aleatorio mediante la función `randn(n,m)`, que genera una matriz de $n \times m$ elementos aleatorios. La función `plot(x1,y1,s1)`, donde `s1` es una cadena que sirve para especificar el tipo y color del gráfico, se ha usado aquí para representar dos gráficos en uno sólo.

Algunos de los tipos y colores posibles de gráficos son:

y	yellow	.	point
m	magenta	o	circle
c	cyan	x	x-mark
r	red	+	plus

Pueden combinarse tipos con colores y viceversa poniendo los códigos juntos en la cadena. Así, `plot(x,y,'rx')` trazará los puntos (x, y) como cruces rojas, `plot(x,y,'g-.')` trazará el mismo gráfico pero con un trazo verde de puntos y rayas, etc.

También es posible trazar varios gráficos con una sola orden. El siguiente ejemplo produciría el mismo efecto que el anterior.

```
plot(t,s,t,sr,'.');
```

Dicha orden trazaría el gráfico $t - s$ en línea continua –ya que, al no haberse especificado el tipo mediante una cadena, la toma por defecto– y trazaría también puntos (t, sr) de color amarillo –al no haberse especificado color, toma el amarillo por defecto–.

Si queremos representar un vector complejo, la función `plot()` pone en abscisas la parte real y en ordenadas la parte imaginaria. Por ejemplo:

```
t = linspace(0,3*pi,100);
y = t.*exp(j*t);
plot(y);
```

trazará los valores complejos que toma la función $f(t) = te^{jt}$ para $t \in [0, 3\pi]$, lo que muestra en pantalla una espiral.

Si uno de los parámetros de `plot(x,y)` es una matriz, se trazarán las columnas o las filas de ésta frente a los elementos del otro parámetro. Por ejemplo

```
t  = linspace(0, 2*pi, 100);    % generamos vector de tiempos
s1 = sin(t);                    % generamos un   vector senoidal
s2 = sin(3*t);                  % generamos otro vector senoidal
s  = [s1;s2]';                  % los reunimos en una matriz de
                                % 100 filas y 2 columnas
plot(t,s);                      % y lo dibujamos todo...
```

Nos aparecen en pantalla, superpuestos, los gráficos de `s1` frente a `t` y de `s2` frente a `t`.

Análogamente, si ejecutásemos `plot(s,t)`, aparecerían superpuestos los gráficos de `t` frente a `s1` y de `t` frente a `s2` (lógicamente esto nos los muestra girados 90^o).

Una instrucción que se usa mucho en Matlab es `hold on/off`. Por defecto, Matlab borra el contenido de una figura cuando va a trazar sobre ella un gráfico. Mediante `hold on`, le decimos a Matlab que no borre el contenido de la figura de manera que, todos los gráficos que se envien a dicha figura en lo sucesivo se pintan unos encima de otros: le hemos dicho a Matlab que se ponga en *modo hold*. Esto es muy útil cuando se quieren comparar gráficos. `hold off` "apaga" el *modo hold* y vuelve a permitir el borrado automático de la figura. Con un ejemplo podemos ver claramente el uso de este comando:

```
% GENERAMOS UNA BASE DE TIEMPOS [0,10s]
t = linspace(0,10,100);

% CREAMOS UN SISTEMA CON POLOS EN s=-1 y s=-2
num1 = [1];
den1 = poly([-1 -2]);

% CREAMOS UN SISTEMA CON POLOS EN s=-1+i y s=-1+i
num2 = [1];
den2 = poly([-1+i -1-i]);

step(num1,den1,t);  % RESPUESTA ESCALON DEL SISTEMA 1
hold on;        % QUEDAR EN MODO HOLD...
step(num2,den2,t);  % RESPUESTA ESCALON DEL SISTEMA 2
hold off;       % QUITAR EL MODO HOLD
grid;
```

Generación y manejo de figuras: En los ejemplos anteriores, aunque no lo hayamos especificado, hemos utilizado figuras. Una figura es una ventana que Matlab abre para poner en ella un gráfico. Cuando llamamos a la función `plot()`, ella misma abre una figura (la figura 1) si no hay ninguna abierta. Suele ser normal que a lo largo de una sesión necesitemos tener más de un gráfico a la vista. Esto se puede hacer de dos maneras:

- La primera consiste en abrir más figuras (figuras 2,3, etc.), es decir, generar ventanas nuevas para trazar en ellas sendos gráficos. Mediante la función `figure(n)` se le indica al Matlab que redirija sucesivos gráficos a la figura número n. Si esta figura no existiera, la creará.
- La segunda forma de mostrar más de un gráfico es mediante la función `subplot(m,n,j)`. Esta función, al contrario de `figure()`, no genera ventanas nuevas, sino que utiliza la ventana actual; lo que hace, por el contrario, es subdividirla en $m \times n$ celdas, y decirle a Matlab que redirija su próxima salida gráfica a la *j-ésima* celda.

Veamos todo esto mediante un ejemplo:

```
>> t = linspace(0,1,1000); w = 3*pi; phi=pi/3;
>> y1 = sin(w*t + phi); y2 = square(w*t + phi);
>> y3 = sin(w*t + phi).^2; y4 = y1.*y2;
>> subplot(2,2,1); plot(t,y1); subplot(2,2,2); plot(t,y2);
>> subplot(2,2,3); plot(t,y3); subplot(2,2,4); plot(t,y4);
```

Finalmente, bastaría con teclear, por ejemplo, `figure(2)` para generar una nueva ventana. Una instrucción `plot()` posterior trazaría en ella el gráfico. Para restablecer la figura 1 como figura por defecto, bastaría con utilizar `figure(1)`.

Títulos y etiquetas en los gráficos: Matlab permite la inserción de títulos y etiquetas en sus gráficos. Esto es posible mediante `title()`, que da título a la figura, y mediante `xlabel()` y `ylabel()`, las cuales etiquetan los ejes de abscisas y ordenadas. Todas ellas reciben como parámetro una cadena alfanumérica con el contenido de la etiqueta. Por ejemplo

```
plot  (t,y);
xlabel('t');
ylabel('y(t)');
title ('y(t) = sin(w*t+phi)');
```

Otros comandos: bar, stem, stairs, etc.: Además de `plot()` existen otras muchas funciones relacionadas con los gráficos 2-D. Básicamente, permiten hacer gráficos con distintas apariencias (barras, palos, escaleras) y su funcionamiento es muy similar al de `plot()`, aunque suelen tener menos opciones. No es el propósito de este documento describir el funcionamiento de cada una de ellas. Se recomienda al lector que experimente con ellas, para lo cual puede hacer uso de la ayuda tecleando

```
help nombre_funcion
```

A continuación se muestran algunos ejemplos:

La función `bar()` es idónea para mostrar datos estadísticos tipo histograma:

```
% ESTE EJEMPLO TRAZA EL HISTOGRAMA DE UNA MUESTRA CON DIST. NORMAL
y = randn(1,1000);  % generamos un vector aleatorio normal N(0,1)
M = 20;             % numero de tramos para el histograma
[n,p] = hist(y,M);  % hist() es una funcion de Matlab que devuelve
                    %   n: numero de elementos en cada tramo.
                    %   p: punto inicial de cada tramo  resultante
                    %      de dividir la zona entre el maximo y el
                    %      minimo de y en M trozos.
bar(p,n);           % se traza el grafico de barras
```

La función `stairs()` permite trazar señales tal y como serían muestreadas por un sample & hold, típicas en conversiones A/D - D/A.

```
% EJEMPLO DE STAIRS()
t = linspace(0,1,100);  % generamos vector de tiempos
y = sin(2*pi*t + pi/6); % generamos una senoide
stairs(t,y);            % mostramos la senoide con escaleras, en
                        % une los puntos con lineas horizontales
                        % formando 'escaleras'
```

No obstante, la función `plot()` es, con mucho, la más útil y versátil, siendo rara vez necesario recurrir a otras funciones.

Impresión de gráficos: Existen tres formas de imprimir un grafico de una figura:

- Activando la ventana de la figura, es decir, entrando en ella, y accediendo con el raton al menú según `File\Print` es posible imprimir directamente el gráfico en cuestión.
- Una segunda forma de imprimir los gráficos es acceder en la misma ventana al menú según
  `Edit\Copy_to_Metafile`
  o según
  `Edit\Copy_to_Bitmap`
  lo que permite copiar la figura al portapapeles para luego "pegarla" en cualquier aplicación de Windows, por ejemplo, un procesador de texto. El formato *Metafile* es casi siempre preferible, porque es vectorial, es decir, almacena cada elemento de la figura mediante puntos que luego la aplicación que los use se encargará de unir con una precisión y definición arbitraria. Sin embargo, el formato *Bitmap* almacena los *pixels* de la figura, lo que, a la hora de imprimir puede originar el "efecto escalera" en el que las líneas aparecen quebradas.
- Una tercera opción, quizás la más adecuada, es utilizar el comando `print` de Matlab. Este comando almacena la figura actual en un archivo con un formato predeterminado (por defecto la almacena en formato *PostScript*, formato universal que entienden algunas impresoras y gran cantidad de aplicaciones). Entre otros, la función `print` es capaz de generar archivos gráficos en formatos `PCX`, `GIF`, `EPS` (*Encapsulated PostScript*), así como en diversas versiones del formato *PostScript*. Como ejemplo, para imprimir el gráfico de la figura 1 en el archivo `prueba.ps` (que podría enviarse directamente a cualquier impresora *PostScript*)

  ```
  print prueba
  ```

  y para imprimirlo en formato `EPS` (archivo `prueba.eps`)

  ```
  print -deps prueba
  ```

1.3.5. Una sesión en Matlab

El *workspace*: Cuando iniciamos una sesión en Matlab, éste almacena todos los objetos que definamos (escalares, vectores, matrices, polinomios, etc.) en lo que llamamos *espacio de trabajo* o *workspace*. En ocasiones es necesario comprobar qué objetos se hallan almacenados en el *workspace*. Esto se hace mediante la instrucción `who`

```
>> who

Your variables are:

M          n          p          y
```

Otra versión más completa de `who` es `whos`

```
>> whos
  Name        Size       Elements     Bytes    Density   Complex

     M       1 by 1              1         8       Full      No
     n       1 by 20            20       160       Full      No
     p       1 by 20            20       160       Full      No
     y       1 by 1000        1000      8000       Full      No

Grand total is 1041 elements using 8328 bytes
```

que nos devuelve una información más detallada de cada objeto.

También es posible borrar variables del *workspace*

```
>> clear n p
```

lo que nos permite borrar las variables `n` y `p` del espacio de trabajo.

Si lo que queremos es borrar todas las variables, teclearemos

```
>> clear
```

Consulta del tamaño de las variables: En ocasiones, especialmente al desarrollar funciones, suele ser muy útil el poder conocer el tamaño de un elemento determinado. Matlab dispone de dos funciones muy utilizadas que son `size` y `length`. La primera nos devuelve un vector de dos elementos con las dimensiones m y n de la matriz o vector que se le pase como parámetro. La función `length` devuelve $\max\{m, n\}$, es decir, nos indica lo "largo" que es elemento. Con un ejemplo, vemos cómo funcionan.

```
>> y = randn(2,3)

y =

    0.2091    1.5160   -0.3866
    0.4482   -0.5753   -0.3325

>> length(y)

ans =

     3

>> size(y)

ans =
```

```
     2     3

>> length(y')

ans =

     3
```

Salvar y recuperar variables: En ocasiones es necesario salvar en disco todo el *workspace* de la sesión en curso para reanudarla en otro momento. Esto es posible hacerlo simplemente introduciendo el comando `save`, lo cual creará un archivo binario que por defecto se llamará `matlab.mat`. Otras posibilidades son

```
save nom_fich
save nom_fich X
save nom_fich X Y Z
```

La primera salvará todo el *workspace* en el archivo `nom_fich.mat`, mientras que las otras sólo salvarán el objeto X o los objetos X, Y, Z, respectivamente en dicho fichero.

Para recuperar los objetos almacenados en `nom_fich` basta ejecutar

```
load nom_fich
```

Una posibilidad interesante es la que permite almacenar los objetos en archivos de tipo *ascii* (texto), lo que nos permite editarlos posteriormente con cualquier editor o importarlos desde cualquier otra aplicación. Esto es posible añadiendo la opción `-ascii`:

```
save nom_fich X Y Z -ascii
```

Análogamente, `load nom_fich` almacena en la variable del mismo nombre que el del fichero (sin la extensión) los datos introducidos en código *ascii*. No es posible almacenar en un fichero de texto más de un objeto (por ejemplo un vector seguido de una matriz, o dos matrices). Es importante, además, que los datos en el fichero estén bien ordenados ya que de lo contrario Matlab dará un error o los leerá en el orden incorrecto. Por ejemplo, si un archivo `prueba.txt` contiene

```
1 3 4 5
3 4 5
2 2 2 2
```

al hacer `load prueba.txt` Matlab dará un error, ya que le estamos diciendo que se trata de una matriz de 3×4 en la que la segunda fila tiene sólo tres elementos.

Diario de una sesión: Otra posibilidad bastante útil es la de salvar toda o parte de la sesión en un fichero de tipo texto. Mediante la instrucción `diary` se le indica a Matlab que almacene en un fichero

todo lo que aparezca en la ventana de comandos a partir de ese momento. La instrucción `diary off` suspende este proceso y `diary on` lo reanuda. El fichero utilizado por defecto es `diary`. aunque puede decírsele a Matlab que utilice otro si tecleamos `diary nom_fich`. A continuación se muestra el contenido del fichero `diary` en una sesión típica:

```
b = fir1(17,0.1)

b =

  Columns 1 through 7

    0.0018   0.0043   0.0114   0.0250   0.0453   0.0704   0.0960

  Columns 8 through 14

    0.1170   0.1289   0.1289   0.1170   0.0960   0.0704   0.0453

  Columns 15 through 18

    0.0250   0.0114   0.0043   0.0018

plot(abs(fft([b zeros(1,100)])))

plot(log(abs(fft([b zeros(1,100)]))))

t = linspace(0,1,1000);
y = sin(10*t);
ruido = 0.1*randn(1,1000);
yruido = y + ruido;
plot(yruido)

yfilt = filter(b,1,yruido);
plot(yfilt);

plot(y); hold on; plot(yfilt,'r'); hold off;

plot(yruido); hold on; plot(y,'g'); plot(yfilt,'r'); hold off;

diary off
```

Como puede verse, se almacenan también las salidas numéricas o textuales (evidentemente, las gráficas no son almacenadas) que devuelven los comandos introducidos.

La ayuda de Matlab: En el transcurso de una sesión se emplean multitud de comandos. Muy a menudo sucede que olvidamos o desconocemos lo que hace exactamente una función determinada, qué parámetros lleva, en qué orden se encuentran, qué parámetros devuelve, etc. En la mayoría de los casos, resulta muy útil la instrucción `help`. La forma de llamarla es muy simple: basta escribirla, seguida de un tópico. El tópico puede ser un nombre de una función o una *toolbox*. Veamos un ejemplo,

```
>> help linspace
```

```
 LINSPACE Linearly spaced vector.
    LINSPACE(x1, x2) generates a row vector of 100 linearly
    equally spaced points between x1 and x2.

    LINSPACE(x1, x2, N) generates N points between x1 and x2.

    See also LOGSPACE, :.
```

La ayuda, además de especificar claramente cómo se llama a la función y cuáles son sus parámetros, nos remite habitualmente a otras funciones relacionadas (en el ejemplo se nos sugiere ver LOGSPACE), en las que podemos encontrar información que nos puede ser útil para lo que buscamos.

Si tecleamos help, sin más, nos aparece toda una lista de tópicos (comandos de propósito general, operadores, *toolboxes*, etc.) acerca de los cuales podemos pedir ayuda tecleando help topico, lo que nos devolverá a su vez toda una lista de funciones asociadas al tópico en cuestion.

Scripts de demostración (*demoscripts*): en la mayoría de las *toolboxes* (algunas de las cuales llegan tener cientos de funciones especializadas) y en algunos directorios del Matlab, se encuentran abundantes demostraciones –generalmente la resolución de algún problema real o típico en la materia–, que, junto con el manejo de help, nos proporcionan, no sólo un educativo "video" sobre la materia que trate la *toolbox* sino las posibilidades que ésta ofrece así como la forma de trabajar con ellas.

Tecleando simplemente demo accedemos a las demostraciones propias de Matlab mediante menús de botones. Para ver las demostraciones de cada toolbox, podemos introducirnos en el directorio correspondiente. Por ejemplo, tecleando lo siguiente desde la línea de comandos de Matlab:

```
cd c:\matlab\toolbox\control
dir
```

Entre las múltiples funciones vemos algunas, como kalmdemo, que son demostraciones que pueden ejecutarse. Normalmente, suelen correr con el echo activado, es decir, muestran por pantalla, paso por paso, todos los comandos que van ejecutando.

1.4. Ejercicios para el manejo de Matlab

A continuación, y con el objetivo de que el lector practique las opciones de Matlab presentadas, se proponen tres ejercicios.

Ejercicio 1. Diseño de un filtro de mediana.: Considérese la siguiente señal o secuencia de datos:

$$x_k = \{x_0, x_1, \cdots, x_n\} \tag{1.11}$$

Un filtro de mediana[2] de orden p toma una secuencia $\{x_k\}$ dando lugar a otra secuencia $\{y_k\}$ tal que

$$y_k \longleftarrow \text{mediana}\{x_k, x_{k-1}, \cdots, x_{k-p+1}\}, \qquad k > p-1 \quad (1.15)$$
$$y_k \longleftarrow \text{mediana}\{x_k, x_{k-1}, \cdots, x_0\}, \qquad \text{en otro caso} \quad (1.16)$$

Se pide:

1. Desarrollar una función en Matlab que, dado el orden del filtro, p, y una secuencia $\{x_k\}$, devuelva una secuencia $\{y_k\}$ resultado de aplicar el filtro de mediana antes definido.
2. Realizar un *script* en el que se aplique el filtro de mediana a los tipos de señales que se enumeran a continuación, mostrando gráficamente los resultados.
   - ➢ Señal de tipo senoidal, $x_k = \sin(\omega k + \phi)$.
   - ➢ Señal aleatoria de ruido normal de media 0 y varianza σ^2, $x_k \leftarrow N(0, \sigma^2)$.
   - ➢ Señal suma de las dos anteriores.
   - ➢ Señal con *outliers*. Los *outliers* son un tipo de ruido de valores aleatorios que se producen de forma esporádica y aislada en puntos también aleatorios a lo largo de la señal; suele llamársele también *ruido impulsional*. Para generar una señal con outliers que pueda servir de prueba pueden seguirse los siguientes pasos: 1) Generar una señal x_k de N elementos (p.e. una senoidal), 2) Elegir p posiciones aleatorias $k_1, k_2, \cdots, k_p$ en las que se van a producir los outliers, y 3) Sustituir el valor de la señal por otro aleatorio en las posiciones anteriores $x_{k_i} \leftarrow N(0, \sigma^2) \qquad i = 1, \cdots, p$

Ejercicio 2. Filtro de media móvil (FIR).: Un filtro de *media móvil* de orden $p-1$ (también conocido como filtro FIR[3]) toma una secuencia $\{x_k\}$ dando lugar a otra secuencia $\{y_k\}$ de manera que

[2]Nota: La mediana de un conjunto de elementos es aquél que tiene el mismo numero de valores mayores y menores que él. Si el número de elementos fuese par, puede tomarse cualquiera de los dos centrales. Por ejemplo, supónganse que $a < b < c < d$. Entonces,

$$\text{mediana}\{a, b, c\} = b \quad (1.12)$$
$$\text{mediana}\{a, b, c, d\} = b \quad (1.13)$$
$$\text{mediana}\{a, b, c, d\} = c \quad (1.14)$$

[3]Siglas de *Finite Impulse Response*. Un filtro de este tipo, responde ante una señal de tipo impulso $\{\delta_k\}$

$$\{\delta_k\} = \{1, 0, 0, \cdots\} \quad (1.17)$$

dando lugar a una secuencia $\{y_k\}$ que se anula a partir de una cierta muestra. Más aún, dicha secuencia, conocida como *respuesta impulsional*, es precisamente, la formada por sus coeficientes:

$$y_k = \{b_0, b_1, \cdots, b_{p-1}, 0, 0, \cdots\} \quad (1.18)$$

$$y_k \longleftarrow \sum_{i=k-p+1}^{k} b_{k-i}x_i, \qquad k > p-1 \tag{1.19}$$

$$y_k \longleftarrow \sum_{i=0}^{k} b_{k-i}x_i, \qquad \text{en otro caso} \tag{1.20}$$

donde $\{b_0, b_1, \cdots, b_{p-1}\}$ son los *coeficientes del filtro* y representan los pesos que se dan a cada uno de los datos.

Obsérvese que en el caso de que

$$b_0 = b_1 = \cdots = b_{p-1} = \frac{1}{p} \tag{1.21}$$

se tiene

$$y_k \longleftarrow \text{media}\{x_k, x_{k-1}, \cdots, x_{k-p+1}\}, \qquad k > p-1 \tag{1.22}$$

$$y_k \longleftarrow \text{media}\{x_k, x_{k-1}, \cdots, x_0\}, \qquad \text{en otro caso} \tag{1.23}$$

Con el objeto de preservar la energía de la señal al pasar por el filtro, se suele exigir la siguiente condición a los coeficientes

$$S = \sum_{i=0}^{p-1} b_i = 1 \tag{1.24}$$

Se pide:

1. Diseñar una función capaz de implementar sobre una señal $\{x_k\}$ un filtro de media móvil de orden p con coeficientes $\{b_0, \cdots, b_{p-1}\}$. La función deberá comprobar si los pesos cumplen la condición antedicha para, en caso contrario, realizar la oportuna normalización de los mismos (dividirlos por S al objeto de que todos sumen 1).
2. Demostrar su funcionamiento mediante un *script* en el que se aplique dicha función a los cuatro tipos de señales indicados en el ejercicio anterior (senoidal, aleatoria, senoidal con ruido y outliers).

Ejercicio 3.: Considérese un objeto X de Matlab constituido por n puntos en $\mathbb{R}^3$ de coordenadas conocidas (por ejemplo, los puntos de un cubo).

Se pretende diseñar en Matlab una función

$$X_g = f(X, \theta, \phi, \psi) \tag{1.25}$$

que gire dicho objeto según los tres ejes del espacio[4] ángulos θ,ϕ y ψ, devolviendo las coordenadas del objeto girado, X_g.

Finalmente, otra función

$$Z = g(X, \text{tipo}) \tag{1.27}$$

deberá ser capaz de devolver en el objeto Z de Matlab una colección de puntos de $\mathbb{R}^2$ correspondientes a la proyección[5] *isométrica* o *caballera*, según la variable de cadena 'tipo' valga `'iso'` o `'cab'`, respectivamente. La función también los representará en una ventana gráfica de Matlab.

[4] Sugerencia: Un giro en torno al eje z puede considerarse como una transformación lineal

$$T_\theta = \begin{pmatrix} \sin\theta & \cos\theta & 0 \\ -\cos\theta & \sin\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.26}$$

[5] Sugerencia: considérese dicha proyección como una aplicación lineal

$$P : \mathbb{R}^3 \longrightarrow \mathbb{R}^2 \tag{1.28}$$