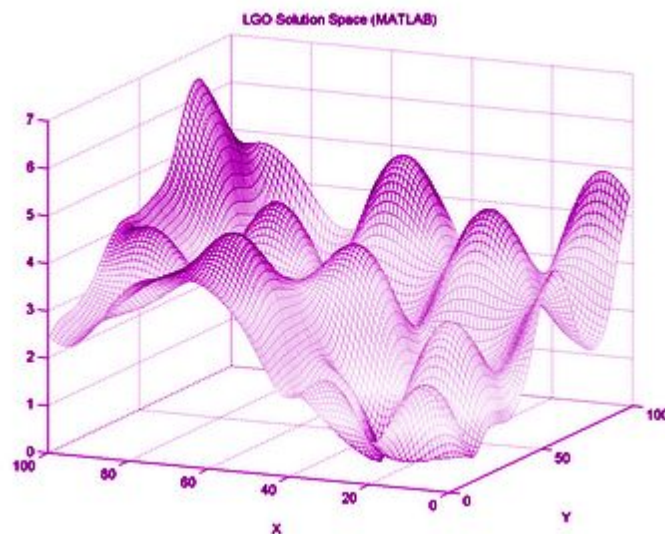


MathWorks®

Señales & Sistemas



Volumen 1 - “Introducción a la plataforma”

Argentina - Marzo 2017

CONTENIDOS

Capítulo 1 - Primeros pasos

[¿Por qué leer este apunte?](#)

[¿Qué es Matlab y por qué es útil para un Ingeniero de Sonido?](#)

[¿Por qué matlab, acaso no hay otros lenguajes?](#)

[Metodología de resolución de problemas](#)

[Diagrama de flujo y pseudocódigo](#)

[Pseudocódigo](#)

[Datos & Variables](#)

[Representación de valores reales](#)

[Errores comunes en el manejo de valores y variables](#)

[Rendimiento y tiempos de ejecución](#)

[Funciones útiles en el uso de variables](#)

[Detectar y corregir errores en el código \(Debugger\)](#)

[Comentario de fin de capítulo](#)

[Comandos, caracteres y funciones usadas en el capítulo](#)

Capítulo 2 - Manejo de array de n-dimensiones

[Comenzando por el principio](#)

[Operaciones con escalares](#)

[Operaciones con arreglos](#)

[Exhibir resultados](#)

[Notación científica](#)

[Guardar el trabajo](#)

[Almacenar variables](#)

[Almacenar comandos](#)

[¿Analogico-continuo o digital-discreto?](#)

[Representación en tiempo continuo y discreto](#)

[Comandos, caracteres y funciones usadas en el capítulo](#)

Capítulo 3 - Gráficos

[Introducción](#)

[Gráficas Bidimensionales](#)

[Gráficas con más de una línea](#)

[Gráficas de arreglos complejos](#)

[Línea, color y estilo](#)

[Subgráficas](#)

[Otros tipos de gráficas](#)

[Comandos, caracteres y funciones usadas en el capítulo](#)

Capítulo 4 - Estructuras de control

[Introducción](#)

[Diagrama de flujo](#)

[Estructuras de selección](#)

[Sentencia IF](#)

[Sentencia SWITCH](#)

[Sentencia FOR](#)

[Sentencia WHILE](#)

[Sentencia BREAK y CONTINUE](#)

[Comentario de fin de capítulo](#)

[Comandos, caracteres y funciones usadas en el capítulo](#)

[Capítulo 5 - Funciones](#)

[Introducción](#)

[Funciones internas](#)

[Ayuda](#)

[Funciones más utilizadas](#)

[Funciones definidas por el usuario](#)

[Comentarios en las funciones](#)

[Funciones sin entrada o salida](#)

[Funciones primarias y secundarias](#)

[Funciones específicas - Audio](#)

[Audiorecorder](#)

[Comandos, caracteres y funciones usadas en el capítulo](#)

[Comentarios finales](#)

[Toolbox de interés](#)

[DSP System Toolbox](#)

[Referencias](#)

[Páginas Web de interés](#)

Capítulo 1 - Primeros pasos

¿Por qué leer este apunte?

Porque en principio el mismo fue pensado y diseñado para orientar al estudiante de la asignatura señales y sistemas con el fin de relacionar los conocimientos adquiridos en la teoría con el resto de las asignaturas de la carrera Ingeniería de Sonido. Lejos de ser un material sumamente teórico, este apunte se apoya en la práctica y su uso para aprender y entender el entorno de programación, agregando un grado de sistematización a la resolución de problemas. Lejos está la intención de reemplazar bibliografía especializada del tema, este apunte aporta una visión más focalizada a la rama de la ingeniería que nos concierne, por este motivo se cuenta con referencias que permiten al lector indagar en detalles y ampliar su capacidades. Entendemos, y en base a nuestra experiencia en la plataforma, que es más útil HACER que LEER (en este campo) este apunte se enfoca en conceptos fundamentales dejando los detalles a bibliografías específicas. **Por último los autores recomendamos usar el formato digital de este material para hacer uso de los links disponibles en el mismo (Se respetan todos los derechos reservados de los autores utilizados en este apunte, todo el material acá expuesto tiene fines académicos y educativos, dejando bajo responsabilidad de los lectores el mal uso de los mismo) y el código presentado como ejemplo. Además agradecemos de antemano toda la ayuda que pueda surgir para mejorar este apunte, así como también la corrección del mismo.**

¿Qué es Matlab y por qué es útil para un Ingeniero de Sonido?

Matlab¹ es una herramienta de cálculo que permite resolver problemas de matemática (Holly M, 1990). Su principal característica, la cual lo diferencia de otras plataformas que cumplen el mismo objetivo (Octave², Maple³, Mathematica⁴, MathCad⁵, Scilab⁶, entre otros), es su capacidad superior en los cálculos que involucran matrices. El nombre mismo Matlab es una abreviatura de Matrix Laboratory. Una de las capacidades más atractivas es la de realizar una amplia variedad de gráficos en dos y tres dimensiones. El lenguaje de programación utilizado es propio de esta plataforma y si bien posee una estructura que resulta familiar

¹ Pagina oficial: <http://www.mathworks.com/>

² Pagina oficial: <https://www.gnu.org/software/octave/>

³ Pagina oficial: <http://www.maplesoft.com/>

⁴ Pagina oficial: <http://www.wolfram.com/mathematica/>

⁵ Pagina oficial: <http://es.ptc.com/product/mathcad>

⁶ Pagina oficial: <http://www.scilab.org/>

para aquellos que tienen conocimiento de programación, las instrucciones y la escritura es diferente a lenguajes de alto nivel.

Este material no tiene intención de suplantar la vasta bibliografía disponible en internet, sino más bien acoplar todo el material disponible en la materia sobre esta plataforma y enfocarlo al diseño de soluciones para la disciplinas relacionadas a señales, en particular las que competen a un ingeniero de sonido. El material se encuentra disponible por volúmenes (comenzando por la introducción, hasta conceptos más avanzados) y dentro del mismo subdividido en capítulos. ¿Por qué un Ingeniero de Sonido, tiene que poseer conocimiento en esta plataforma? la respuesta a esta pregunta no es tan sencilla y es objetivo de este material ayudarlos a responder una vez leído por completo, realizando las prácticas y ejercicios propuesto por el mismo, esto no quiere decir que es fundamental conocerla pero es necesario hoy en día tener conocimiento de algún lenguaje de programación para poder expresar sus ideas en códigos. Seguramente usted a lo largo de la lectura encuentre otras utilidades para esta plataforma. De antemano, es posible relacionar el medio analogico de las ondas acústicas que viajan por un medio isotrópico, con funciones matemáticas gracias a las ecuaciones que reinan en la física. Es aquí donde Matlab entra en juego, esta herramienta permite resolver un sinfín de cálculos de una manera más rápida y eficiente, aplicando métodos numéricos. Además como se muestra en este material el trabajo no solo se limita en el dominio temporal o frecuencial, es posible analizar las señales del medio analogico al digital y discreto de una manera detallada y precisa, permitiendo dar soluciones a problemas propios y para los cuales aún no existen herramientas disponibles. Además que Matlab se ha convertido en una herramienta estándar para ingenieros y científicos (Holly M, 1990).

¿Por qué matlab, acaso no hay otros lenguajes?

Claro que hay una gran cantidad de lenguajes disponibles, algunos pagos (como es el caso de Matlab⁷) y otros de uso libre o open source (como es el caso de Python⁸). Por lo general, los programas de alto nivel no ofrecen acceso fácil a la graficación y/o a la generación de entornos gráficos para interactuar, que son alguno de las aplicaciones en la que se destaca Matlab. El área principal de interferencia entre Matlab y los programas de alto nivel es el “procesamiento de números”: programas que requieren cálculos repetitivos o el procesamiento de grandes cantidades de datos. Tanto Matlab como los programas de alto nivel son buenos en el procesamiento de números (Holly M, 1990). Por lo general, es más fácil escribir un programa que “procese números” en Matlab, pero usualmente se ejecutará más rápido en C++ o Python + complementos. La única excepción a esta regla son los cálculos que involucran matrices: puesto que Matlab es óptimo para matrices, si un problema se puede formular con una solución matricial, Matlab lo ejecuta sustancialmente más rápido que un programa similar en un lenguaje de alto nivel, esto esta demostrado empíricamente en el siguiente link (<http://wiki.scipy.org/PerformancePython>).

Además Matlab cuenta con una ventaja más, una gran cantidad de material de ayuda, no solo el dispuesto por la firma, sino además lo expuestos en foros, páginas de universidades,

⁷ Si bien Matlab es pago se pueden conseguir licencias gratuitas de prueba o licencias de estudiante.

⁸ Pagina oficial: <https://www.python.org/>

blog, entre otros, esto se debe a la cantidad de años en el mercado. Como si fuera poco, Matlab cuenta con un sistema de ayuda muy completo, que además de contar con ayuda en el uso de funciones particulares, posee ejemplo, explicación y en algunos casos los fundamentos teóricos de los algoritmos.

Por este motivo y otros que se irán desprendiendo a lo largo de este apunte es que actualmente Matlab es la plataforma más acorde y cómoda para el procesamiento de señales (por ahora...).

A continuación se presentan algunos ejemplos significativos del uso de matlab en ingeniería (Holly M, 1990):

- **Ingeniería de automatización:** Matlab se utiliza mucho en ingeniería eléctrica para aplicaciones de procesamiento de señales. Por ejemplo, en la figura 1.1 se presentan varias imágenes creadas durante un programa de investigación en la University of Utah para simular algoritmos de detección de colisiones que usan las moscas domésticas (y adaptados en el laboratorio a sensores de silicio). La investigación dio como resultado el diseño y fabricación de un chip de computadora que detecta colisiones inminentes. Esto tiene una aplicación potencial en el diseño de robots autónomos que usen la visión para navegar y en particular en aplicaciones para la seguridad en automóviles.

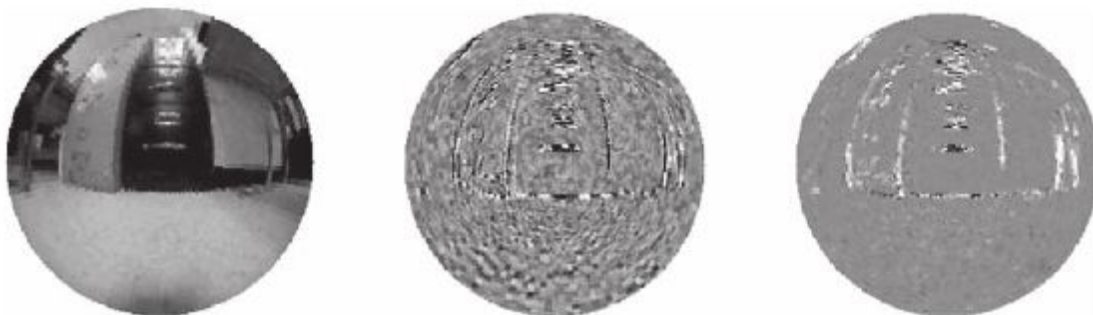


Figura 1.1 - Procesamiento de imágenes con el uso de una cámara con objetivo de ojo de pescado para simular el sistema visual del cerebro de una mosca doméstica (Holly M, 1990).

- **Ingeniería biomédica:** Por lo general, las imágenes médicas se guardan como archivos dicom (el estándar Digital Imaging and Communications in Medicine: imágenes digitales y comunicaciones en medicina). Los archivos dicom utilizan la extensión de archivo .dcm. La compañía MathWorks ofrece una caja de herramientas adicional (toolbox), para imágenes que puede leer esos archivos, lo que hace que sus datos estén disponibles para procesamiento en Matlab. El toolbox también incluye un amplio rango de funciones de las que muchas son especialmente apropiadas para las imágenes médicas.
- **Procesamiento de señales:** En el campo del procesamiento de señales el sistema no solo se limita a imágenes, es posible manipular señales del medio físico, como ser señales: acústicas, térmicas, velocidad, aceleración, entre otros. Por medio de una correcta conversión analógica-digital. Otro campo de trabajo son los que

contemplan las señales del habla para su procesamiento y reconocimiento (McLoughlin I, 2009).

Muchas más son las posibilidades y más aún gracias a las librerías de expansión con que cuenta Matlab denominadas toolbox.

Metodología de resolución de problemas

En las disciplinas de ingeniería, ciencias y programación de computadoras, es importante tener un enfoque consistente para resolver los problemas técnicos. Cuando nos piden que hagamos un programa debemos seguir una cierta cantidad de pasos para asegurarnos de que tendremos éxito en la tarea. La acción irreflexiva (me piden algo, me siento frente a la computadora y escribo rápidamente y sin pensarlo lo que me parece que es la solución) no constituye una actitud profesional (e ingenieril) de resolución de problemas. Toda construcción tiene que seguir una metodología, un protocolo de desarrollo, dado. Hay muchas técnicas para resolver esta cuestión, a continuación se menciona algunos puntos importantes para tener en cuenta, en función a lo presentado en la siguiente referencia (Holly M, 1990).

- **Plantear el problema:** En esta etapa con frecuencia es útil hacer un dibujo o diagrama. Si no se tiene una comprensión clara del problema, es improbable que pueda resolverlo. Entender profundamente cuál es el problema que se trata de resolver, incluyendo el contexto en el cual se usará. Una vez analizado el problema, asentar el análisis por escrito
- **Describir el problema en analogía a una caja negra:** Describir los valores de entrada (conocidos) y las salidas (incógnitas) que se requieran, incluyendo las unidades conforme describe los valores de entrada y salida. El manejo descuidado de las unidades con frecuencia lleva a respuestas incorrectas. Identificar las constantes que tal vez requiera en el cálculo.
- **Desarrollar un algoritmo para resolver el problema:** En aplicaciones de cómputo, es frecuente que esto se logre con una prueba de escritorio. Para ello se necesita identificar ecuaciones que relacionen los valores conocidos con las incógnitas. Trabajar con una versión simplificada del problema, a mano o con calculadora.
- **Resolver el problema.** En este apunte, esta etapa involucra la creación de una solución con Matlab. Traducir a un lenguaje de programación el diseño que elegimos en el punto anterior. La implementación también se debe documentar, con comentarios dentro y fuera del código, al respecto de qué hace el programa, cómo lo hace y por qué lo hace de esa forma.
- **Probar la solución.** ¿Los resultados tienen sentido? ¿Coinciden con los cálculos de la muestra? ¿La respuesta es la que se pedía en realidad? Las gráficas con frecuencia son formas útiles de verificar que los cálculos son razonables. Diseñar un conjunto de pruebas para probar cada una de sus partes por separado, y también la correcta integración entre ellas. Utilizar el depurador como instrumento para descubrir dónde se producen ciertos errores. Al ejecutar las pruebas, documentar los resultados obtenidos.

- **Mantener el programa.** Realizar los cambios en respuesta a nuevas demandas. Cuando se realicen cambios, es necesario documentar el análisis, la especificación, el diseño, la implementación y las pruebas que surjan para llevar estos cambios a cabo.

Una buena estrategia para encarar la solución a problemas complejos son los diagramas de flujo y el pseudocódigo.

Diagrama de flujo y pseudocódigo

Pseudocódigo

Los diagramas de flujo son un enfoque gráfico para crear un plan de codificación y el pseudocódigo es una descripción verbal de este plan. Para programas simples, el pseudocódigo es el mejor enfoque para resolver problemas de manera más ágil. Para esto es impredecible poner el enunciado de forma tal que cumple con los siguientes pasos (Holly M, 1990):

- Desglosar el enunciado en partes más pequeñas, de manera que el problema quede descrito por pasos.
- Convertir los pasos en comentarios (En Matlab se utiliza el carácter de porcentaje “%”, para ingresar comentarios).
- Insertar el código correspondiente debajo de cada comentario.

Por ejemplo a continuación se presenta el procedimiento para calcular la velocidad del sonido en el aire. Para lo cual se parte de la ecuación de Boyle-Mariotte (Möser M., Barros J.L, 2009):

$$c = \sqrt{k \frac{R}{M_{mol}} T_0} \quad (1.1)$$

Esta depende solo del medio y de la temperatura absoluta, pero no de la presión o densidad estáticas. Si se ingresan los valores típicos para el aire:

- $M_{mol} = 28,8 \times 10^{-3} \text{ kg/mol}$
- $T_0 = 288 \text{ K (15°C)}$
- $k = 1,4$
- $R = 8,314 \text{ J/mol K}$ como $J = N \text{ m}$, $N = \text{kg m/s}^2$

A partir de estos valores se obtiene que $c \approx 341 \text{ m/s}$. A continuación se expresa en pseudocódigo lo expresado anteriormente, con el objetivo de introducir a la plataforma de estudio.

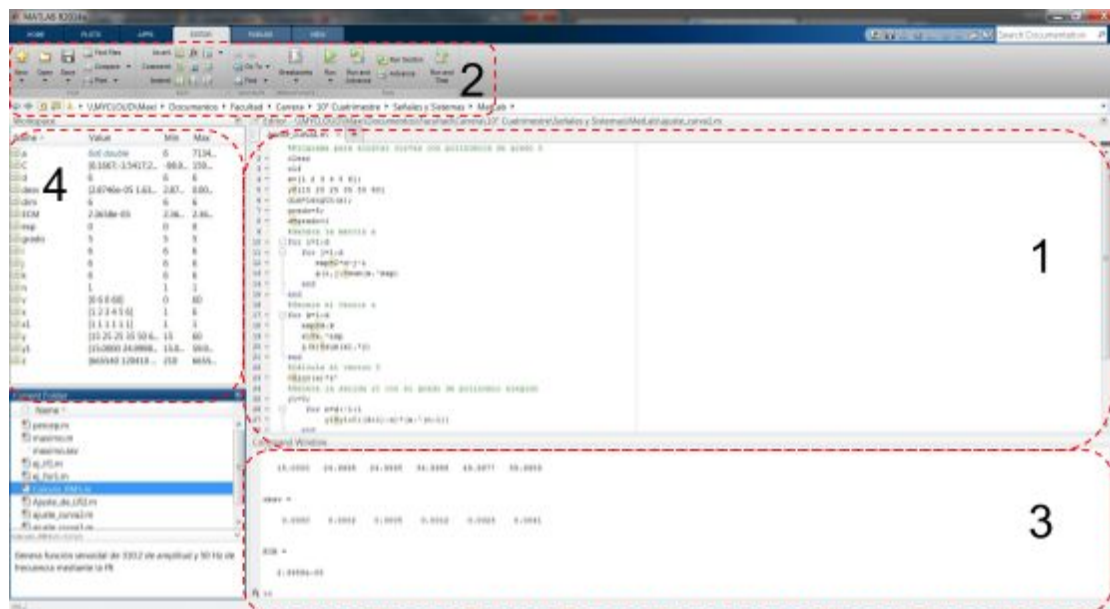


Figura 1.2 - Ventana de apertura de Matlab R2014a⁹. El ambiente consta de algunas ventanas, cuatro de las cuales se abren en la vista por defecto. Otras se abren conforme se necesiten durante una sesión y son configurables desde las opciones de Matlab.

Para lo cual damos inicio al ambiente de Matlab¹⁰ (se recomienda leer el capítulo 2 de la referencia (Holly M, 1990), en especial los incisos 2.1 y 2.2). En la figura 1.2, se observa la venta de apertura de Matlab.

En la figura 1.2, se observan 4 ventanas principales que se describen brevemente, para mayor detalle leer el inciso 2.2 de (Holly M, 1990):

- **Ventana 1 (Editor):** esta ventana contiene el código de las funciones principales de Matlab. La misma es utilizada en la creación de archivos-m¹¹ (el detalle de estos archivos se verá más adelante). Como programador, puede crear y guardar código en archivos llamados archivos-m. Un archivo-m es un archivo de texto ASCII similar a los archivos de código fuente de C. Se puede crear y editar con el editor/debugger (depurador) de archivo-m. Dicha ventana puede abrirse escribiendo `edit` en el Command Windows. Dado que su estructura al final al cabo es un texto ASCII es posible editar un archivo-m en un editor de texto diferente, asegúrese de que los archivos que guarde sean archivos ASCII. **Una consideración muy importante es cuando se guarda un archivo-m, éste se almacena en el directorio actual, además será necesario nombrar al archivo con un nombre**

⁹ Suele existir dos lanzamientos por año de la empresa Mathworks, es por este motivo que suele aparecer con frecuencia versiones a o b, en el siguiente link se observa las distintas versiones y su características (<http://www.mathworks.com/help/matlab/release-notes.html>).

¹⁰ En este apunte se trabaja con las versiones 2010 en adelante, en particular la 2014.

¹¹ Es posible definir funciones propias, aquellas que son usadas más comúnmente en su programación. Las funciones definidas por el usuario se almacenan como archivos-m y Matlab puede acceder a ellas si están almacenadas en el directorio actual.

válido, esto es, un nombre que comience con una letra y contenga solo letras, números y el guión bajo (_). No se permiten los espacios. Usar archivos-m script permite trabajar en un proyecto y guardar la lista de comandos para usarlo en el futuro (se recomienda el uso de comentarios como recordatorios). El operador comentario en Matlab es el signo de porcentaje (%), como en Matlab no ejecutará código alguno en una línea comentada¹².

```
% Esto es un comentario13
```

También puede agregar comentarios después de un comando, pero en la misma línea:

```
t=0:1:100; %La variable t (tiempo) se define de 0 a 100 a  
pasos de 1.
```

Es importante aclarar que agregar comentarios a nuestro código no es sencillamente clarificar que hace cada línea, es mucho más que eso y el poco o exceso del mismo puede perjudicar la interpretación del mismo. Por este motivo se recomienda leer el siguiente link (<http://www.learncpp.com/cpp-tutorial/12-comments/>).

- **Ventana 2 (Barra de herramientas):** difiere de versión en versión, pero lo fundamental de este elemento es que es aquí donde se depositan todas las herramientas de Matlab, a lo largo de este apunte se irá nombrando las más importantes. Un buen ejercicio es explorarlas por cuenta propia, no solo porque es sumamente sencillo, sino que además Matlab cuenta con muy buena documentación de ayuda que permite avanzar de forma autónoma (más adelante se hará referencia al uso de las herramientas de ayuda y asistencia que dispone esta plataforma).
- **Ventana 3 (Command Window):** la ventana de comandos ofrece un ambiente similar a una memoria de trabajo auxiliar (scratch pad). El empleo de la ventana de comandos permite guardar los valores calculados y no los comandos para generarlos. Para guardar la secuencia de comandos, necesitará emplear la ventana de edición (editor) para crear un archivo-m (m-file).
- **Ventana 4 (Workspace):** la ventana del área de trabajo mantiene informado de las variables que se define conforme se ejecute comandos en la ventana de comandos. Por ejemplo la figura 1.3, muestra workspace típico de una aplicación. En la imagen se observan todas las variables utilizadas por la aplicación ejecutada, además de la información detallada de las mismas. En este ejemplo solo se desplegaron algunos detalles, pero es posible tener la siguiente información de las variables: size (dimensión del arreglo) y bytes (tamaño lógico), name (nombre), value (valor), class (clase), entre otras. En próximos capítulos se menciona la importancia de una correcta gestión de variables y las buenas prácticas respecto a la administración de las mismas.

¹² Para aquellos que tengan problemas para encontrar caracteres ASCII, se recomienda utilizar la combinación de la tecla Alt + (algun NUMERO) del siguiente link (<http://www.elcodigoascii.com.ar/>)

¹³ Todo lo que se encuentre en estos recuadros es código del ambiente Matlab. En el formato digital dicho código está habilitado para copiarlo y probarlo.

Name	Value	Min	Max	Class
a	6x6 double	6	7134...	double
C	[0.1667;-3.541...	-98.9...	159...	double
d	6	6	6	double
desv	[2.8746e-05 1...	2.87...	0.00...	double
dim	6	6	6	double
ECM	2.3658e-05	2.36...	2.36...	double
exp	0	0	0	double
grado	5	5	5	double
i	6	6	6	double
j	6	6	6	double
k	6	6	6	double
n	1	1	1	double
v	[0 6 0 60]	0	60	double
x	[1 2 3 4 5 6]	1	6	double
x1	[1 1 1 1 1 1]	1	1	double
y	[15 25 25 35 5...	15	60	double
y1	[15.0000 24.99...	15.0...	59.9...	double
z	[665540 12041...	210	6655...	double

Figura 1.3 - Ventana de workspace típico de una aplicación. La información que brinda esta ventana es de suma importancia para gestionar, controlar y administrar todas las variables que forman parte de nuestro código.

Retomando al ejercicio del cálculo de la velocidad del sonido a partir de la ecuación (1.1), planteemos el pseudocódigo para resolver dicho problema. Para lo cual se utilizará la ventana editor para generar un archivo-m. Como se mencionó anteriormente el primer paso para desarrollar el pseudocódigo es enunciar el problema en partes más pequeñas, las cuales son agregadas como comentarios (%) al código para usarlas de guía, a continuación se observa un ejemplo de esto:

```
%Declaración e inicialización de variables que intervienen

%Ingresar un intervalo de temperaturas (en grados centígrados)

%Convertir temperatura de grados centígrados [C] a grados Kelvin [K]

%Combinar valores de temperatura y velocidad en una matriz

%Crear un título de tabla

%Crear encabezados de columna

%Desplegar la tabla
```

Con los comentario en su lugar y formado una suerte de guía paso a paso para dar solución al problema, se comienza a insertar el código Matlab apropiado a cada ítem, como se observa a continuación:

```

%Declaración e inicialización de variables que intervienen
M_mol=28.8*10^(-3);
k=1.4;
R=8.314;

%Ingresar un intervalo de temperaturas (en grados centígrados)
t=0:1:25;

%Convertir temperatura de grados centígrados [C] a grados Kelvin [K]
T=t+273.5;

%Obtener los valores de velocidad correspondientes a la temperatura
c=sqrt(k*R*T/M_mol);

%Combinar valores de temperatura y velocidad en una matriz
tabla=[t;c];

%Crear un título de tabla
disp('Tabla de velocidad del sonido en el aire en función de la temperatura');

%Crear encabezados de columna
disp('Temperatura [C°]    Velocidad [m/s] ');

%Desplegar la tabla
fprintf('%8.0f %8.2f \n', tabla);

```

Muchas de las funciones que se observan en este código no fueron explicadas anteriormente. En la tabla final del capítulo se las mencionan, además si y sólo si es necesario se las explicara en detalle en futuras secciones.

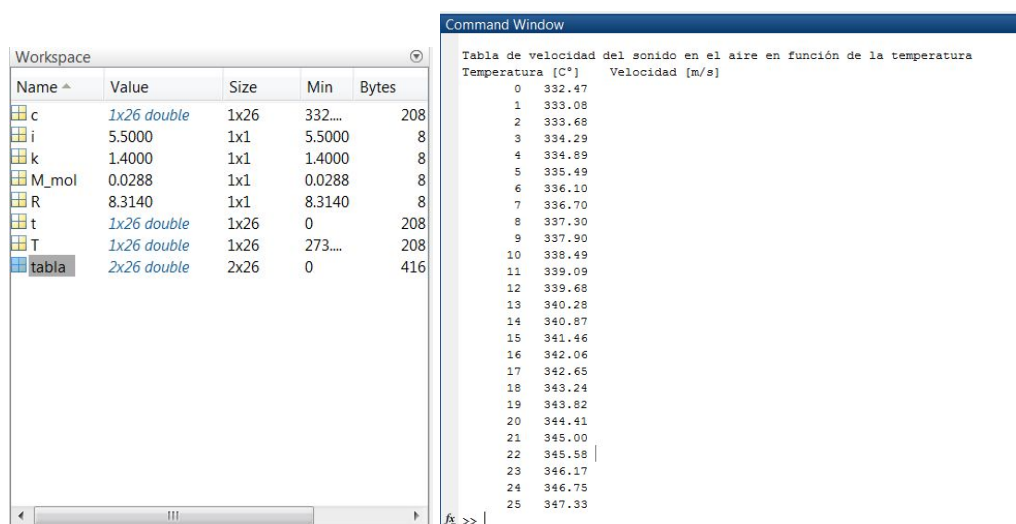


Figura 1.4 - Workspace con las variables utilizadas. Además del command window con los resultados arrojados por el script.

Antes de dar por finalizado este ejercicio es importante recalcar la importancia del uso de comentarios para organizar el código y el uso de punto y coma (;) al final de cada línea no

comentada, no solo para acostumbrar a un tipo de escritura limpia¹⁴, sino que además evita que dicha línea sea reproducida en el command window.

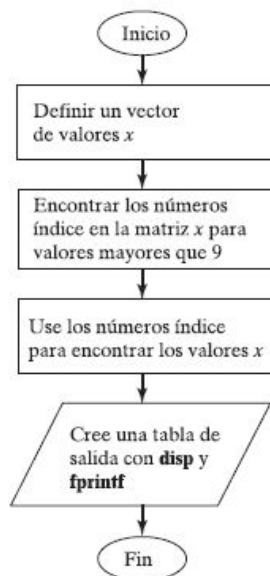


Figura 1.5 - Diagrama de flujo que ilustra el comando `find`¹⁵ (Holly M, 1990).

Retomando la ideas de diagrama de flujo y pseudocódigo, la explicación de diagrama de flujo se extenderá en los próximos capítulos en particular donde se definan las estructuras de control y funciones lógicas, a continuación se presenta un figura el diagrama de flujo, con el objetivo de mencionar sobre su estructura, la figura 1.5 describe un diagrama de flujo típico de una función particular. El uso de cada uno de ello es muy específico del problema a resolver y de la habilidad del programador, los mismo pueden ser combinados o utilizados por separado, la práctica y el tiempo dirá.

Datos & Variables

Como se mencionó en un principio Matlab es un programa preparado para trabajar con vectores y matrices. Como caso particular también trabaja con variables escalares (matrices de dimensión 1). Por defecto, si no se declara lo contrario, **Matlab trabaja siempre en doble precisión, es decir guardando cada dato en 8 bytes (una precisión exagerada para algunos casos)** (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

Representación de valores reales

Para representar números enteros muy grandes o números fraccionarios muy pequeños es necesarios utilizar muchos bits (1 byte = 8 bits). Para comprender la magnitud de este valor es necesario conocer sobre la notación científica. El primer caso son los **números representados en punto fijo (o coma fija)** son aquellos números en los que la coma

¹⁴ Hay muchos lenguajes de alto nivel que consideran el (;) como elemento fundamental del código. Sin el mismo, la línea se considerar con error.

¹⁵ El comando `find` regresa sólo una respuesta: un vector de los números de elemento solicitados.

fraccionaria tiene una posición fija. Por ejemplo, si tenemos números de 8 bits podemos dedicar 5 bits a la parte entera y 3 a la fraccionaria. Esto supone una severa limitación, en primer lugar no podemos usar más de 5 bits para la parte entera aunque la fraccionaria sea cero y por otro lado, no podemos usar más de 3 bits para la parte fraccionaria aunque la entera sea cero. La alternativa es, **representación en punto flotante (o coma flotante)**, resuelve este inconveniente y está basado en la notación científica, utilizada para representar números muy grandes o muy pequeños. Un número representado en punto flotante se compone de la siguiente manera:

$$|m| \times 10^{\varepsilon} \quad (1.2)$$

El número m se denomina mantisa (representa la magnitud del número), ε es el exponente (indica el desplazamiento de la coma fraccionaria) y por último, la base que en este caso está representado por 10 (base_{10}). Los números decimales en coma flotante se normalizan, desplazando la coma fraccionaria de manera que la parte entera del número siempre valga cero ej:

$$241506800 = 0,2415068 \times 10^9 \quad (1.3)$$

En este caso el signo = +, mantisa = 2415068 y el exponente = 9. No es necesario representar la base del número, ya que está implícita en el formato. Dado que la parte entera de un número normalizado siempre es cero, tampoco es necesario representarla.

Retomando al entorno de Matlab, la base en este caso es 2. El formato utilizado para la representación de números binarios en coma flotante está definido por el estándar 754-1985 ANSI/IEEE¹⁶. El formato establecido por este estándar se observa en la figura 1.6:

Signo	Exponente	Mantisa
1 bit	8 bits	23 bits

Signo	Exponente	Mantisa
1 bit	11 bits	52 bits

Figura 1.6: En primer lugar se observa el formato de simple precisión (32 bits) y luego el de doble precisión (64 bits).

Se recomienda al lector leer la normativa para conocer más sobre este formato de representación de números o de manera más resumida el capítulo 2.6 de la siguiente referencia (Floyd T., 2006).

Todo esto tiene el objetivo de introducir a las buenas prácticas en el manejo de variables en la plataforma de Matlab, lo cual se verá reflejado en un mejor rendimiento del código. Como ya se ha comentado, por defecto Matlab trabaja con variables de punto flotante y doble precisión (double) 64 bits por variable. **Con estas variables pueden resolverse casi todos los problemas prácticos y con frecuencia no es necesario complicarse la vida**

¹⁶ Consulta: <http://twins.ee.nctu.edu.tw/~tjlin/courses/co01/IEEE754.pdf>

declarando variables de tipos distintos, como se hace con cualquier otro lenguaje de programación. Sin embargo, en algunos casos es conveniente declarar variables de otros tipos porque puede ahorrarse mucha memoria y pueden hacerse los cálculos mucho más rápidamente (García de Jalón J., Rodríguez I. J, Vidal J., 2005), aparte Matlab permite crear variables enteras con 1, 2, 4 y 8 bytes (8, 16, 32 y 64 bits). A su vez, estas variables pueden tener signo o no tenerlo. Las variables con signo representan números en intervalos "casi" simétricos respecto al 0; las variables sin signo representan número no negativos, desde el 0 al número máximo, para más información de estas formas de representar números con o sin signo leer el capítulo 2 de la siguiente referencia (Floyd T., 2006).

Los tipos de los enteros con signo son `int8`, `int16`, `int32` e `int64`, y sin signo `uint8`, `uint16`, `uint32` y `uint64`. Para crear una variable entera de un tipo determinado se pueden utilizar el siguiente código (García de Jalón J., Rodríguez I. J, Vidal J., 2005):

```
i=int32(100); %se crea un entero de 32 bits (4 bytes) con valor 100.
j=zeros(100); i=int32(j); %se crea un entero i a partir de j
i=zeros(1000,1000,'int32'); %se crea una matriz 1000x1000 de enteros.
```

Las funciones `intmin('int64')` e `intmax('int64')` permiten por ejemplo saber el valor del entero más pequeño y más grande que puede formarse con variables enteras de 64 bits. Otra función muy útil a la hora de conocer las variables de nuestro código es, `class()`, la cual devuelve el tipo de variable del argumento.

Matlab dispone de dos tipos de variables reales (float) `single` (32 bits) y `double` (64 bits). Las funciones `single(x)` y `double(y)` permiten realizar conversiones entre ambos tipos de variables, respectivamente. Las funciones `realmin()` y `realmax()` permiten saber los números `double` más pequeño y más grande (en valor absoluto) que admite el computador. Para los correspondientes números de simple precisión habría que utilizar `realmin('single')` y `realmax('single')`. La función `isfloat(x)` permite saber si `x` es una variable real, de simple o doble precisión (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

Matlab dispone de tres funciones útiles relacionadas con las operaciones de coma flotante. Estas funciones, que no tienen argumentos, son las siguientes: `eps` devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. Da una idea de la precisión o número de cifras almacenadas. En un PC (en particular), `eps` vale `2.2204e-016`. Por otra parte la función `realmin` devuelve el número más pequeño con que se puede trabajar (`2.2251e-308`) y la función `realmax` devuelve el número más grande con que se puede trabajar (`1.7977e+308`).

Otro tipo de variable de sumo interés es la variable lógica (boolean) que sólo pueden tomar los valores `true` (1) y `false` (0). Las variables lógicas surgen como resultado de los

operadores relacionales (explicados en detalles en los siguientes capítulos). La función `logical(A)` produce una variable lógica, con el mismo número de elementos que A, con valores 1 ó 0 según el correspondiente elementos de A sea distinto de cero o igual a cero. Una de las aplicaciones más importantes de las variables lógicas es para separar o extraer los elementos de una matriz o vector que cumplen cierta condición, y operar luego selectivamente sobre dichos elementos (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

```
A=magic(4); %Matriz especial
    A =     16     2     3    13
         5    11    10     8
         9     7     6    12
         4    14    15     1
j=A>10; %Establecer la condición de selección
    j =     1     0     0     1
         0     1     0     0
         0     0     0     1
         0     1     1     0
isa(j,'logical'); %Pregunto si la variable j es del tipo lógica
    ans = 1
A(j)=-10; %Reemplazo los valores
    A =    -10     2     3   -10
         5   -10    10     8
         9     7     6   -10
         4   -10   -10     1
```

Observar cómo se reemplaza todos los valores mayores a 10. La función `magic()` es utilizada para crear la matriz inicial que tiene características especiales, como la suma de las filas, las columnas o sus diagonales siempre es el mismo número¹⁷. El mecanismo de solución de este ejemplo, permite ahorrar mucha memoria puesto que el cálculo se realiza entre matrices y sin recurrir a una rutina del tipo bucle (por ejemplo, `For`) la cual consume mucho tiempo de cálculo (El detalle de estas rutina se verán más adelante) .

Errores comunes en el manejo de valores y variables

Matlab mantiene una forma especial para los números muy grandes (más grandes que los que es capaz de representar), que son considerados como infinito. Por ejemplo, obsérvese cómo responde el programa al ejecutar el siguiente comando:

```
>> 1.0/0.0
Warning: Divide by zero ans = Inf
```

¹⁷ Para mas información de la funcion `magic()`: <http://www.mathworks.com/help/matlab/ref/magic.html>

De esta manera en Matlab el **infinito se representa como inf ó Inf**. Por otra parte, también existe una representación especial para los resultados que no están definidos como números. Por ejemplo, en el caso de la siguiente línea de comandos:

```
>> 0/0
Warning: Divide by zero ans = NaN
>> inf/inf ans = NaN
```

En ambos casos la respuesta es NaN, que es la abreviatura de Not a Number. Este tipo de respuesta, así como la de Inf, son enormemente importantes en Matlab, pues permiten controlar la fiabilidad de los resultados de los cálculos matriciales. Los NaN se propagan al realizar con ellos cualquier operación aritmética, por ejemplo cualquier número sumado a un NaN da otro NaN. Algo parecido sucede con los Inf.

Rendimiento y tiempos de ejecución

Como se mencionó anteriormente, la idea de aprender una correcta gestión y administración de variables no es solo para tener un código limpio y legible, es por sobre todas las cosas para disminuir el uso de memoria, lo que se ve reflejado directamente en un aumento sustancial del rendimiento. Para lo cual Matlab dispone de funciones que permiten calcular el tiempo empleado en las operaciones matemáticas realizadas. Algunas de estas funciones son las siguientes: `cputime` devuelve el tiempo de CPU (con precisión de centésimas de segundo) desde que el programa arrancó. Llamando antes y después de realizar una operación y restando los valores devueltos, se puede saber el tiempo de CPU empleado en esa operación. Este tiempo sigue corriendo aunque Matlab esté inactivo (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

```
n=1000;
A=rand(n);
b=rand(n,1);
x=zeros(n,1);
tiempoIni=clock;
x=A\b;
tiempo=etime(clock, tiempoIni);
time=cputime;
x=A\b;
time=cputime-time;
tic;
x=A\b;
toc;
```

La función `etime(t2, t1)` devuelve el tiempo transcurrido entre los vectores `t1` y `t2` (el orden es importante), obtenidos como respuesta al comando `clock`. Por otra parte, `tic` y

`toc` imprime el tiempo en segundos. El comando `tic` pone el reloj a cero y `toc` obtiene el tiempo transcurrido..

El `profile`¹⁸ es una función que permite saber cómo se ha empleado el tiempo de la CPU en la ejecución de un determinado programa. El profiler es una herramienta muy útil para determinar los cuellos de botella de un programa, es decir las funciones y las líneas de código que más veces se llaman y que se llevan la mayor parte del tiempo de ejecución (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

```
%Limpiado de command window y workspace
clc;
clear;
%Inicialización del timer
tic;
%Declaración e inicialización de variables que intervienen
M_mol=single(28.8*10^(-3));
k=single(1.4);
R=single(8.314);
%Ingresar un intervalo de temperaturas (en grados centígrados)
t=single(0:1:25);
%Convertir temperatura de grados centígrados [C] a grados Kelvin [K]
T=single(t+273.5);
%Obtener los valores de velocidad correspondientes a la temperatura
c=single(sqrt(k*R*T/M_mol));
%Combinar valores de temperatura y velocidad en una matriz
tabla=[t;c];
%Crear un título de tabla
disp('Tabla de velocidad del sonido en el aire en función de la temperatura');
%Crear encabezados de columna
disp('Temperatura [C°]    Velocidad [m/s]');
%Desplegar la tabla
fprintf('%8.0f %8.2f \n', tabla);
toc
```

¹⁸ Para mas información de la funcion profiler: <http://www.mathworks.com/help/matlab/ref/profile.html>

Command Window		Command Window	
Tabla de velocidad del sonido en el aire en función de la temperatura		Tabla de velocidad del sonido en el aire en función de la temperatura	
Temperatura [C°]	Velocidad [m/s]	Temperatura [C°]	Velocidad [m/s]
0	332.47	0	332.47
1	333.08	1	333.08
2	333.68	2	333.68
3	334.29	3	334.29
4	334.89	4	334.89
5	335.49	5	335.49
6	336.10	6	336.10
7	336.70	7	336.70
8	337.30	8	337.30
9	337.90	9	337.90
10	338.49	10	338.49
11	339.09	11	339.09
12	339.68	12	339.68
13	340.28	13	340.28
14	340.87	14	340.87
15	341.46	15	341.46
16	342.06	16	342.06
17	342.65	17	342.65
18	343.24	18	343.24
19	343.82	19	343.82
20	344.41	20	344.41
21	345.00	21	345.00
22	345.58	22	345.58
23	346.17	23	346.17
24	346.75	24	346.75
25	347.33	25	347.33
Elapsed time is 0.027018 seconds.		Elapsed time is 0.030678 seconds.	

Figura 1.7 - Resultados del código declarando las variables (izquierda) y resultado del código sin declarar variables (derecha)

Funciones útiles en el uso de variables

Como se mencionó anteriormente el nombre de la variable no puede tener cualquier forma y estructura, ahora se agrega además que no puede ser ninguna de las palabras reservadas por Matlab. El comando `iskeyword` hace que Matlab elabore una lista de tales nombres reservados. El comando `clc`, limpia la ventana de comandos y deja una página en blanco. Sin embargo, no borra de la memoria las variables reales que creó. El comando `clear` borra todas la variables guardadas.

Detectar y corregir errores en el código (Debugger)

La técnica que se explicará a continuación es de suma utilidad cuando se trabaja en la ventana Editor, como se mencionó anterior esta ventana crea archivos-m. El primer paso para detectar un error en el código es ordenarlo para entender e identificar los pasos que realiza el algoritmo paso a paso. Para lo cual Matlab cuenta con los comentarios (lo cuales ya fueron presentados anteriormente), los mismo pueden ser agregados de manera más rápida y varias líneas a la vez seleccionando dichas líneas y haciendo click con el botón derecho aparece un menú contextual cuya sentencia Comment permite entre otras cosas comentar con el carácter `%` todas las líneas seleccionadas. Estos comentarios pueden volver a su condición de código ejecutable seleccionandolos y ejecutando Uncomment en el menú contextual, esto nos permite entre otras cosas, habilitar o deshabilitar parte del código. Otra opción muy útil de ese menú contextual es Smart Indent, que organiza el tabulado de los bucles y bifurcaciones de las sentencias seleccionadas (García de Jalón J., Rodríguez I. J, Vidal J., 2005), dando una impresión visual más organizada de nuestro código, un ejemplo de esto procedimiento se observa en la figura 1.8.

```

10
11 % Indent Nested Functions
12 function nested_one
13 -   disp('Main function code')
14     function nested_two
15 -         disp('Nested function code')
16 -     end
17 - end
18
19 % Indent All Functions
20 function all_one
21 -     disp('Main function code')
22     function all_two
23 -         disp('Nested function code')
24 -     end
25 - end
26 |

```

Figura 1.8 - Ejemplo de ordenado correctamente (Imagen extraída: <http://www.mathworks.com/>)

La Figura 1.9 corresponde a la ejecución del Debugger¹⁹, sobre un código un poco más complejo. Dicha ejecución comienza eligiendo el comando Run en el menú Debug (o tecla F5). Los puntos rojos que aparecen en el margen izquierdo son breakpoints (puntos en los que se detiene la ejecución de programa); la flecha verde en el borde izquierdo indica la sentencia en que está detenida la ejecución (antes de ejecutar dicha sentencia); cuando el cursor se coloca sobre una variable aparece una pequeña ventana con los valores numéricos de esa variable, como se observa en la figura 1.10 (García de Jalón J., Rodríguez I. J., Vidal J., 2005).

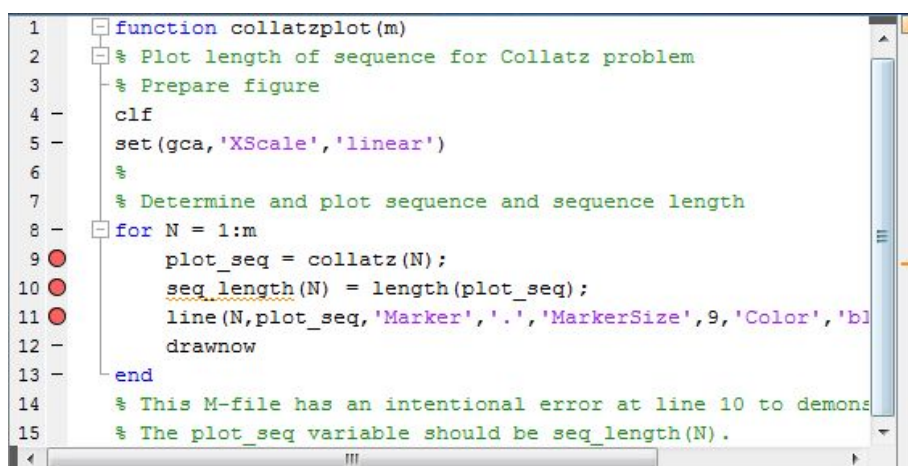


Figura 1.9 - Ejecución del Debugger (Imagen extraída: <http://www.mathworks.com/>)

¹⁹ Una explicación muy completa de este proceso se encuentra en el siguiente link:
http://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html#brqxeeu-178

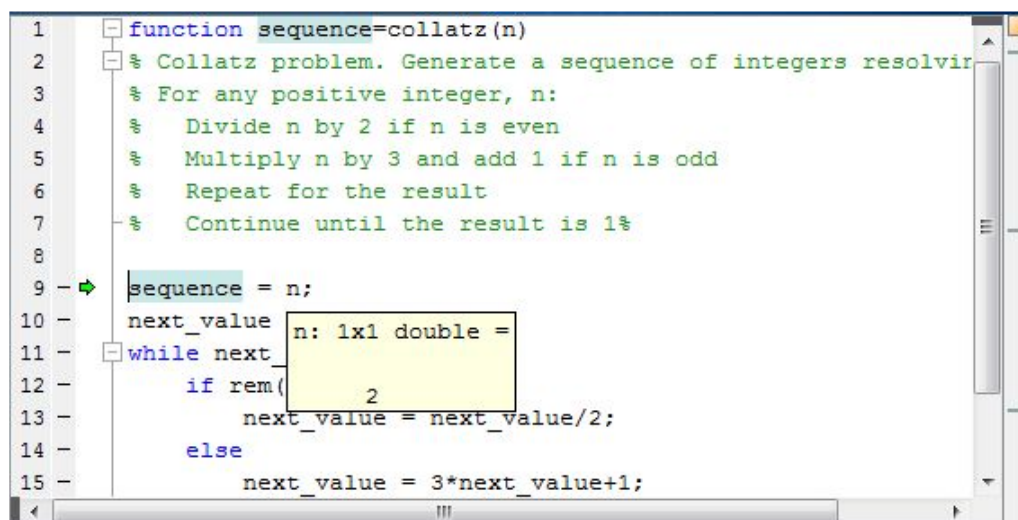


Figura 1.10 - Ejecución del Debugger (Imagen extraída: <http://www.mathworks.com/>)

Una vez ingresado en el proceso de Debug, se habilitan una variedad de iconos que permiten interactuar con el código paso a paso. El significado de estos iconos es el siguiente:

Tabla 1.1 - Botones del menú Debug.

Iconos	Nombres de icono del menú Debug	Descripción
	Run to Cursor	Continúa la ejecución del archivo hasta la línea donde está situado el cursor.
	Step	Avanzar un paso sin entrar en las funciones de usuario llamadas en esa línea.
	Step In	Avanzar un paso, y si en ese paso hay una llamada a una función cuyo fichero *.m está accesible, entra en dicha función.
	Continue	la ejecución hasta el siguiente breakpoint
	Step Out	Salir de la función que se está ejecutando en ese momento
	Quit Debugging	Terminar la ejecución del Debugger

El Debugger es un programa que hay que conocer muy bien, pues es muy útil para detectar y corregir errores. Es también enormemente útil para aprender métodos numéricos y técnicas de programación. Para aprender a manejar el Debugger lo mejor es practicar.

Cuando se está ejecutando un programa con el Debugger, en cualquier momento se puede ir a la línea de comandos de Matlab y teclear una expresión para ver su resultado. También se puede seleccionar con el mouse una sub-expresión en cualquier línea vista en el Editor/Debugger, hacer click con el botón derecho y en el menú contextual que se abre elegir Evaluate Selection. El resultado de evaluar esa sub-expresión aparece en la línea de comandos de Matlab.

Por último, también es posible introducir breakpoints condicionales (indicados con un punto amarillo, en vez de rojo), en los que el programa se para sólo si se cumple una determinada condición. Para introducir un breakpoint condicional basta clicar con el botón derecho en la correspondiente línea del código en la ventana del Editor/Debugger y elegir en el menú contextual que resulta Set/Modify Conditional Breakpoint.

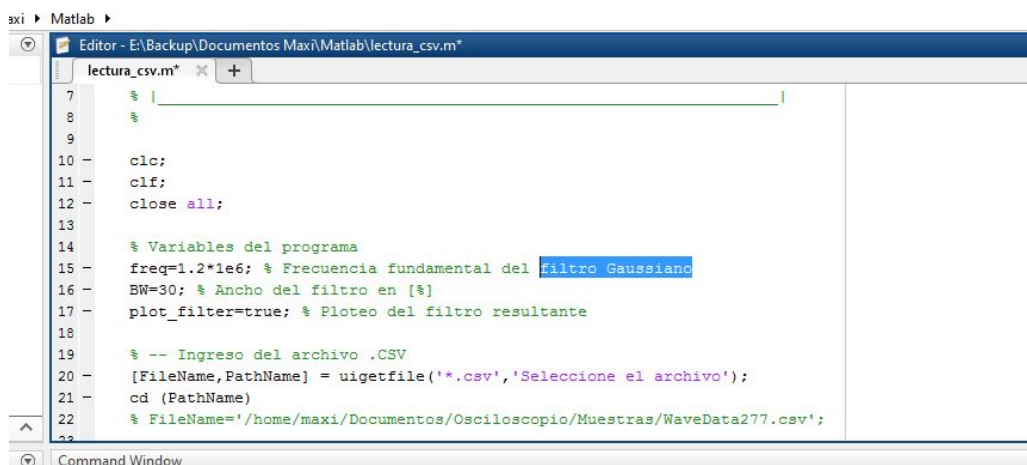
Comentario de fin de capítulo

A lo largo de todo el capítulo se habló mucho el costo computacional que conlleva algunas funciones, estructura y hasta ciertos tipos de variables, pero hoy en día es posible disminuir aún más los tiempo de procesamiento utilizando cálculo paralelo y distribuido. Para aprender más sobre esto recomendamos ver el siguiente link:

<http://es.mathworks.com/videos/parallel-computing-with-matlab-and-simulink-82042.html>

Por otra parte es muy común que a lo largo del programa las variables sufren muchos cambios, no solo en lo que respecta a su valor sino también en su nombre, para lo cual y con el objetivo de evitar estar trabajando de más, a continuación dos consejos para estos casos:

- Primero es recomendable (si bien al principio es engorroso) establecer un sector al principio del código donde se declaren todas las variables y/o constantes del programa. Obviamente hay casos en los que este consejo es excesivo, pero la idea del mismo es mantener un orden, con el tiempo cada uno adaptará esta práctica a sus necesidades. En la siguiente figura se observa un ejemplo de esto.



```

7  % |
8  %
9
10 clc;
11 clf;
12 close all;
13
14 % Variables del programa
15 freq=1.2*1e6; % Frecuencia fundamental del filtro Gaussiano
16 BW=30; % Ancho del filtro en [Hz]
17 plot_filter=true; % Ploteo del filtro resultante
18
19 % -- Ingreso del archivo .CSV
20 [FileName,PathName] = uigetfile('*.csv','Seleccione el archivo');
21 cd (PathName)
22 % FileName='/home/maxi/Documentos/Osciloscopio/Muestras/WaveData277.csv';

```

Figura 1.11 - Declaración de variable en el comienzo del código.

- Por otra parte es muy común que deseemos modificar el nombre de la variable o constante. Para lo cual es imprescindible, para que el código siga funcionando, que modifiquemos todas las instancias donde invocamos dicha variable o constante. Para facilitar la tarea cuando esté modificando el nombre presione Mayús+Intro y modificará automáticamente todas las instancias, como se observa en la siguiente imagen.

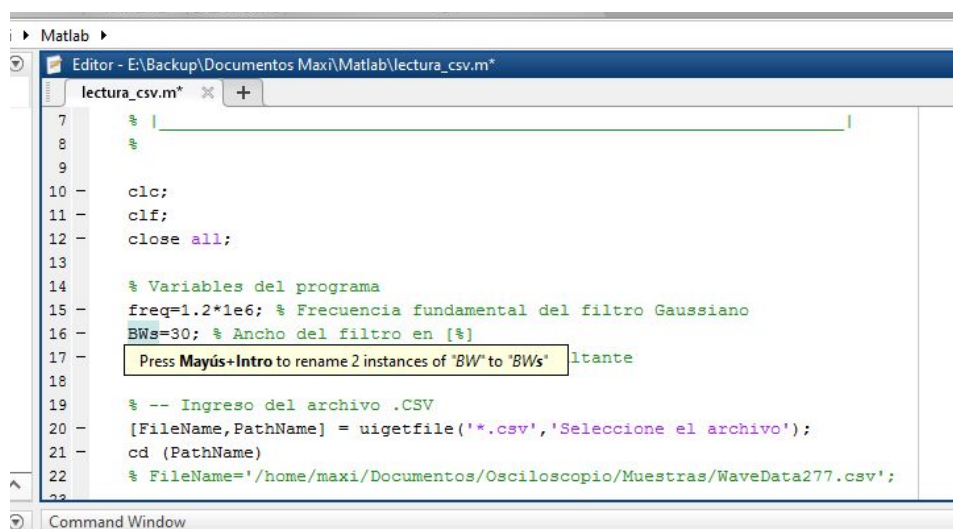


Figura 1.12 - Renombrado de variables o constantes.

Seguramente a esta altura se preguntará ¿Qué pasa si no recuerdo el nombre de una función o solo se las primeras letras del mismo? Bien la mejor forma de refrescar la memoria es a través de help, como se verá más adelante, pero pruebe presionando TAB, para desplegar todas las funciones que Matlab dispone con las letras iniciales ingresadas, como se observa en la siguiente figura.

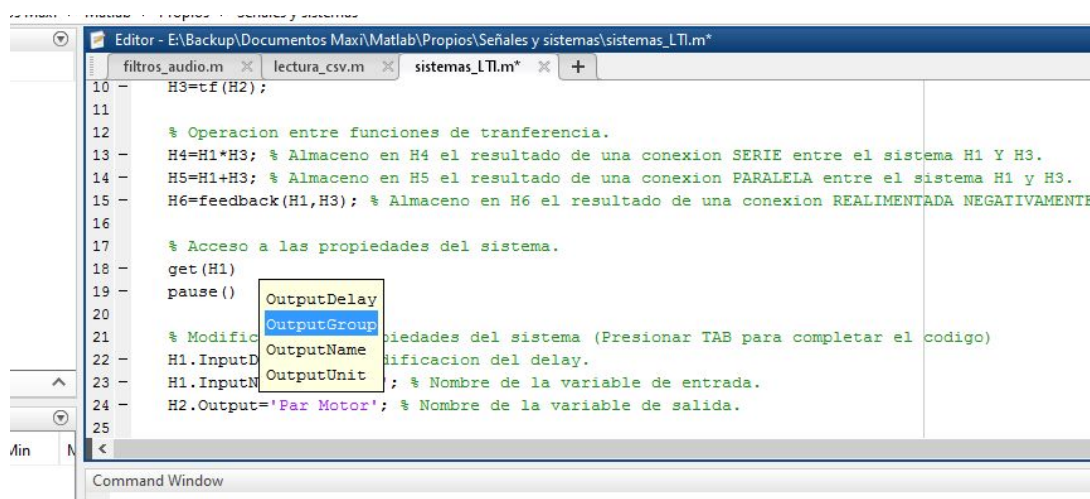


Figura 1.13 - Despliegue de opciones de funciones, por medio de la tecla TAB.

Comandos, caracteres y funciones usadas en el capítulo

Tabla 1.2 - Comandos, caracteres y funciones vistos a lo largo del capítulo 1.

Elementos	Definiciones
%	Indica un comentario.
%%	Indica una sección del código.
[]	Forma matricial.
()	Se utiliza para agrupar operaciones Además como elemento para encerrar el argumento de una función.
;	Separar filas en una definición de matriz. Además se utiliza para dar por finalizada una línea (Evita que dicha línea sea reproducida en el command window)
=	Operador de asignación, asigna un valor a una ubicación de memoria (ojo, no es lo mismo que una igualdad).
+	Suma escalar y de arreglos.
-	Resta escalar y de arreglos.
*	Multiplicación escalar y matricial.
/	División escalar y matricial.
\	El operador división-izquierda por una matriz (barra invertida \) equivale a pre-multiplicar por la inversa de esa matriz.
^	(Alt + 94) Exponenciación escalar y matricial.
'	Traspuesta.
. (En combinación con *, / o ^)	La operación que se combina con este carácter se realiza término a término (punto a punto).
'...'	(Alt + 39) Indica el ingreso de un string.
sqrt(x)	Indica raíz cuadrada del argumento.
fprintf (x)	Controla el formato del command window.
int(8), int(16), int(32) e int(64) uint(8), uint(16), uint(32) y uint(64)	Crea una variable entera de 8, 16, 32 y 64 bits respectivamente. La u, define a las variables sin signo.
intmin('int64') e intmax('int64')	Permiten saber el máximo y el mínimo valor que se puede representar con este tipo de variable. (Esta función se puede aplicar a: int8, int16, int32, uint8, uint16, uint32 y uint64)
class()	Devuelve el tipo de variable del argumento.

single()	Convierte la variable en single.
double()	Convierte la variable en double.
realmin() y realmax()	Permiten saber los números double más pequeño y más grande respectivamente.
realmin('single') y realmax('single')	Permiten saber los números single más pequeño y más grande respectivamente.
isfloat(x)	Permite saber si x es una variable real, de simple o doble precisión.
eps	Devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior.
realmin	Devuelve el número más pequeño con que se puede trabajar.
realmax	Devuelve el número más grande con que se puede trabajar.
logical (a)	Convierte valores numéricos a lógicos.
magic()	Crea una matriz inicial que tiene características especiales.
cputime	Devuelve el tiempo de CPU desde que el programa arrancó.
etime(t2, t1)y clock	Devuelve el tiempo transcurrido entre los vectores t1 y t2 (el orden es importante).
tic y toc	Imprime el tiempo en segundos. El comando tic pone el reloj a cero y toc obtiene el tiempo transcurrido.
iskeyword	Elabora una lista de nombres reservados por Matlab.
clc	Limpia la ventana de comandos.
clear	Borra todas las variables guardadas.
profiler	Permite saber cómo se ha empleado el tiempo de la CPU en la ejecución de un determinado programa.

Capítulo 2 - Manejo de array de n-dimensiones

Comenzando por el principio

Como se mencionó anteriormente el tipo de datos básico que se usa en Matlab es la matriz. Un solo valor, llamado escalar, se representa como una matriz 1×1 . Una lista de valores, ordenados o en una columna o en una fila, es una matriz unidimensional que se llama vector. Una tabla de valores se representa como una matriz bidimensional. Aunque este capítulo se limitará a escalares, vectores y matrices, Matlab puede manejar arreglos de orden superior (Holly M, 1990), como es el caso de hipermatrices²⁰. **A partir de ahora se denominan matrices a todo tipo de arreglo, sin importar su dimensión, a menos que se describa lo contrario.**

Para establecer una convención que seguiremos a lo largo del apunte, a continuación definimos:

- Escalar: $A = [10]$ / Dimensión= 1×1
- Vector: $B = [5 \ 2 \ 10 \ 23 \ 52]$ / Dimensión= 1×5
- Matriz: $C = \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix}$ / Dimensión= 2×2

Como se observa la dimensión $F \times C$ con F =filas y C = columnas. La ventaja de usar representación matricial es que todos los grupos de información se pueden representar con un solo nombre, por ejemplo una tabla como se observa a continuación

Tabla 2.1 - Mediciones de un recinto.

Posición	RT [seg]	Ruido Fondo [dB]
1	0.56	52
2	0.36	40
3	0.40	45

²⁰ Como por ejemplo una hipermatriz, es decir matrices de más de dos dimensiones. Una posible aplicación es almacenar con un único nombre distintas matrices del mismo tamaño (resulta una hipermatriz de 3 dimensiones). Los elementos de una hipermatriz pueden ser números, caracteres, estructuras, y vectores o matrices de celdas. El tercer subíndice representa la tercera dimensión: la "profundidad" de la hipermatriz

La tabla 2.1 tranquilamente puede pensarse como una matriz, como se observa a continuación:

$$\text{Mediciones} = \begin{bmatrix} 0.56 & 52 \\ 0.36 & 40 \\ 0.40 & 45 \end{bmatrix} \quad (2.1)$$

En esta matriz se combina dos tipos de datos con dos unidades diferentes, con la ventaja de estar en una sola variable.

Operaciones con escalares

Matlab maneja operaciones aritméticas entre dos escalares en forma muy parecida que en una calculadora. En la tabla 1.2, son presentadas la mayoría de la sintaxis.

Tabla 2.2 - Operaciones aritméticas entre dos escalares.

Operación	Sintaxis algebraica	Sintaxis en Matlab
Suma	$a + b$	$a + b$
Resta	$a - b$	$a - b$
Multiplicación	$a \times b$	$a * b$
División	$\frac{a}{b}$ $a \div b$	a/b
Exponenciación	a^b	$a \wedge b$

Por ejemplo:

```
A=1+2;
B=5;
C = A*B;
D=(C/B) ^A-100;
```

Lo importante de esto, tan simple, es aprender cómo se asignan los valores por medio del carácter “=”. **Entonces la primera línea del código se debe leer como “a A se le asigna un valor de 1 más 2”, que es la suma de dos cantidades escalares.** El operador asignación hace que el resultado de sus cálculos se almacenen en una ubicación de memoria de la computadora. Se podrá consultar el valor o los elementos que componen a una variable del workspace, si se ingresa su nombre en el command windows seguido de un ENTER (Holly M, 1990).

Como sucede en la sintaxis algebraica el orden de las operaciones es importante, Matlab utiliza las mismas reglas:

- Primero realiza los cálculos adentro de paréntesis, desde el conjunto más interno hasta el más externo.
- A continuación, realiza operaciones de exponenciación.
- Luego realiza operaciones de multiplicación y división de izquierda a derecha.
- Finalmente, realiza operaciones de suma y resta de izquierda a derecha.

Por lo cual, es importante agregar paréntesis “ () ” en expresiones grandes y de esta manera identificar claramente el orden de las operaciones. Si la expresión es un cociente de expresiones muy grandes, es recomendable subdividir la expresión en partes, como se observa a continuación.

$$f = \frac{\log(ax+bx+c) - \tan(ax+bx+c)}{4\pi x^2 + \cos(x-2) * ax} \quad (2.2)$$

Sería muy fácil cometer un error ingresando esta ecuación. Para minimizar la posibilidad de est, descomponerla ecuación en muchas piezas, es una buena estrategia. Por ejemplo, primero asigne valores para x, a, b y c:

```
x=9;
a=1;
b=3;
c=5;
```

Luego defino los argumentos y denominador:

```
argumentos=a*x+b*x+c;
denominador=4*pi*x^2+cos(x-2)*a*x;
```

La combinación de estos elementos da como resultado:

```
f=(log(argumento)-tan(argumento))/denominador;
```

Está muy claro que esta técnica depende del problema, la idea es minimizar la oportunidad de error, a través del ingreso de la menor cantidad de términos aislados.

Operaciones con arreglos

Como se mencionó en un principio, la principal fortaleza de Matlab es la manipulación de arreglos de n-dimensiones. A continuación se presenta la forma de definirlo de manera *explícita*:

```
x=[1 2 3 4]; %Arreglo de 1x4.
y=[1;2;3;4]; %Arreglo de 4x1.
z=[1 2 3 4; 5 6 7 8 9; 10 11 12 13]; %Arreglo de 4x4.
```

Aunque una matriz complicada tiene que ingresarse a mano, las matrices con intervalos regulares se pueden ingresar mucho más fácilmente:

```
b=[1:5];
```

El incremento por defecto es 1, pero si usted quiere usar un incremento diferente, colóquelo entre el primero y último valores en el lado derecho del comando. Por ejemplo,

```
c=[1:2:10];
```

También es posible hacer una matriz por intervalo decreciente. Matlab calcula el espaciado entre los elementos con el comando `linspace`. Especifique el valor inicial, el valor final y cuántos valores quiere en total. Por ejemplo,

```
d=linspace(1,10,3);
```

Las matrices se pueden usar en muchos cálculos con escalares sin problemas; sin embargo, la multiplicación y la división son un poco diferentes. En matemáticas matriciales, el operador de multiplicación (*) tiene un significado específico. Puesto que todas las operaciones Matlab pueden involucrar matrices, es necesario un operador diferente para indicar multiplicación elemento por elemento. Dicho operador es `.*` (que se llama multiplicación punto), por ejemplo:

```
x=[1 2 3 4];
y=[1 2 2 1];
z=x.*y;
```

Resulta que el elemento 1 de la matriz x, es multiplicado por el elemento 1 de la matriz y, elemento 2 de la matriz x, es multiplicado por el elemento 2 de la matriz y, elemento n de la matriz x, es multiplicado por el elemento n de la matriz y. Dando como resultado $z=[1 \ 4 \ 6 \ 4]$, lo cual es muy diferente a:

```
z=a*b;
```

Ya que sólo usar * implica una multiplicación matricial, que en este caso regresaría un mensaje de error porque x e y no satisfacen aquí las reglas para multiplicación en álgebra matricial (recuerden que para realizar este cálculo uno de los vectores debería ser un vector columna). La moraleja es: tenga cuidado al usar el operador correcto cuando quiera realizar multiplicación elemento por elemento (también llamado arreglo). La misma sintaxis se cumple para la división elemento por elemento (./) y la exponenciación (.^). de elementos individuales.

Todas las operaciones de la tabla 2.1, pueden ser aplicadas a arreglos de dimensiones mayores a 2.

Exhibir resultados

Notación científica

Aunque es posible ingresar cualquier número en notación decimal, no siempre es la mejor forma de representar números o muy grandes o muy pequeños. Por ejemplo, un número que se usa frecuentemente en acústica es el valores de presión sonora de referencia $0,00002 \text{ Pa}$ ²¹o de manera más significativa la constante de Avogadro²² que es 60220000000000000000000. La notación científica expresa un valor como un número entre 1 y 10, multiplicado por una potencia de 10 (el exponente). En notación científica, el número de Avogadro se convierte en 6.022×10^{23} , y el valor de presión de referencia para el aire es 20×10^{-6} . En Matlab, los valores en notación científica se designan con una e²³ entre el número decimal y el exponente. (Probablemente su calculadora usa notación similar.) Por ejemplo

```
Pref=20e-6;
Avogadro=6.022e23
```

²¹ Representa el valor de presión sonora mínimo que el ser humano promedio puede escuchar, siendo el aire el medio de transmisión, por ejemplo en el caso del agua el valor es 1uPa

²² En química y en física, la constante de Avogadro es el número de partículas elementales (usualmente átomos o moléculas) en un mol de una sustancia cualquiera.

²³ Aunque es una convención común usar e para identificar una potencia de 10, a veces confunden esta nomenclatura con la constante matemática e, que es igual a 2.7183. Para elevar e a una potencia, use la función exp().

Es importante omitir los espacios en blanco entre el número decimal y el exponente, sino se interpretan como dos valores a parte.

No importa cuál formato de despliegue elija, **Matlab usa en sus cálculos números punto flotante de doble precisión**. Exactamente cuántos dígitos se usan, depende de su cálculo. Sin embargo, cambiar el formato de despliegue no cambia la precisión de sus resultados. A diferencia de algunos otros programas, Matlab maneja los números enteros y decimales como números de punto flotante. Cuando los elementos de una matriz se despliegan en Matlab, los enteros siempre se imprimen sin punto decimal. No obstante, los valores con fracciones decimales se imprimen en el formato corto por defecto que muestra cuatro dígitos decimales (Holly M, 1990).

Tabla 2.3 - Formatos de despliegue numérico.

Comando	Tamaño	Ejemplo
<code>format short</code>	4 dígitos decimales	3.1416
<code>format long</code>	14 dígitos decimales	3.14159265358979
<code>format short e</code>	4 dígitos decimales	3.1416e+0
<code>format long e</code>	14 dígitos decimales	3.14159265358979e+0
<code>format bank</code>	2 dígitos decimales	3.14

Si ninguno de estos formatos de despliegue numérico predefinidos son adecuados es posible personalizar la salida con la función `fprintf`, la cual se explicará en detalle más adelante.

Guardar el trabajo

Como se observó hasta el momento todo el código o comando ingresado en el command windows es volátil al cerrar el Matlab, esto quiere decir que no es posible almacenar las variables o el código ingresado. En esta sección, primero se mostrará cómo guardar y recuperar variables (los resultados de las asignaciones y los cálculos) a archivos `.mat` o a archivos `.dat`. Luego se introducirán los archivos-m script, que se crean en la ventana del editor (que ya fue presentada). Los archivos-m script le permite guardar una lista de comandos y ejecutarlos más tarde.

Almacenar variables

Para preservar las variables que crean en la ventana de comandos y se visualizan en el workspace, es necesario guardar los contenidos de la ventana a un archivo. El formato por defecto es un archivo binario llamado archivo `.mat`. Pero sólo las variables, no la lista de comandos, son almacenadas en el archivo. Desde la barra de herramientas (figura 1.2), hacer click en `File` → `Save Workspace As`, luego se ingresa el nombre de archivo que

contiene las variables del workspace. Para restaurar un workspace, se utiliza el comando `load` (Holly M, 1990).

```
load file_name
```

Recordar utilizar los comandos `clc` y `clear`, para limpiar el command windows y el workspace respectivamente, y de esa manera evitar superponer información. Por otra parte como se observa en la figura 2.1, es posible desde el menú, importar información al workspace.

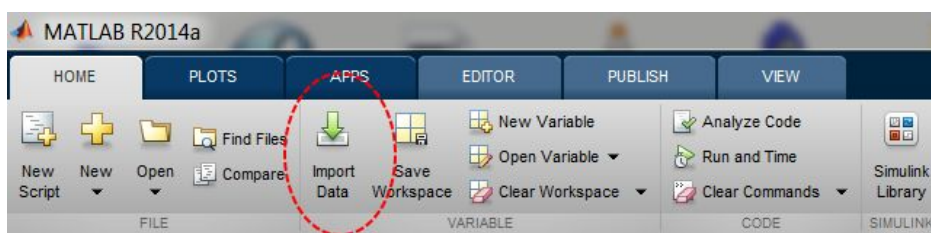


Figura 2.1 - Menú para cargar variables.

En el caso de querer almacenar una variable en particular el comando es el siguiente:

```
save file_name variable_list
```

donde `file_name` es el nombre de archivo definido por el usuario que designa la ubicación en memoria en la que desea almacenar la información²⁴, y dónde `variable_list` es la lista de variables a almacenar en el archivo. Por ejemplo,

```
save my_new_file a b
```

guardaría sólo las variables `a` y `b` en `my_new_file-mat`. Si los datos se usarán en un programa distinto a Matlab (como C o C++), el formato `.mat` no es apropiado porque los archivos `.mat` son exclusivos de Matlab. El formato ASCII es estándar entre plataformas de computadoras y es más apropiado para compartir archivos. Matlab le permite guardar archivos como archivos ASCII al modificar el comando `save` a:

```
save file_name variable_list -ascii
```

²⁴ Por defecto este archivo se almacena en el current folder, para más información:

http://www.mathworks.com/help/matlab/matlab_env/matlab-startup-folder.html?searchHighlight=current%20folder

El comando `-ascii` le dice a Matlab que almacene los datos en un formato de texto estándar de ocho dígitos. Los archivos ASCII se guardarán en un archivo `.dat` en lugar de en un archivo `.mat`; sólo hay asegúrese de agregar `.dat` en el nombre de archivo ya que sino Matlab le pondrá por defecto `.mat` (Holly M, 1990).

Para mayor precisión, los datos se pueden almacenar en un formato de texto de 16 dígitos:

```
save file_namen variable_list -ascii -double
```

Para recuperar los datos del directorio actual (current folder):

```
load file_name
```

Almacenar comandos

Retomando lo presentado en el capítulo 1, en la figura 1.2 (2), observamos la ventana del editor, donde se puede crear y guardar código en archivos llamados archivos-m. Un archivo-m es un archivo de texto ASCII similar a los archivos de código fuente de C. Se puede crear y editar con el editor/debugger (depurador) de archivo-m de Matlab o puede usar otro editor de texto de su elección asegúrese de que los archivos que guarde sean archivos ASCII. Notepad es un ejemplo de editor de texto que por defecto origina una estructura de archivo ASCII. Otros procesadores de palabra, como WordPerfect o Word, requerirán que usted especificar la estructura ASCII cuando guarde el archivo. Estos programas dan por defecto estructuras de archivo propietario que no se someten a ASCII, y pueden producir algunos resultados inesperados al intentar usar código escrito en ellos si no especifica que los archivos se guarden en formato ASCII. **Cuando se guarda un archivo-m, éste se almacena en el directorio actual (current folder). El nombre de los archivos deben ser válidos, esto es, un nombre que comience con una letra y contenga solo letras, números y el guión bajo (_). No se permiten los espacios** (Holly M, 1990).

Existen dos tipos de archivos-m, llamados scripts y funciones. Un archivo-m script es simplemente una lista de enunciados Matlab que se guardan en un archivo (por lo general, con una extensión de archivo `.m`). El script puede usar cualesquiera variables que se hayan definido en el área de trabajo, y cualesquiera variables que se creen en el script se agregaran al área de trabajo cuando el script termine. Se puede ejecutar un script creado en la ventana de edición Matlab al seleccionar el icono Save and Run (guardar y correr) de la barra de menú. De manera alternativa, se puede ejecutar un script al escribir el nombre del archivo o al usar el comando `run` de la ventana de comandos.

Se puede ejecutar una porción de un archivo-m al resaltar una sección y luego hacer clic derecho y seleccionar Evaluate Section (evaluar sección). También desde este menú puede comentar o “descomentar” secciones completas de código.

Comentario de fin de capítulo (Extraído de Chaparro L. F., 2011)

¿Analogico-continuo o digital-discreto?

En cálculo infinitesimal o simplemente cálculo, la variable independiente que interviene en las funciones es continua y por lo tanto la representación del concepto de derivada e integral son la razón del cambio de la función y el área bajo la gráfica de la función o el volumen de la misma, respectivamente. Por otro lado, en el cálculo discreto estos conceptos de derivada e integral son reemplazados por diferencias y sumatorias, respectivamente. Por lo cual, en un caso hablamos de ecuaciones diferenciales y en otro ecuaciones en diferencia. La matemática discreta (o cálculo discreto) hace posible la realización de cálculos en el dominio digital por medio de computadoras, a través de métodos numéricos que permiten la aproximación a derivadas e integrales o la solución a ecuaciones diferenciales.

En ingeniería, como en muchas áreas de la ciencia, las señales pueden ser de origen eléctrico, mecánico, químico o procesos biológicos los cuales se miden en función del tiempo con amplitudes expresadas en términos de tensión, corriente, torque, presión, etc. Estas funciones se denominan señales analógicas o continua en el tiempo, y para ser procesadas en una computadora deben ser convertidos en secuencias binarias o una cadena de unos y ceros por medio del muestreo, para luego realizar algún tipo de cálculo matemático sobre estas muestras y en vista que solo se posee una cantidad finita de muestras estos cálculos se pueden hacer sólo de manera aproximadamente, mediante métodos numéricos.

Representación en tiempo continuo y discreto

Hay una significativa diferencia entre tiempo continuo y tiempo discreto, una señal de tiempo discreto es una secuencia de adquisiciones tomadas a tiempos uniformes mientras que la señal analógica depende de forma continua en el tiempo. Así una señal de tiempo discreto $x[n]$ y su correspondiente señal en tiempo continuo $x(t)$ son relacionadas en un proceso de muestreo como se describe a continuación:

$$x[n] = x(nT_s) = x(t) |_{t=nT_s} \quad (2.3)$$

Donde $x[n]$ es obtenida por el muestreo de $x(t)$ a tiempos de $t = nT_s$, con n entero y T_s el periodo de muestreo o el tiempo entre muestra y muestra (como mejor lo entiendan). A este proceso se lo denomina muestreo (o Sampling) o discretización de una señal analógica. Claramente un valor pequeño de T_s hace una muy buena aproximación a la señal original en dominio analógico $x(t)$, pero a expensas de un consumo muy grande de memoria por la cantidad de muestras. Entonces surge una decisión de compromiso, ya que si disminuye considerablemente el valor de T_s se corre con el riesgo de obtener una señal muy poco precisa, lejos de ser una aproximación a $x(t)$. Por ejemplo, si consideramos la señal generada por un generador de señales del tipo:

$$x(t) = 2\cos(2\pi t) \Rightarrow \text{Con } 0 \leq t \leq 10 \text{ sec} \quad (2.4)$$

Si definimos 3 periodos de muestreo iguales a $T_{s1} = 0.1 \text{ sec}$, $T_{s2} = 0.5 \text{ sec}$ y $T_{s3} = 1 \text{ sec}$. La señal discreta que se desprende de (2.4) es:

$$x_1[n] = x(t)|_{t=0.1n} = 2\cos(2\pi n) \Rightarrow \text{Con } 0 \leq n \leq 10/0.1 \rightarrow 0 \leq n \leq 100 + 1 \text{ muestras} \quad (2.5)$$

$$x_2[n] = x(t)|_{t=0.5n} = 2\cos(2\pi n) \Rightarrow \text{Con } 0 \leq n \leq 10/0.5 \rightarrow 0 \leq n \leq 20 + 1 \text{ muestras} \quad (2.6)$$

$$x_3[n] = x(t)|_{t=1n} = 2\cos(2\pi n) \Rightarrow \text{Con } 0 \leq n \leq 10 \rightarrow 0 \leq n \leq 10 + 1 \text{ muestras} \quad (2.7)$$

En código esto es equivalente a escribir:

```
clc;
clear all;
%% Definicion del eje temporal [s]
t_final=10;
t=0:0.0001:t_final;
%% Definicion de los periodos de muestreo [s]
T1=0.1;
T2=0.5;
T3=1;
%% Definicion de la cantidad de muestras en funcion de T
n1=0:1*T1:t_final;
n2=0:1*T2:t_final;
n3=0:1*T3:t_final;
%% Definicion de la funcion continua (Analogica)
xt=2*cos(2*pi*t);
%% Definicion de las funciones discretas (Digital)
x1=2*cos(2*pi*n1);
x2=2*cos(2*pi*n2);
x3=2*cos(2*pi*n3);
%% Ploteo de los resultados
subplot(3,1,1)
stem(n1,x1)
hold on
plot(t,xt,'r')
title('Señal muestreada a T=0.1s')
ylabel('x1[nT1]')
subplot(3,1,2)
stem(n2,x2)
```



```

hold on
plot(t,xt,'r')
title('Señal muestreada a T=0.5s')
ylabel('x2[nT2]')
subplot(3,1,3)
stem(n3,x3)
hold on
plot(t,xt,'r')
title('Señal muestreada a T=1s')
xlabel('Tiempo [s]')
ylabel('x3[nT3]')

```

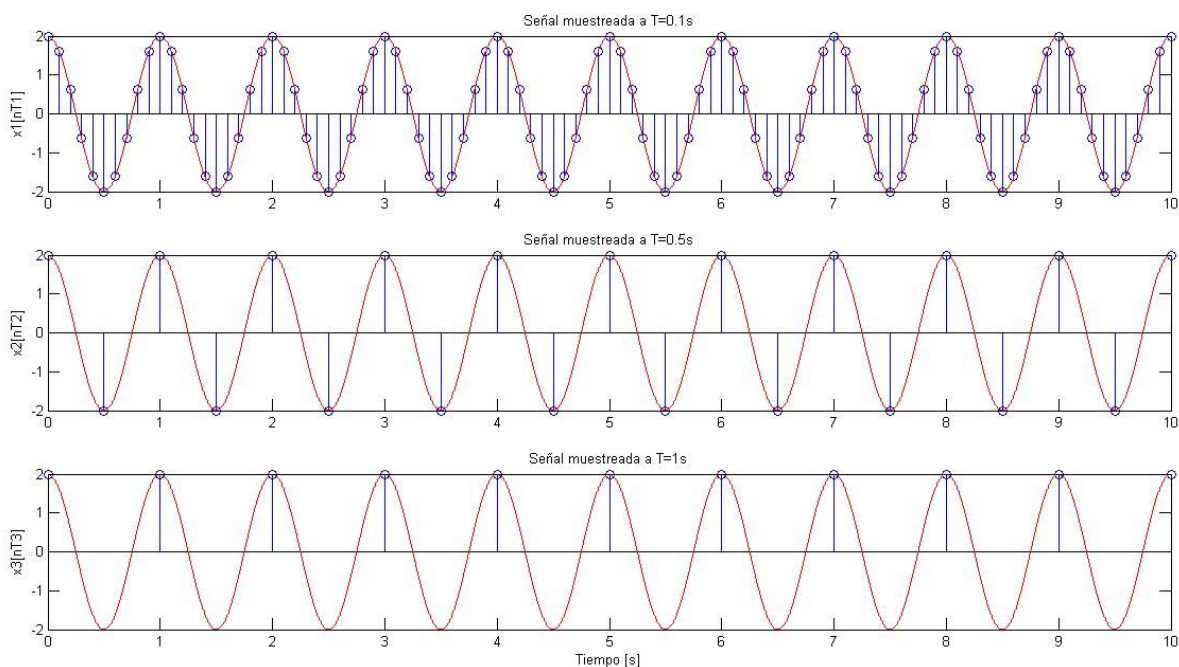


Figura 2.2 - Ploteo de la señal

Con esto se ha demostrado, el impacto significativo del periodo de muestreo T_s en una señal discreta en el tiempo sin perder información. Es inevitable pensar en en cual es el mejor T_s , bien la elección del período de muestreo depende del contenido de frecuencia de la señal continua a muestrear. En el ejemplo anterior (2.4) la frecuencia de la señal continua muestreada es de 1 Hz. Recordemos que una señal sinusoidal se define como se observa a continuación:

$$x(t) = A \cos(\omega t) \quad (2.8)$$

Donde A se define como la amplitud de la señal, $\omega = 2\pi f$, f la frecuencia de la señal en Hz y por último t es la variable independiente que en este caso describe la variable temporal de

la señal. Para entender el impacto de f , definimos la frecuencia de muestreo como $F_s = 1/T_s$. A partir de esto el código es:

```
clc;
clear all;
%% Definicion de la frecuencia de la señal [Hz]
f=10;
%% Definicion del eje temporal [s]
t_final=2;
t=0:0.0001:t_final;
%% Definicion de los frecuencia de muestreo [Hz]
F1=20;
F2=10;
F3=1;
%% Definicion de los periodos de muestreo [s]
T1=1/F1;
T2=1/F2;
T3=1/F3;
%% Definicion de la cantidad de muestras en funcion de T
n1=0:1*T1:t_final;
n2=0:1*T2:t_final;
n3=0:1*T3:t_final;
%% Definicion de la funcion continua (Analogica)
xt=2*cos(2*pi*f*t);
%% Definicion de las funciones discretas (Digital)
x1=2*cos(2*pi*f*n1);
x2=2*cos(2*pi*f*n2);
x3=2*cos(2*pi*f*n3);

%% Ploteo de los resultados
subplot(3,1,1)
stem(n1,x1)
hold on
plot(t,xt,'r')
title('Señal muestreada a Fs=20 Hz')
ylabel('x1[nT1]')
subplot(3,1,2)
```

```

stem(n2,x2)
hold on
plot(t,xt,'r')
title('Señal muestreada a Fs=10 Hz')
ylabel('x2[nT2]')
subplot(3,1,3)
stem(n3,x3)
hold on
plot(t,xt,'r')
title('Señal muestreada a Fs=1 Hz')
xlabel('Tiempo [s]')
ylabel('x3[nT3]')

```

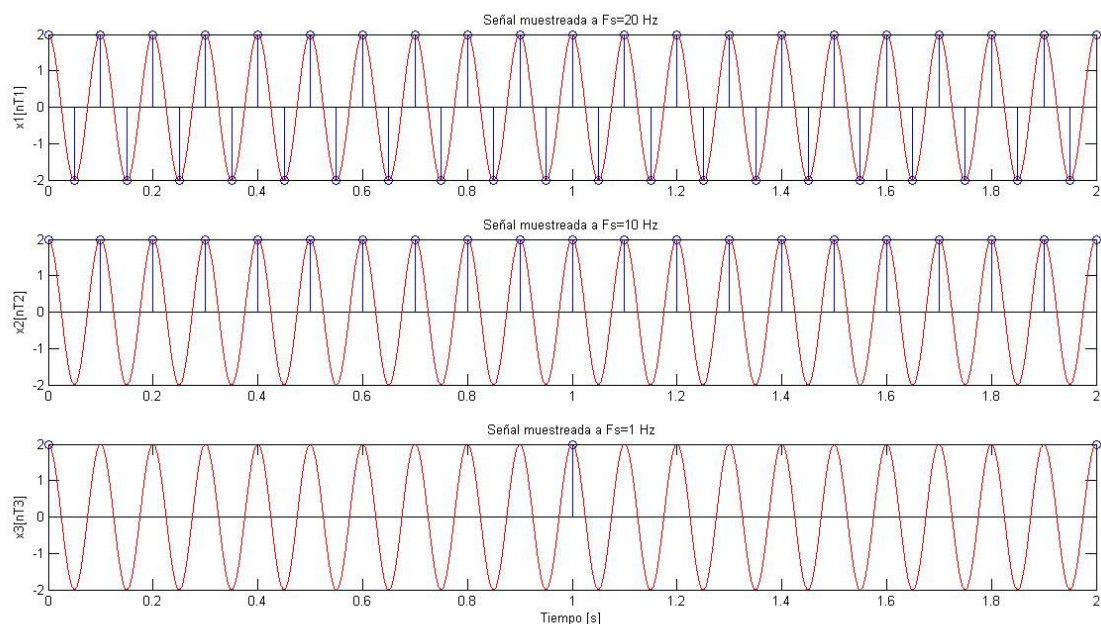


Figura 2.3 - Ploteo de la señal

En el caso de este último ejemplo, la frecuencia de la señal es de 10 Hz, observando los 3 plots se nota claramente cómo a medida que disminuimos F_s , la señal reconstruida está lejos de la señal original. Pero en el caso del primer plot donde F_s es exactamente el doble de la frecuencia de la señal continua, la misma puede ser reconstruida, esto es una primera aproximación al teorema de Nyquist-Shannon donde la condición mínima de muestreo para reconstruir una señal del dominio continuo es $F_s \geq 2f_{max}$.

Para conocer un poco más sobre muestreo de señales de tiempo continuo se recomienda leer el capítulo 4 de Tratamiento de señales en tiempo discreto de Alan V. Oppenheim.

Comandos, caracteres y funciones usadas en el capítulo

Tabla 2.4 - Comandos, caracteres y funciones vistos a lo largo del capítulo 2.

Elementos	Definiciones
linspace	Genera un vector con un espaciado determinado.
format short	Formato de despliegue de 4 dígitos decimales.
format long	Formato de despliegue de 14 dígitos decimales.
format short e	Formato de despliegue de 4 dígitos decimales.
format long e	Formato de despliegue de 14 dígitos decimales.
format bank	Formato de despliegue de 2 dígitos decimales.
save file_name variable_list	Almacena una lista de variable (variable_list) en current folder con el nombre file_name
load file_name	Recupera las variables del file_name

Capítulo 3 - Gráficos

Introducción

Las tablas de datos muy grandes son difíciles de interpretar. Los ingenieros usan técnicas de graficación para hacer que la información se entienda fácilmente. Con una gráfica es fácil identificar tendencias, elegir altos y bajos y aislar puntos de datos que pueden ser mediciones o cálculos de errores. Las gráficas también se pueden usar como una rápida verificación para determinar si una solución de computadora produce los resultados esperados (Holly M, 1990). Aquel que haya leído con anterioridad un manual o tutorial de matlab, notará muy apresurado la introducción a la graficación, pero en vista de la importancia de esta herramienta, se dispuso este orden, para introducir al lector conceptos que serían muy difícil de conceptualizar sin la ayuda de gráficos.

Por ejemplo, es muy útil representar la amplitud en función del tiempo de una señal sonora, amplitud en función de la frecuencia y fase en función del tiempo. Un caso muy útil de graficación es el caso de un espectrograma: Se puede observar como varía la amplitud y la frecuencia en función del tiempo, como se muestra en la figura 3.1.

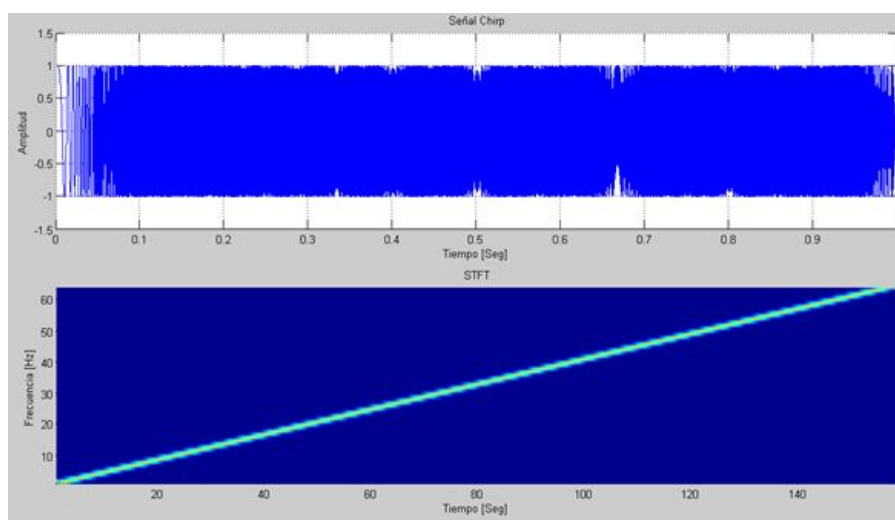


Figura 3.1 – Ejemplo de utilizar un espectrograma para analizar una función tipo Chirp.

Matlab nos permite realizar tanto gráficas bidimensionales como tridimensionales. A continuación se dará una breve explicación sobre cómo graficar y también se brindarán algunos ejemplos y ejercicios.

Gráficas Bidimensionales

Veamos primero la gráfica más simple: Una gráfica bidimensional. Para ello vamos a necesitar definir dos vectores, uno que represente al eje “x” y otro al valor del eje de “y”.

```
>> x = [1,2,3,4];
>> y = [0,1,1,2];
```

Una vez definidos ambos vectores, se prosigue a utilizar una función ya utilizada anteriormente en este apunte, pero no estudiada, la función `plot`.

Como ya se explicó, los usuarios pueden hacer uso de las distintas herramientas que ofrecen ayuda en MATLAB, para aprender a utilizar esta función con su máximo potencial.

Para graficar los puntos de los vectores x y y se prosigue a escribir:

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

Inmediatamente se abre una ventana la cual Matlab nombra automáticamente como “Figure 1”, con la gráfica correspondiente, como se aprecia en la figura 3.2

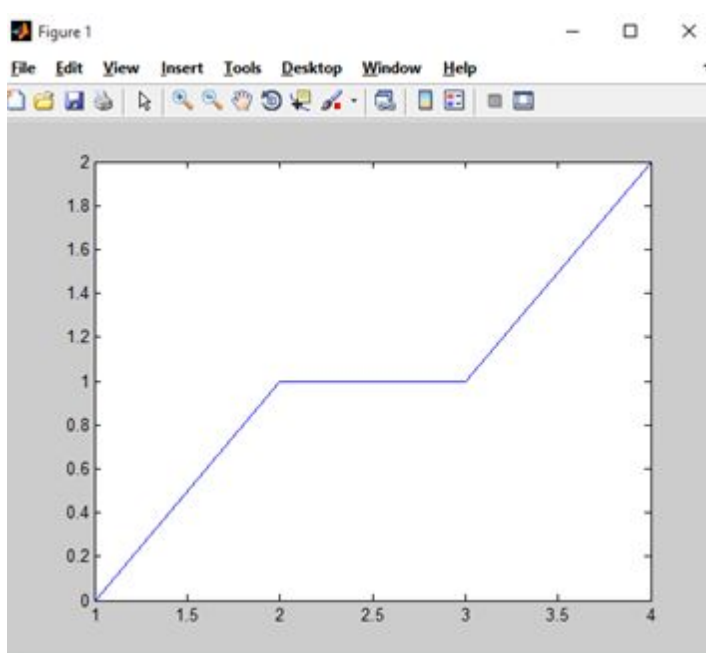


Figura 3.2 – Resultado de utilizar la función `plot`.

Como se puede observar, a cada elemento del vector x se le asigna un valor correspondiente de cada elemento del vector y . También se podría pensar que los puntos graficados son los pares ordenados conformados por:

Punto 1: $[x(1), y(1)]$
 Punto 2: $[x(2), y(2)]$
 Punto 3: $[x(3), y(3)]$
 Punto 4: $[x(4), y(4)]$

Al usar el comando `plot` automáticamente se interpolan los puntos graficados, obteniendo una sensación visual de una línea continua. El usuario debe acordarse de que, en el mundo digital, no existe lo continuo, y en realidad se grafica una sucesión de puntos muy cercanos entre sí.

Como se mencionó anteriormente, al utilizar las herramientas de ayuda se puede aprender qué parámetros de entrada admite la función `plot` y que parámetros se pueden modificar. Como se observa en el siguiente código, se pueden agregar un título y nombres a los ejes y agregarle una cuadrilla, entre otras cosas.

```
>> plot(x, y);
>> title 'Ejemplo de graficación sencilla'
>> ylabel 'Temperatura [°C]'
>> xlabel 'Tiempo [Seg]'
>> grid on
```

Obteniendo el resultado observable en la figura 3.3:

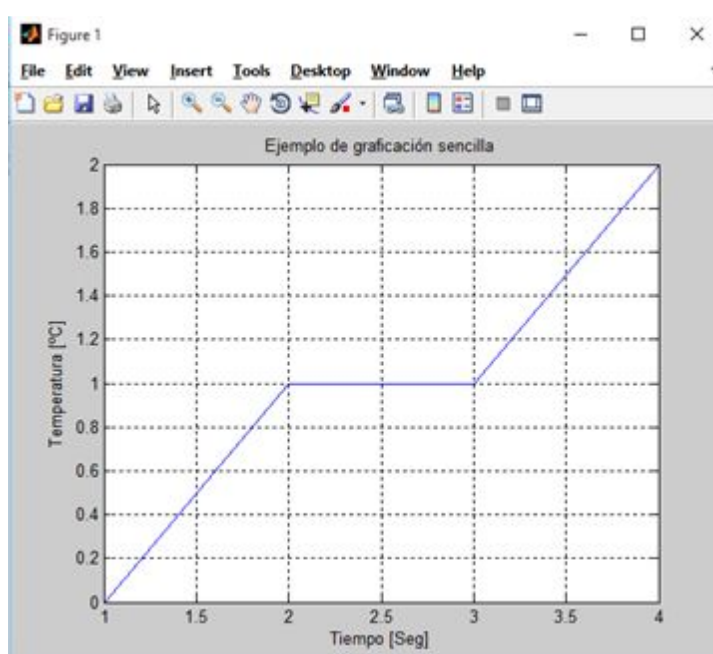


Figura 3.3 – Agregar títulos y nombre a los ejes.

También se puede elegir previamente en qué figura se procederá a realizar el gráfico, agregando el comando `figure(num)` antes de `plot`. Esto es de mucha utilidad cuando se desea realizar más de un gráfico. Si no decimos donde queremos el gráfico, Matlab sobrescribirá la figura existe (la figura abierta en primer lugar).

El formato debería ser el siguiente:

```
>> figure(1);
>> plot(x1, y1);
>> figure(2);
>> plot(x2, y2)
...
>> figure(n);
>> plot(xn, yn)
```

En la siguiente tabla se encuentran las funciones más básicas que dispone Matlab a la hora de realizar gráficos:

Tabla 3.1 - Funciones de graficación básicas (Holly M, 1990).

Función	Descripción	Ejemplo
<code>plot</code>	Crea una gráfica x-y	<code>plot(x, y)</code>
<code>title</code>	Agrega un título a una gráfica	<code>title('My Graph')</code>
<code>xlabel</code>	Agrega una etiqueta al eje x	<code>xlabel('Independent Variable')</code>
<code>ylabel</code>	Agrega una etiqueta al eje y	<code>ylabel('Dependent Variable')</code>
<code>grid</code>	Agrega una retícula a la gráfica	<code>grid</code> <code>grid on</code> <code>grid off</code>
<code>pause</code>	Detiene la ejecución del programa, lo que permite al usuario ver la gráfica	<code>pause</code>
<code>figure</code>	Determina cuál figura se usará para la gráfica actual	<code>figure(2)</code>
<code>hold</code>	Congela la gráfica actual de modo que se puede recubrir una gráfica adicional	<code>hold on</code> <code>hold off</code>

Gráficas con más de una línea

La creación de una gráfica con más de una línea se puede lograr en muchas formas. Por defecto, la ejecución de un segundo enunciado `plot` borrará la primera gráfica. Sin embargo, puede apilar las gráficas unas encima de otras con el comando `hold on`. Ejecute los siguientes enunciados para crear una gráfica con ambas funciones graficadas en la misma gráfica, como se muestra en la figura 3.4 (Holly M, 1990):

```
>> x = 0:pi/100:2*pi;
>> y1 = cos(x*4);
>> plot(x, y1);
>> y2 = sin(x);
>> hold on;
>> plot(x, y2);
```

Los puntos y coma son opcionales tanto en el enunciado `plot` como en el enunciado `hold on`. Matlab continuará poniendo en capa las gráficas hasta que se ejecute el comando `hold off`.

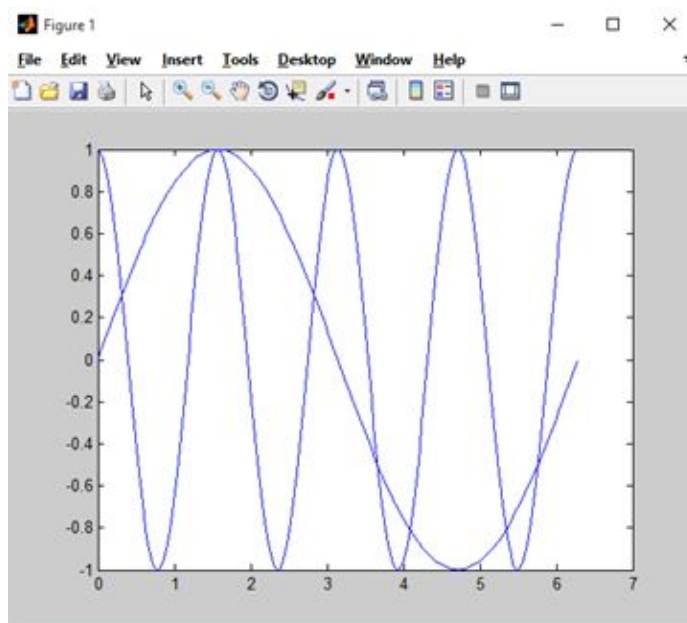


Figura 3.4 - Se puede usar el comando `hold on` para poner en capas las gráficas sobre la misma figura.

Otra forma de crear una gráfica con múltiples líneas es solicitar ambas líneas en un solo comando `plot`. Matlab interpreta la entrada a `plot` como vectores alternos `x` y `y`, como en

```
>> Plot(x, y1, x, y2)
```

Obteniendo de esta forma el mismo resultado que en el caso anterior, salvo por un detalle no insignificante: Con este método Matlab interpreta que son líneas distintas y les asigna un color diferente a cada curva, como se muestra en la figura 3.5.

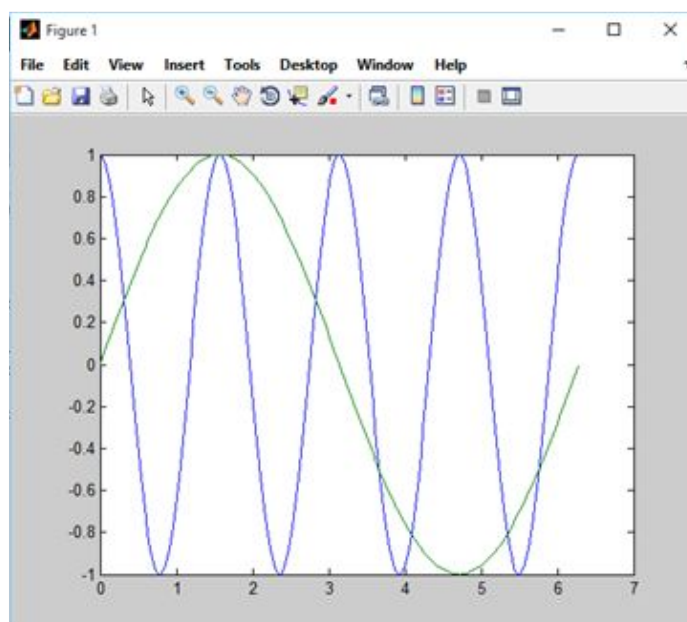


Figura 3.5 - Asignación automática de colores.

Gráficas de arreglos complejos

Si la entrada al comando `plot` es un arreglo sencillo de números complejos, Matlab grafica el componente real en el eje x y el componente imaginario en el eje y. Por ejemplo, si

```
>> A = [0+0i, 1+2i, 2+5i, 3+4i];
```

entonces regresa la gráfica que se muestra en la figura 3.6 (Holly M, 1990).

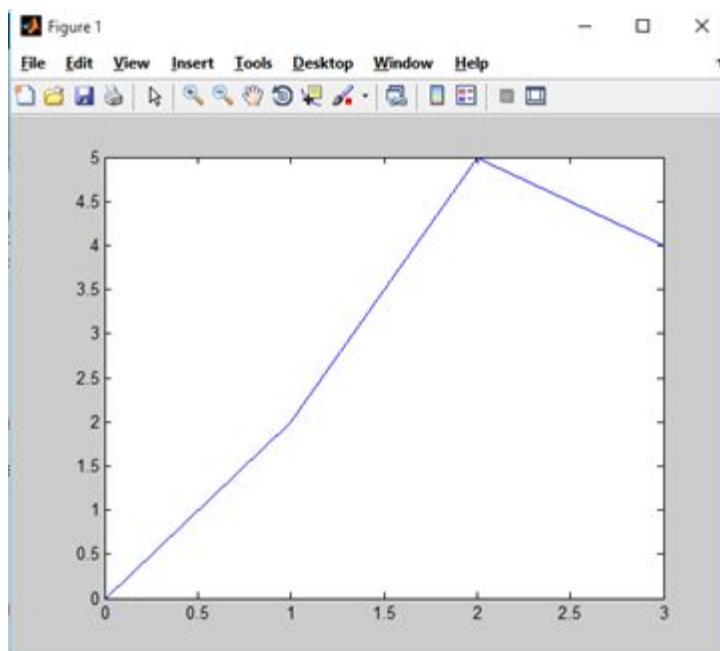


Figura 3.6 – Uso de `plot` con números complejos.

Línea, color y estilo

El usuario puede cambiar la apariencia de las gráficas que realiza. para ello, se recomienda usar `help plot` para aprender a utilizar estas aplicaciones. A continuación se presenta un ejemplo de un código que cambia parámetros como color y línea.

```
% Cambio de línea y color
x = 1:1/100:10;
y = sin(x);
figure(1);
plot(x, y, 'r--');
figure(2);
plot(x, y, 'Color', 'red', 'LineWidth', 2);
```

Como se puede observar, el parámetro `'r--'` cambia el estilo de línea y el color. `r` especifica color rojo (red) y `--` el tipo de trazado a utilizar. El usuario puede investigar los

distintos colores y distintos trazos que acepta Matlab para graficar.

En el caso del `plot` de la `figure(2)`, el color se cambia de otra manera distinta, válida también, según aparece en la ayuda. Otro parámetro que se modifica es el ancho de la curva, a través de la entrada '`LineWidth`', como muestra la figura 3.7.

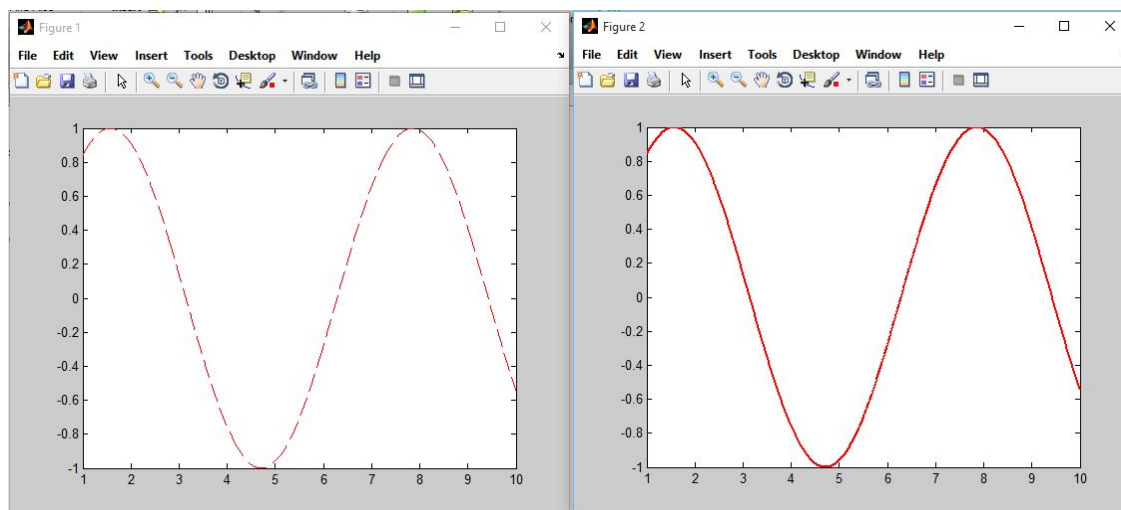


Figura 3.7 - Línea, color y trazo del graficador.

Subgráficas

Con el comando `subplot` se puede dividir la ventana de graficación en para realizar muchos gráficos en ella. La función se utiliza de esta manera: `subplot(m,n,N)`. De esta manera, se indica que la figura se divide en $m \times n$ y que se grafica en la ventana N . Para ilustrar su funcionamiento, nada mejor que un ejemplo:

```
% Ejemplo subplot

x = 0:1/100:10;
y1 = sin(x);
y2 = cos(x);
y3 = cos(x) + sin(x);

subplot(3, 1, 1);
plot(x, y1, 'b-');
title 'Sen (x) '
subplot(3, 1, 2);
plot(x, y2, 'r--');
title 'Cos (x) '
subplot(3, 1, 3);
plot(x, y3, 'g-.');
title 'Sen (x) + Cos (x) '
```

En el código anterior, se divide la ventana de gráficos en tres partes, por eso se especifica dentro de `subplot` una matriz de 3×1 . Con la última variable de entrada se indica qué sub-ventana estará activa. Corriendo el código se obtiene lo representado en la figura 3.8:

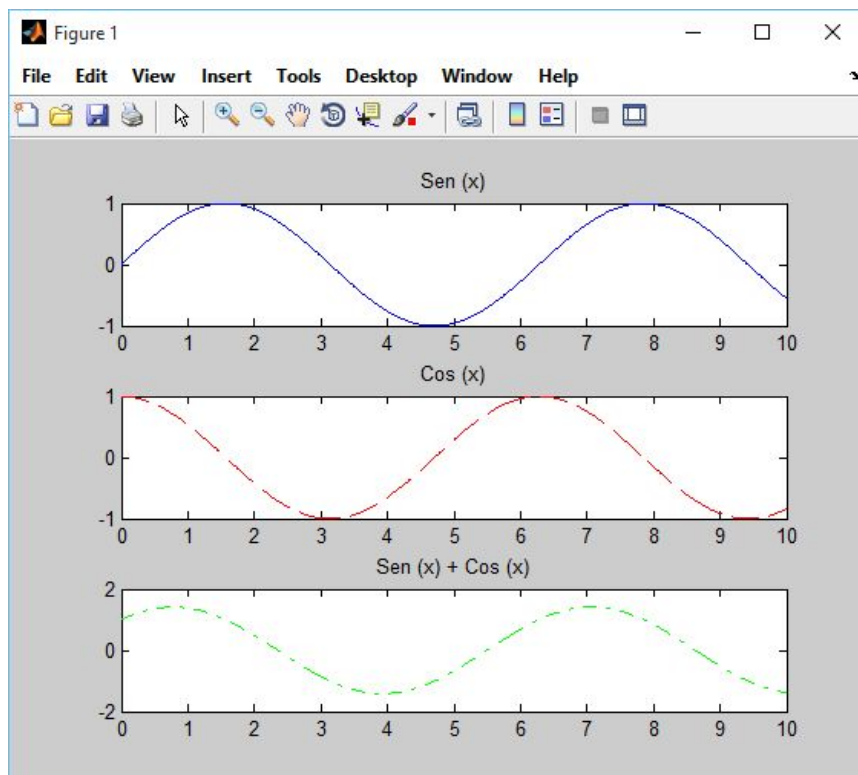


Figura 3.8 - Múltiples gráficas en una figura.

Otros tipos de gráficas

Matlab admite otro tipo de gráficas, aparte de los gráficos tipo x-y. A continuación se mencionan algunos, acompañados de un ejemplo ilustrativo:

- `polar(theta, r)`: Realiza un gráfico en coordenadas polares, cuyas variables de entradas son `theta` (ángulo en radianes) y `r` (radio).
Ejemplo:

```
% Polar
theta = (0:0.01:2)*pi;
r = 5*cos(4*theta);
polar(theta, r);
```

El resultado de correr este código en Matlab, es el gráfico en coordenadas polares que se muestra en la figura 3.9:

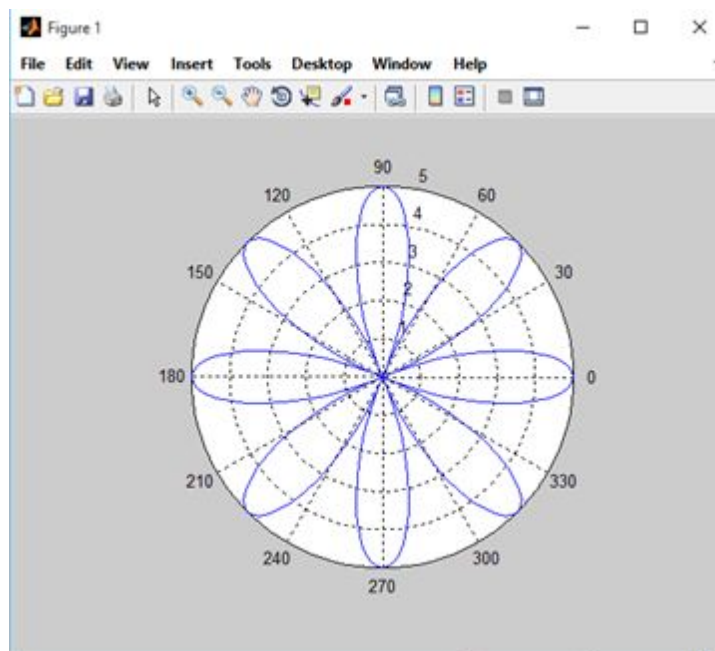


Figura 3.9 - Uso de función `polar`.

- `semilogx(x, y)`: Genera un gráfico x-y como es el caso de `plot` salvo por el hecho que aplica una escala logarítmica al eje "x". Esto resulta muy útil, por ejemplo, cuando se desea graficar el espectro de una señal y visualizar en el eje de frecuencias las bandas de fracción de octava normalizados. También están disponibles las funciones `semilogy` y `loglog`. La primera funciona igual que el caso de `semilogx` pero aplica una escala logarítmica al eje "y" y, la segunda, aplica el escalonamiento a ambos ejes.

Ejemplo:

```
% Semilogx - Semilogy - Loglog
x = 0:0.1:10;
y = 2*x + 5;

subplot(2,2,1)
plot(x, y)
title 'Lineal'
grid on
subplot(2,2,2)
semilogx(x, y)
title 'Eje "x" logarítmico'
grid on
subplot(2,2,3)
semilogy(x, y)
title 'Eje "y" logarítmico'
grid on
subplot(2,2,4)
loglog(x, y)
title 'Ambos ejes logarítmicos'
```

```
grid on
```

En el ejemplo anterior se grafica una recta utilizando las funciones vistas. En la figura 3.10 se observan los resultados:

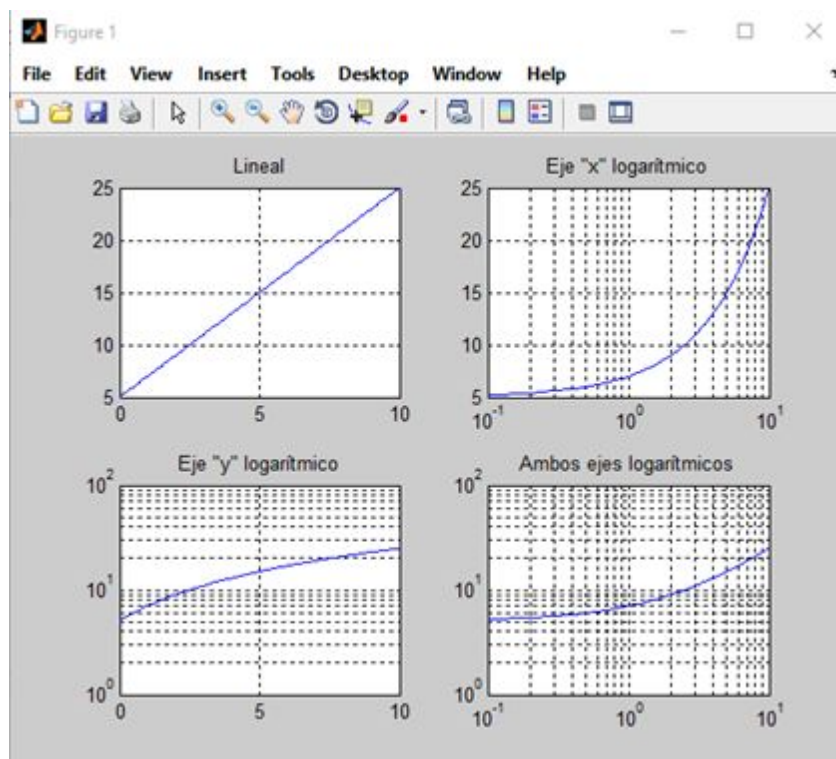


Figura 3.10 - Ejes logarítmicos.

- `bar(x)`, `bar3(x)`, `pie(x)`: Estas funciones generan gráficos de barra, de barra en tres dimensiones y tipo torta, respectivamente. En el caso de `bar` hay diferencias cuando la entrada `x` es un vector o una matriz. Recordamos nuevamente al lector utilizar siempre la ayuda antes de utilizar una función por primera vez, o que su uso no le sea muy familiar.

Ejemplo:

```
% Gráfico de barra y torta
x = [.5,1,3,2,6];
y = [.5,1.5;1,2;3,4;2,3;6,7];

subplot(2,2,1)
bar(x)
title 'Gráfico de barras'
subplot(2,2,2)
bar(y)
title 'Gráfico de barras - Comparación'
subplot(2,2,3)
bar3(x)
title 'Gráfico de barras en 3D'
```

```
subplot(2,2,4)
pie(x)
```

En este ejemplo, se utiliza la función `bar` tanto con una matriz como con un vector, para que el lector aprecie las diferencias. también se emplean las funciones `pie` y `bar3`, y el resultado de ejecutar este código se muestra en la figura 3.11.

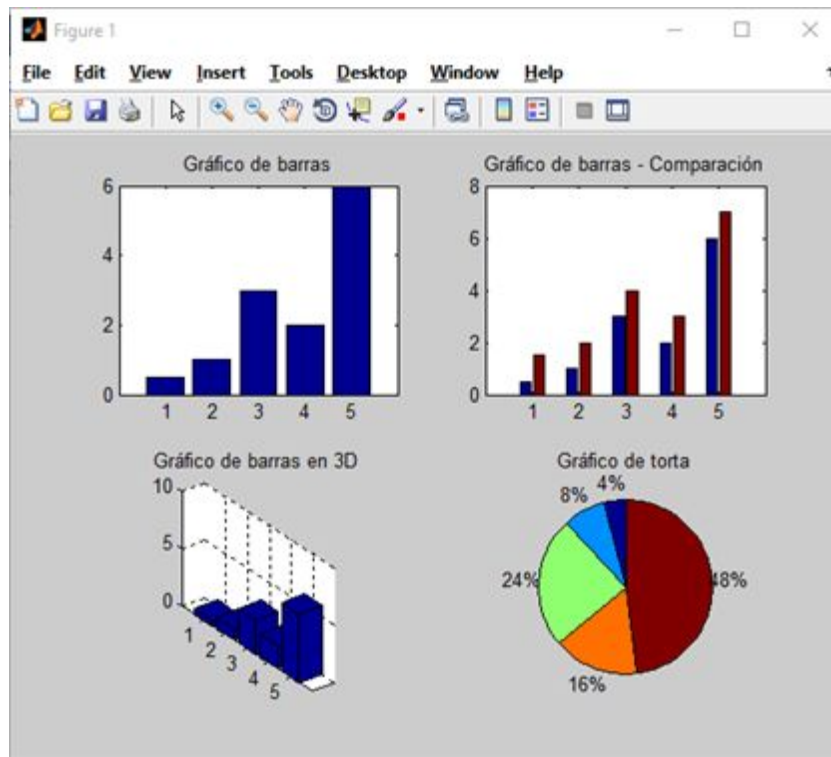


Figura 3.11 - Gráficos de barra y de torta.

- `stem(x,y)`: Esta función realiza un gráfico tipo x-y, con la excepción que no interpola los puntos graficados, sino que los dibuja como puntos aislados sobre líneas verticales. Muy útil para esta materia donde se puede utilizar para graficar señales discretas.

Ejemplo:

```
% Stem
x = 0:.2:8;
y = cos(x);
stem(x, y);
```

Como se puede observar, la función `stem` se utiliza de la misma forma que `plot`. Los resultados se muestran en la figura 3.12:

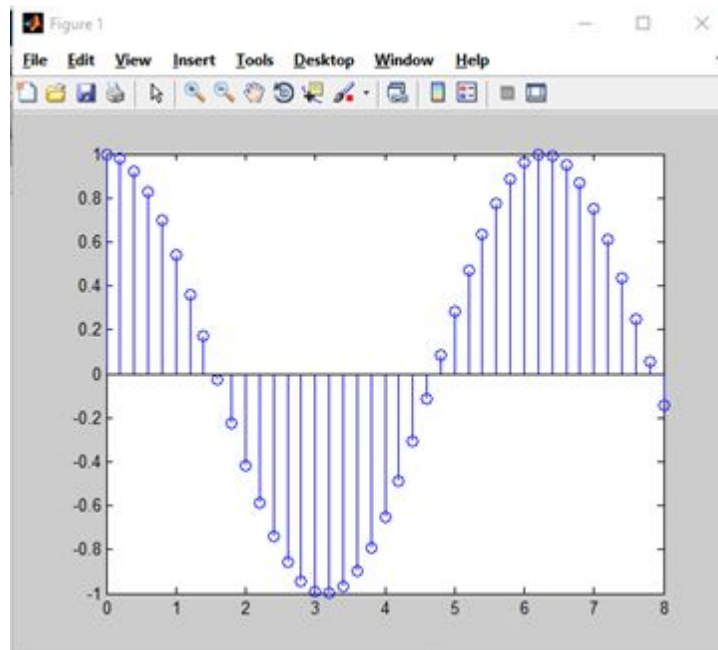


Figura 3.12 - Uso de la función `stem`.

Comandos, caracteres y funciones usadas en el capítulo

Tabla 3.2 - Comandos, caracteres y funciones usadas en el capítulo 3

Elemento	Descripción
plot	Crea una figura y realiza una gráfica con las matrices ingresadas
grid	Habilita la grilla en una figura
title	Permite colocar un título en una figura
xlabel ylabel	Permite colocar nombre en los ejes “x” e “y”
figure	Abre una figura e indica dónde se realizará la gráfica
hold	Permite mantener una figura activa
subplot	Divide una figura en subgráficas
lineWidth	Comando para establecer el espesor de línea en una gráfica
color	Permite seleccionar el color de línea
semilogx semilogy loglog	Crea una gráfica con el eje “x” logarítmico Crea una gráfica con el eje “y” logarítmico Crea una gráfica con ambos ejes logarítmicos
polar	Crea una gráfica en coordenadas polares
bar bar3 pie	Gráfico de barra Gráfico de barra en tres dimensiones Gráfico tipo torta
stem	Gráfica sin interpolar

Capítulo 4 - Estructuras de control

Introducción

Una forma de resolver un problema es considerar la solución como una lista de enunciados, que dan una suerte de paso a paso, hasta llegar al producto final. Por lo general, las secciones de un código se pueden categorizar en una de tres estructuras: secuenciales, estructuras de selección y estructuras de repetición (Holly M, 1990).

- Las secuencias son listas de comandos que se ejecutan una después de otra.
- Una estructura de selección permite al programador ejecutar un comando (o conjunto de comandos) si algún criterio es verdadero, y un segundo comando o conjunto de comandos si el criterio es falso. Un enunciado de selección proporciona los medios de elegir entre dichas rutas, con base en una condición lógica. Las condiciones que se evalúan con frecuencia contienen operadores tanto relacionales como lógicos o funciones.
- Una estructura de repetición, o bucle, hace que un grupo de enunciados se ejecute varias veces. El número de veces que se ejecuta un bucle depende de un contador o de la evaluación de una condición lógica (Holly M, 1990).

Diagrama de flujo

Como se presentó anteriormente en el capítulo 1, todas estas estructuras son más fácil de representar a través de un diagrama de flujo. Matlab posee un lenguaje que dispone de sentencias para realizar selección (o bifurcaciones) y repeticiones (o bucles). A continuación se muestran algunos ejemplos de bifurcación.

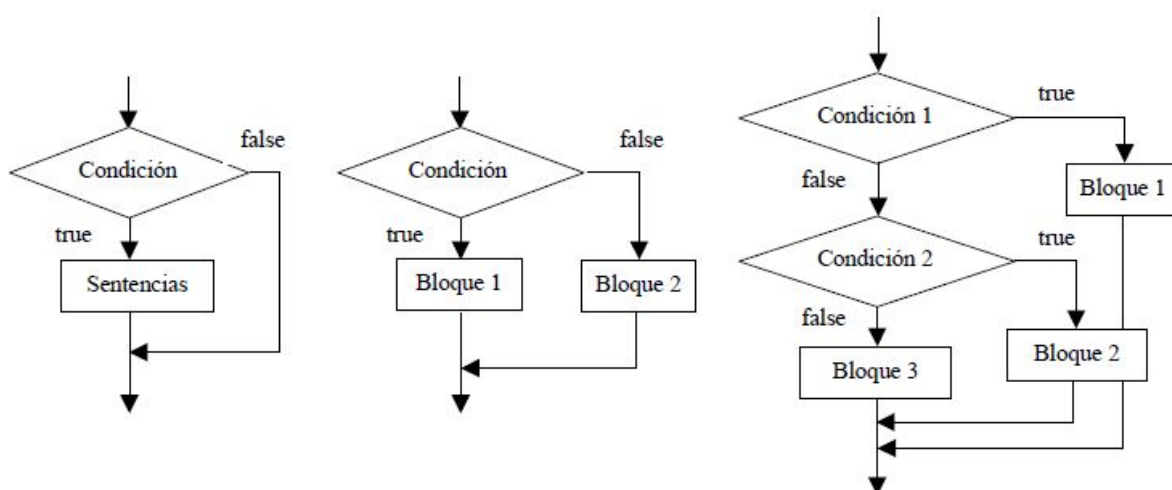


Figura 4.1 - Diagramas de flujo de bifurcaciones (García de Jalón J., Rodríguez I. J, Vidal J., 2005)

Los bucles permiten repetir las mismas o análogas operaciones sobre datos distintos. Mientras que en C/C++/Java el "cuerpo" de estas sentencias se determinaba mediante llaves {...}, en Matlab se utiliza la palabra `end` con análoga finalidad. Existen también algunas otras diferencias de sintaxis.

La Figura 4.2 muestra dos posibles formas de bucle, con el control situado al principio o al final del mismo. Si el control está situado al comienzo del bucle es posible que las sentencias no se ejecuten ninguna vez, por no haberse cumplido la condición cuando se llega al bucle por primera vez. Sin embargo, si la condición está al final del bucle las sentencias se ejecutarán por lo menos una vez, aunque la condición no se cumpla (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

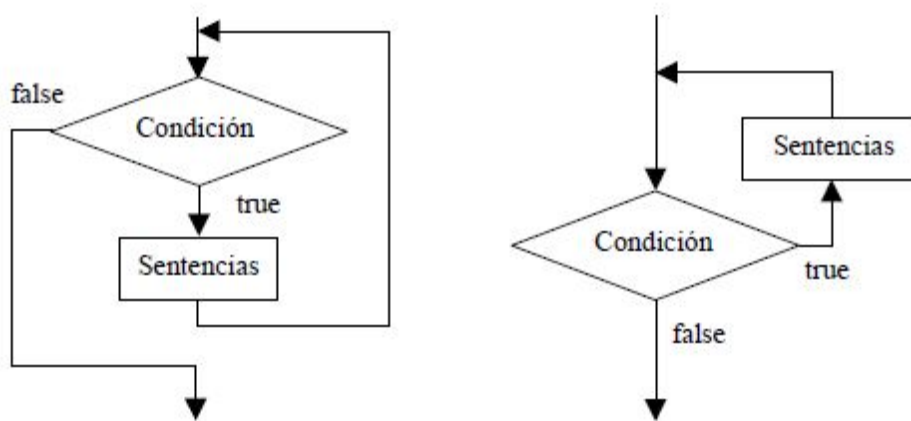


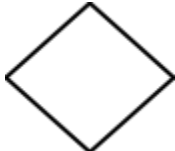


Figura 4.1 - Diagramas de flujo de bucles (García de Jalón J., Rodríguez I. J, Vidal J., 2005).

La siguiente tabla muestra el significado de cada elemento en un diagrama de flujo:

Tabla 4.2 - Operadores relacionales.

Forma	Significado
	El óvalo se usa para indicar el comienzo o final de una sección de código.
	El paralelogramo se usa para indicar procesos de entrada o salida.
	El rombo indica un punto de decisión.

	El cuadrado indica cálculos.
---	------------------------------

Estructuras de selección

Sentencia *IF*

En su forma más simple, la sentencia if se escribe en la forma siguiente (obsérvese que a diferencia de C/C++/Java la condición no va entre paréntesis, aunque se pueden poner si se desea²⁵)

```
if condicion
    sentencia
end
```

Existe también la bifurcación múltiple, en la que pueden concatenarse tantas condiciones como se desee, y que tiene la forma:

```
if condicion1
    bloque1
elseif condicion2
    bloque2
elseif condicion3
    bloque3
else
    bloque 4
end
```

else cumple la función de ser la opción por defecto **cuando no se cumple** las condiciones 1, 2 o 3, la misma puede ser omitida: si no está presente no se hace nada en caso de que no se cumpla ninguna de las condiciones.

Una observación muy importante: la condición del if puede ser una condición en donde intervengan matrices, del tipo $A=B$, donde A y B son matrices del mismo tamaño. Para que

²⁵ Recomendamos escribir el código con una metodología más genérica, esto quiere decir que a pesar que Matlab pasa muchas cosas por alto con respecto a la escritura y su formato, es imprescindible mantener una coherencia con el resto de lenguajes para que el pasaje de uno a otro no sea tan brusco. Por ejemplos en el siguiente ejemplo las sentencias aparecen desplazadas hacia la derecha respecto al if o end. Esto se hace así para que el programa resulte más legible, resultando más fácil ver dónde empieza y termina la bifurcación o el bucle. **Es muy recomendable seguir esta práctica de programación. Esto se realiza fácilmente presionando la tecla Tab, para mover el código una cantidad fija de espacios. Por último, estas estructuras se cierran con la end, la cual debe estar alineada con su palabra de inicialización en este caso if.**

se considere que la condición se cumple, es necesario que sean iguales dos a dos todos los elementos de las matrices A y B ($a_{ij}=b_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq n$).

Las condiciones pueden ser combinadas con operadores relacionales o operadores lógicos:

Tabla 4.2 - Operadores relacionales.

Operación	Sintaxis Matlab
Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=
Igual que	==
Distinto que	~=

Tabla 4.3 - Operadores lógicos.

Operación	Sintaxis Matlab
And	&
Or	
XOR (Or exclusivo)	xor (A,B)
Not	~

En Matlab los operadores relaciona pueden aplicarse a vectores y matrices. Al igual que en C, si una comparación se cumple el resultado es 1 (true), mientras que si no se cumple es 0 (false). Recíprocamente, cualquier valor distinto de cero es considerado como true y el cero equivale a false. La diferencia con C está en que cuando los operadores relacionales de Matlab se aplican a dos matrices o vectores del mismo tamaño, la comparación se realiza elemento a elemento, y el resultado es otra matriz de unos y ceros del mismo tamaño, que recoge el resultado de cada comparación entre elementos. Los operadores lógicos operan con números binarios como se observa a continuación:

Tabla 4.4 - Operaciones con operadores lógicos.

Inputs A - B		and (A & B)	or (A B)	xor (A,B)	not (~A)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

A continuación se presenta ejemplos de aplicación de los operadores presentados anteriormente:

```
x=[1, 2, 3, 4, 5];
y=[-2, 0, 2, 4, 6];
z=4;
a=x<y;
b=x==y;
c=x~=y;
d=x>=y;
e=x==y & x==z;
```

El resultado de estas operaciones son:

```
a= 0 0 0 0 1
b= 0 0 0 1 0
c= 1 1 1 0 1
d= 1 1 1 1 0
e= 0 0 0 1 0
```

Si combinamos la sentencia if con los operadores recientemente presentados, logramos los siguientes resultados:

```
if x>0
    y=log(x);
else
    beep
    disp26('La entrada a la función log debe ser positiva' );
end
```

A continuación se presenta un ejemplo un poco más avanzado donde interviene una estructura más compleja. Se utilizara esta estructura para determinar si el nivel de presión sonora expuesto por un operario es dañino en función de las recomendaciones de la Organización Mundial de la Salud:

```
if dB<=85
    disp('Nivel confortable');
elseif dB>=85 & dB<100
    disp('Nivel dañino');
elseif dB>=100 & dB<120
    disp('Nivel dañino, con posibles daños irreversibles' );
else
    disp('Límite del umbral del dolor, daños irreversibles' );
```

²⁶ La función disp permite imprimir en pantalla resultados.

end

El diagrama de flujo del anterior código es:

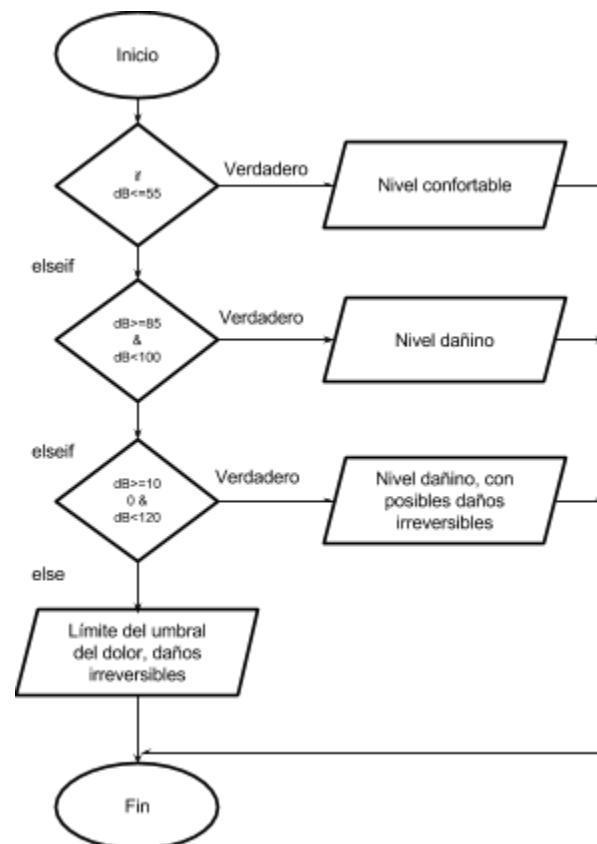


Figura 4.2 - Diagramas de flujo de una ejemplo de sentencia if.

Una alternativa a la sentencia if es el comando `find`²⁷, dada una matriz e identifica cuáles elementos en dicha matriz satisfacen un criterio dado y regresa los números índice de la matriz que satisfacen el criterio. Por ejemplo

```
x=[63, 67, 65, 72, 69];
y=find(x>=66);
```

El resultado de esta operación es:

```
a= 0 0 0 0 1
```

²⁷ Para más información de la función `find`: <http://www.mathworks.com/help/matlab/ref/find.html>

Sentencia SWITCH

La sentencia switch realiza una función análoga a un conjunto de if...elseif concatenados. Se usa con frecuencia cuando existe una serie de opciones de ruta para una variable dada, dependiendo de su valor. Su forma general es la siguiente:

```
switch variable
    case opcion1
        bloque1
    case opción2
        bloque2
    ...
    otherwise, % opción por defecto
        bloqueN
end
```

Al principio se evalúa la variable, cuyo resultado debe ser un número escalar o una cadena de caracteres. Este resultado se compara con las case opcion1, y se ejecuta el bloque de sentencias que corresponda con ese resultado y así sucesivamente con cada opción. Si ninguno caso cumple n se ejecutan las sentencias correspondientes a otherwise. El siguiente ejemplo muestra la aplicación de esta sentencia:

```
ponderacion=menu( 'Ingrese la ponderación que desea aplicar:' ,
'Ponderacion A' , 'Ponderacion C' , 'Ponderacion Z' );
switch ponderacion
    case 1
        disp( 'Ponderacion A' )
    case 2
        disp( 'Ponderacion C' )
    case 3
        disp( 'Ponderacion Z' )
    otherwise
        disp( 'No es una ponderación permitida' )
end
```

`input`²⁸ es una función que permite a los usuarios ingresar datos a la variable que la precede, se puede decir al comando `input` que espere una cadena al agregar 's' en un segundo campo.

La función `menu`²⁹ se usa con frecuencia en conjunto con una estructura `switch`. Esta función hace que aparezca un recuadro de menú en la pantalla, con una serie de botones definidos por el programador, como se observa en la siguiente imagen:

²⁸ Para más información de la función `input`: <http://www.mathworks.com/help/matlab/ref/input.html>

²⁹ Para más información de la función `menu`: <http://www.mathworks.com/help/matlab/ref/menu.html>

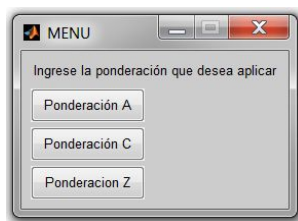


Figura 4.3 - Menú generado con la función `menu`.

Estructuras de repetición

Matlab soporta dos tipos diferentes de bucles: el bucle `for` y el bucle `while`. Los bucles `for` son la opción más sencilla cuando se sabe cuántas veces necesita repetir el bucle. Los bucles `while` son las opciones más sencillas cuando necesita mantener la repetición de las instrucciones hasta que se satisface un criterio.

A pesar de lo útiles que son estas estructuras, es conveniente evitar bucles, dado que los mismo consume mucho recursos. Sin embargo, hay técnicas que evitan los bucles, ya sea mediante el comando `find` o mediante la vectorización del código. (En la vectorización se opera sobre vectores enteros a la vez, en lugar de un elemento a la vez.) (Holly M, 1990).

Sentencia **FOR**

La primera línea identifica el bucle y define un índice, que es un número que cambia en cada paso a través del bucle. Después de la línea de identificación viene el grupo de comandos que se quiere ejecutar. Finalmente, la terminación del bucle se identifica mediante el comando `end`. En resumen, se tiene:

```
for i=1:n
    sentencias
end
```

o bien una matriz con los distintos valores que tomará la variable índice, en este caso i:

```
for i=[matriz]
    sentencias
end
```

En este caso el bucle se ejecuta una vez para cada elemento de la matriz índice identificada en la primera línea.

En el siguiente ejemplo se presenta la estructura más general para la variable índice del bucle (valor inicial : incremento : valor final); el bucle se ejecuta por primera vez con $i=n$, y luego i se va reduciendo de 0.2 en 0.2 hasta que llega a ser menor que 1, en cuyo caso el bucle se termina:

```
for i=n:-0.2:1
    sentencias
end
```

En el siguiente ejemplo se presenta una estructura correspondiente a dos bucles anidados. La variable *j* es la que varía más rápidamente (por cada valor de *i*, *j* toma todos sus posibles valores):

```
for i=1:m
    for j=1:n
        sentencias
    end
end
```

Por último, se utilizará un ciclo for para completar una matriz:

```
for k = 1:5
    a(k)=k^2;
end
```

Este bucle define una nueva matriz, *a*, un elemento a la vez. Dado que el programa repite su conjunto de instrucciones cinco veces, a la matriz *a* se agrega un nuevo elemento cada vez a lo largo del bucle con la siguiente salida en la ventana de comandos. Este procedimiento también es factible para recorrer una matriz.

Como se mencionó anteriormente una técnica para minimizar el uso de bucles es la vectorización de código, esto se resume en el siguiente ejemplo, que cumple con el mismo objetivo del último ejemplo presentado:

```
k=1:5
a=k.^2
```

De esta manera no solo se ahorra en la cantidad de líneas de código sino que además en recursos para ejecutar el bucle.

La siguiente imagen representa un diagrama de flujo de la sentencia for:

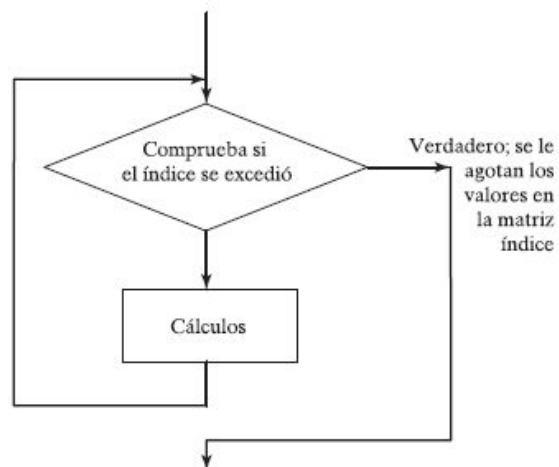


Figura 4.4 - Diagrama de flujo de un ciclo `for` (Holly M, 1990).

A continuación se presenta un ejemplo de la aplicación de sentencias `for`, el mismo fue extraídos de la referencia (Holly M, 1990).

Comenzamos con la creación de un archivo-m que permite crear una tabla que convierte ángulos en grados a radianes, para lo cual se utiliza la siguiente expresión:

$$radianes = (x) \frac{\pi}{180} \quad (4.1)$$

Donde x es el valor del ángulo en grados. En la figura 4.5 se presenta el diagrama de flujo que resuelve este problema. A continuación se presenta el código:

```
% Crear una tabla de grados a radianes
clear all, clc

% Inicialización del ciclo For
for k=1:36
    grados(k)=k*10;
    radianes(k)=grados(k)*pi/180;
end

% Crear una tabla
table=[grados;radianes]

% Presentar resultados
disp('Grados a radianes')
disp('Grados radianes')
fprintf('%8.0f %8.2f \n', table)
```

`fprintf`³⁰ es una función para mostrar un mensaje o un resultado, pero que permite mezclar texto y valores numéricos de las variables y también se puede ajustar el formato de los números.

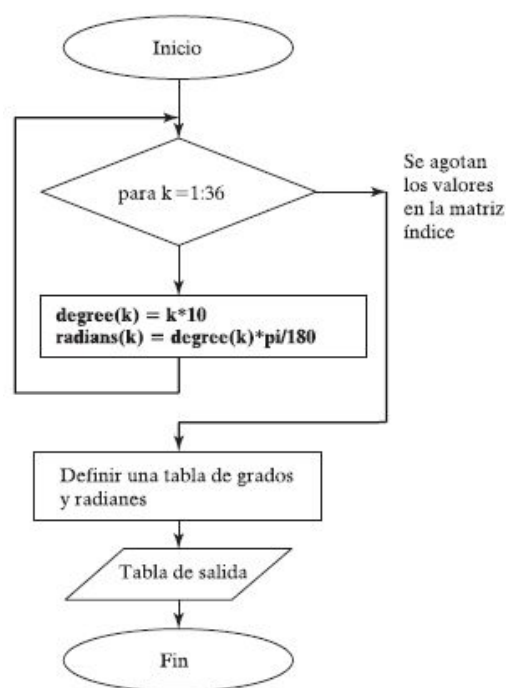


Figura 4.5 - Diagrama de flujo del código que permite convertir ángulo de grados a radianes (Holly M, 1990).

Por último, y con el objetivo de hacer hincapié en la utilización de la técnica de vectorización de código, se presenta la solución a este problema por medio de dicha técnica:

```

% Crear una tabla de grados a radianes
clear all, clc

% Vectorización del código
grados=0:10:360
radianes=grados*pi/180

% Crear una tabla
table=[grados;radianes]

% Presentar resultados
disp('Grados a radianes')
disp('Grados radianes')
fprintf('%8.0f %8.2f \n', table)
  
```

³⁰ Para más información de la función `fprintf`: <http://www.mathworks.com/help/matlab/ref/sprintf.html>

Sentencia *WHILE*

La estructura del bucle while es muy similar a la de C/C++/Java. Además son similares a los bucles for. La gran diferencia es la forma en que Matlab decide cuántas veces repetir el bucle. Los bucles while continúan hasta que se satisface algún criterio. El formato para un bucle while es:

```
for criterio
    sentencias
end
```

donde criterio puede ser una expresión vectorial o matricial. Las sentencias se siguen ejecutando mientras haya elementos distintos de cero en criterio, es decir, mientras haya algún o algunos elementos true. El bucle se termina cuando todos los elementos de criterio son false (es decir, cero). Por ejemplo:

```
k=0
while k<3
    k=k+1
end
```

En este caso se inicializa un contador, k, antes del bucle. Entonces el bucle se repite en tanto k sea menor que 3. En cada ocasión, k aumenta por 1 a lo largo del bucle, de modo que el bucle se repite tres veces. Una observación importante es que cualquier problema que se pueda resolver con un bucle while también se podría resolver con un bucle for (Holly M, 1990).

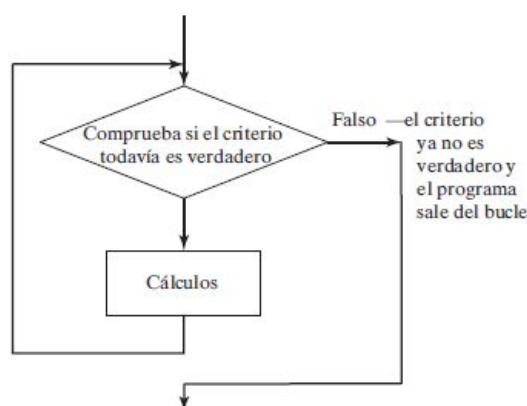


Figura 4.6 - Diagrama de flujo de la sentencia *while* (Holly M, 1990).

La variable que se usa para controlar el bucle while se debe actualizar cada vez que pase por el bucle. Si no, generará un bucle interminable. Cuando un cálculo toma demasiado tiempo para completarse, puede confirmar que la computadora realmente trabaja en él al

observar el indicador “busy”, en la esquina inferior izquierda. Para salir manualmente del cálculo, escriba **ctrl c**.

La figura 4.6, muestra el diagrama de flujo de la sentencia while.

Sentencia **BREAK** y **CONTINUE**

El comando break se puede usar para terminar un bucle prematuramente (mientras que la comparación en la primera línea todavía es verdadera). Un enunciado break provocará la terminación de la parte más pequeña que encierra un bucle while o for. He aquí un ejemplo:

```
n=0;
while (n<10)
    n=n+1;
    a=input( 'Ingrese un valor mayor que 0: ' );
    if (a<=0)
        disp( 'Debe ingresar un número positivo ' )
        disp( 'Este programa terminará' )
        break
    end
    disp( 'El logaritmo natural de este número es' )
    disp(log(a))
end
```

En este programa, el valor de n se inicializa a fuera del bucle. Cada vez que se pasa por el bucle, el comando de entrada se usa para pedir un número positivo. El número se comprueba y, si es cero o negativo, se envía un mensaje de error a la ventana de comandos y el programa salta del bucle. Si el valor de a es positivo, el programa continúa y ocurre otro paso a través del bucle hasta que n finalmente es mayor que 10.

El comando continue es similar a break; sin embargo, en lugar de terminar el bucle, el programa sólo salta al paso siguiente.

Comentario de fin de capítulo

Seguramente a esta altura ya se dio cuenta la gran diferencia entre la asignación (=) y la igualdad (==). Pero es importante que esto quede bien claro, dado que fácilmente uno caiga en el siguiente error:

$A = 1; B = 2;$

$A = A \times 2 + B \Rightarrow A - A \times 2 = B \Rightarrow A(1 - 2) = B \Rightarrow -A = B$

Lo cual no es cierto, bien este error reside en que el signo (=) no es un operador, es un signo que no puede ser tratado matemáticamente y simplemente asigna un valor a una variable.

Comandos, caracteres y funciones usadas en el capítulo

Tabla 4.5 - Comandos, caracteres y funciones vistos a lo largo del capítulo 4.

Elementos	Definiciones
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual que
~=	Distinto que
&	Operador lógico AND
	Operador lógico OR
xor (A,B)	Operador lógico XOR
~	Operador lógico NOT
disp ()	Permite mostrar mensaje en el command window
find()	Función que dada una matriz e identifica cuáles elementos en dicha matriz satisfacen un criterio dado y regresa los números índice de la matriz que satisfacen el criterio
input()	Es una función que permite a los usuarios ingresar datos, los cuales son tomados como cadena de caracteres.
sum()	Función se entrega la suma de todos los elementos que componen la matriz.

Capítulo 5 - Funciones

Introducción

La gran mayoría de los cálculos de ingeniería requieren funciones matemáticas muy complicadas, incluidos logaritmos, funciones trigonométricas y funciones de análisis estadístico. Matlab tiene una extensa librería de funciones internas que le permiten realizar dichos cálculos (Holly M, 1990).

En este capítulo busca ser una guía para que el lector pueda entender cómo utilizar las diferentes funciones que ofrece Matlab, así como brindar herramientas para lograr programar sus propias funciones.

¿Por qué utilizar las funciones que brinda Matlab? Esta respuesta es muy sencilla y obvia: Porque son pseudo-programas que alguien ya desarrolló, y que el usuario puede utilizar. Imagine si cada vez que se necesite calcular un coseno o un seno de un número, el programador tuviese que desarrollar una la serie de Taylor para aproximar este valor. Las funciones deben ser pensadas como herramientas para hacer algo. Un ejemplo sencillo y conciso: Las funciones de Matlab son como el clavo y el martillo. Alguién los creó y otro los utiliza. **No se inventa ni se fabrica su propio martillo cada vez que quiere arreglar algo.**

¿Para qué crear nuevas funciones? Esta respuesta es más amplia y hay varios motivos. El primero y principal es que el usuario no encuentra la función que desea dentro de las librerías de Matlab, por eso necesita desarrollar su propia función. Entonces ahí surge otra pregunta: ¿Cual es la diferencia entre desarrollar una función y escribir un código que realice la misma operación que necesito realizar? Bueno, la ventaja principal que tiene una función es que es reutilizable: una vez creada, puede llamarse varias veces en un mismo programa, sin necesidad de escribir reiteradas veces los mismo. Incluso una vez guardada puede utilizarse en varios programas. Otra ventaja es que aporta comodidad al tratar de interpretar un código escrito: No es lo mismo leer un código de 500 líneas que un código de 10 líneas que llame a varias funciones. El lector puede interpretar la estructura general de un programa en si y evaluar las funciones por separado de ser necesario inmiscuirse en los detalles. Un programa que se subdivide en tareas y cada una de ellas es llevada a cabo por una función, a esta forma de resolver los problemas se denomina **programación estructurada**. Una última ventaja, pero no la menos importante, tiene que ver con el rendimiento. **Al utilizarse una función, las variables que son definidas dentro de la misma, solo existen mientras se esté utilizando dicha función, obteniendo beneficios como ahorro de espacio en memoria y poder repetir nombres de variables que se encuentren en otras funciones o fuera de ellas. Estas variables son conocidas como variables locales, las cuales se diferencian de las variables globales, que al contrario, existen y ocupan espacio en memoria todo el tiempo mientras se ejecuta un programa.** Como este texto busca ser una guía para comprender y utilizar Matlab, y no un curso de programación, no se darán mayores detalles, pero se recomienda que sea un tema

a leer, principalmente a los interesados en aprender a programar en otros lenguajes como C, C++, Java, etc.

Funciones internas

Generalmente, las funciones tienen tres componentes: **Nombre**, **Entrada** y **Salida**. La forma de invocar y utilizar las funciones preestablecidas de Matlab es la siguiente:

$$Salida = Nombre(Entrada) \quad (5.1)$$

Donde *Entrada* (argumento de la función) puede ser una variable de cualquier tipo, dependiendo de la variable que solicite la función (Una función que sume dos números no va a funcionar con una variable tipo string). *Nombre* es el nombre de la función, como `sum`, `max`, `plot` por citar algunos ejemplos y *Salida* es la variable que devuelve la función.

También existen funciones que requieren múltiples entradas y/o múltiples salidas, siendo su formato el siguiente:

$$[Salida_1, Salida_2, Salida_3, \dots, Salida_n] = Nombre(Entrada_1, Entrada_2, Entrada_3, \dots, Entrada_n); \quad (5.2)$$

Donde las diferentes $Salida_i$ y $Entrada_i$ pueden ser diferentes tipos de variables independientemente entre sí, como se puede apreciar en el siguiente ejemplo:

```
% Ejemplo
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y, '--rs', 'LineWidth', 2);
```

En este caso, x e y son vectores, pero a la función `plot` también acepta strings como `'--rs'` y `'LineWidth'`, que sirven para especificar características como color y ancho de línea al realizar un gráfico (esta función y su uso será estudiada en mayor profundidad en el capítulo 5).

Muchos de los nombres de las funciones internas de Matlab son los mismos que los definidos no sólo en el lenguaje de programación C. Por ejemplo, para sacar la raíz cuadrada de la variable x , se escribe:

```
>> b = sqrt(x)
```

Una de las grandes ventajas de Matlab es que los argumentos de la función, por lo general, pueden ser escalares o matrices. En el ejemplo, si x es un escalar, se regresa un resultado escalar. Por tanto, el enunciado

```
>> x = 9;
>> b = sqrt(x)
```

Regresa un escalar:

```
b =
3
```

Pero si x es un vector, el resultado también lo será:

```
>> x = [4, 9, 16];
>> b = sqrt(x)

b =     2         3         4
```

Ayuda

Matlab incluye extensas herramientas de ayuda, lo que es especialmente útil para entender cómo usar las funciones. Existen dos formas de obtener ayuda desde el interior de Matlab: una función de ayuda de línea de comando (`help`) y un conjunto de documentos HTML disponibles al seleccionar Help de la barra de menú o al usar la tecla de función F1, que, por lo general, se ubica en la parte superior de su teclado (o que se encuentra al escribir `helpwin` en la ventana de comandos). También existe un conjunto de documentos de ayuda en línea, disponibles a través del botón Start o el icono Help en la barra de menú. Debe usar ambas opciones de ayuda, pues ellas ofrecen diferente información y pistas acerca de cómo usar una función específica. Para usar la función de ayuda de línea de comando, escriba `help` en la ventana de comandos (Holly M, 1990).

En la siguiente figura (figura 5.1) se muestran los resultados de introducir el comando `help` en la ventana de comandos:

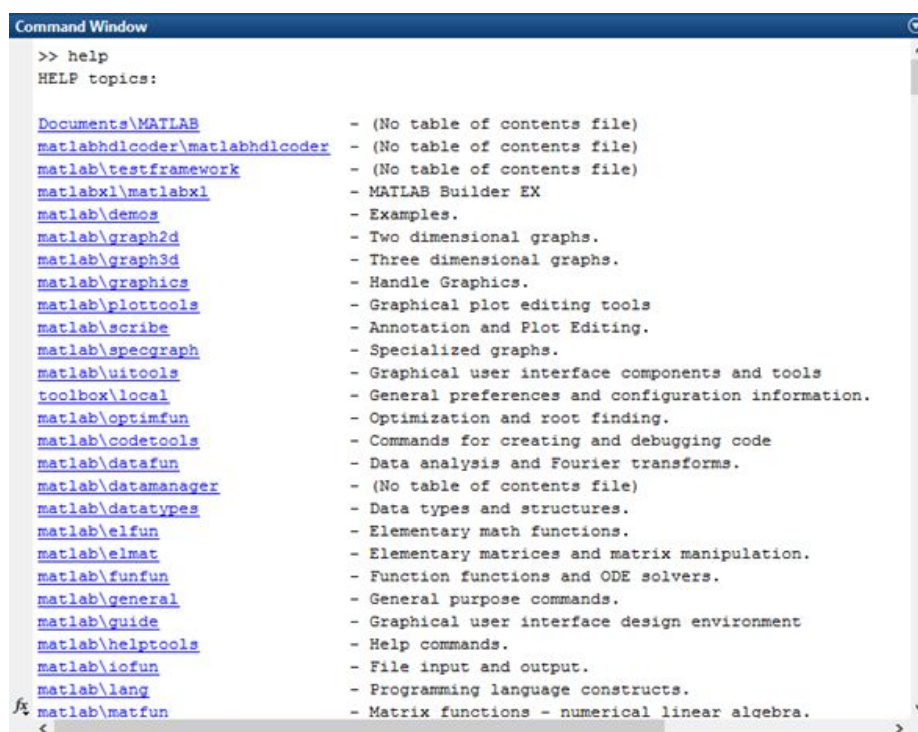
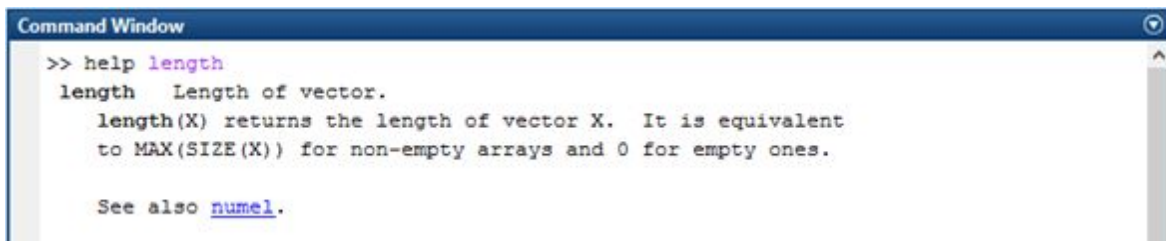


Figura 5.1 – Resultado de utilizar comando `help`

Como resultado de utilizar `help`, se obtiene una lista de tópicos, donde el usuario puede indagar y buscar la función que desee.

Otra forma de utilizar `help` es para conocer el funcionamiento de una función que ya conoce el nombre. Por ejemplo, si se desea saber cómo utilizar `length`, se escribe `help length` y se observa lo siguiente:



```

>> help length
length    Length of vector.
length(X) returns the length of vector X. It is equivalent
to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.

See also numel.
  
```

Figura 5.2 – Resultado de utilizar `help` para una función específica.

Para utilizar la ventana de ayuda, puede escribir `helpwin` en el Command Window, buscar “ayuda” o “help” en el menú, o apretar la tecla “F1” del teclado. Como resultado, aparecerá una ventana donde se puede buscar y obtener ayuda acerca de las diferentes librerías y sus funciones que abarca Matlab.

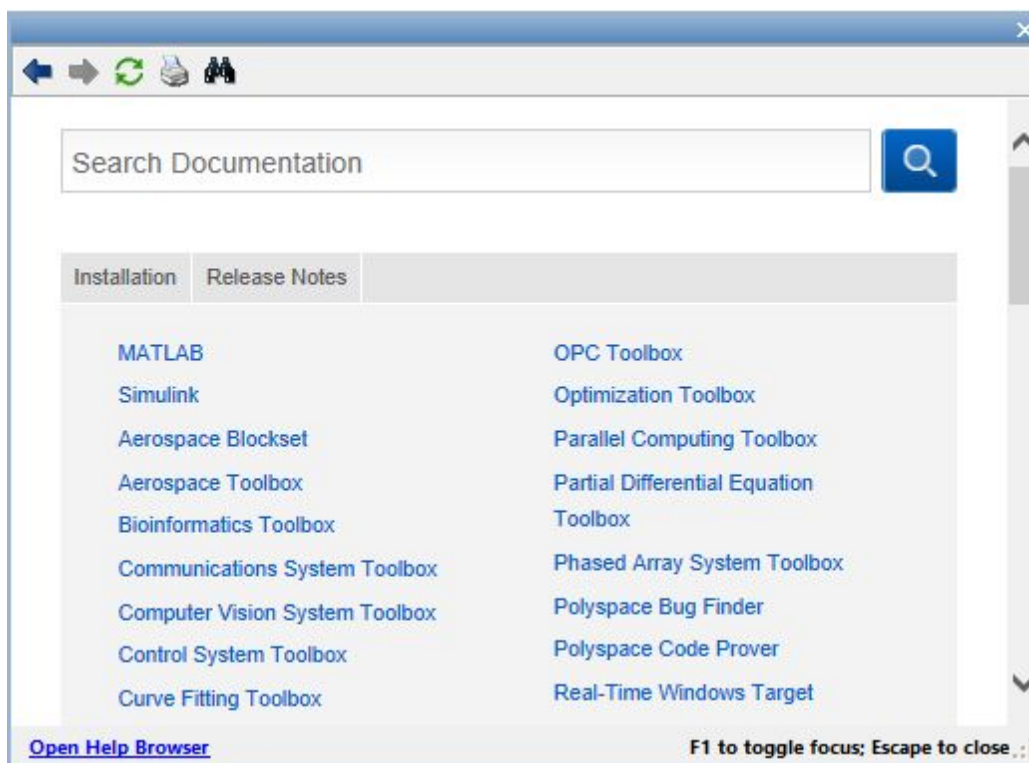


Figura 5.3 – Ventana de ayuda gráfica

Funciones más utilizadas

A continuación, en la tabla 5.1, se enumeran las funciones matemáticas más comunes. En el libro de Holly (Holly M, 1990), se pueden encontrar tablas más completas y mayores ejemplos.

Tabla 5.1 – Funciones matemáticas comunes (Holly M, 1990)

Función	Descripción	Ejemplo
<code>abs (x)</code>	Encuentra el valor absoluto de x	<code>abs (-3)</code> <code>ans = 3</code>
<code>sqrt (x)</code>	encuentra la raíz cuadrada de x	<code>sqrt (85)</code> <code>ans =</code> 9.2195
<code>nthroot (x,n)</code>	Encuentra la n-ésima raíz real de x. Esta función no regresará resultados complejos. Por tanto, $(-2)^{(1/3)}$ no regresa el mismo resultado, aunque ambas respuestas son legítimas raíces cúbicas de -2	<code>nthroot (-2, 3)</code> <code>ans =</code> -1.2599
<code>sign (x)</code>	Regresa un valor de -1 si x es menor que cero, un valor de 0 si x es igual a cero y un valor de 1 si x es mayor que cero	<code>sign (-8)</code> <code>ans = -1</code>
<code>rem (x, y)</code>	Calcula el residuo de x/y	<code>rem (25, 4)</code> <code>ans = 1</code>
<code>exp (x)</code>	Calcula el valor de e^x , donde e es la base para logaritmos naturales, o aproximadamente 2.7183	<code>exp (10)</code> <code>ans =</code> 2.20262+004
<code>log (x)</code>	Calcula $\ln(x)$, el logaritmo natural de x (a la base e)	<code>log (10)</code> <code>ans =</code> 2.3026
<code>log10 (x)</code>	Calcula $\log_{10}(x)$, el logaritmo común de x (a la base 10)	<code>log10 (10)</code> <code>ans = 1</code>

Otras funciones muy utilizadas son las que permiten al usuario de un programa interactuar con el mismo, ingresando datos o visualizando resultados. Por ejemplo, imagine un programa que sume números. Es muy útil que el usuario pueda ingresar esos valores como variables para que el programa realice la operación entre ambos y regrese un mensaje lindo y prolijo como “El resultado es: ”. El programador escribe un código del siguiente estilo:

```
% Sumador
```

```
var_1 = 1;
var_2 = 3;

Suma = var_1 + var_2;
```

Este código obviamente cumple su propósito a la perfección, pero solo es útil para el programador. Generalmente y casi siempre, el código de un programa no es modificado por el usuario. Imagine si cada vez que el usuario desee utilizar el programa tenga que cambiar los valores de `var_1` y `var_2`. Y este es un programa ultra sencillo, si tuviésemos algo mucho más complejo sería una locura.

Necesitamos que el usuario utilice el programa sin tocar el código del mismo, que interactúe a través de una interfaz gráfica de usuario (se verá en otro momento) o a través de una consola o ventana de comandos.

Para ello, Matlab dispone de dos funciones básicas de interacción, una es `input` y la otra es `disp`. Como se menciona al comienzo de este apunte, **programar se aprende al HACER**, por eso empezamos con un ejemplo, para que el lector copie, intérprete y razone su funcionamiento (y utilice la función `help`):

```
% Sumador

var_1 = input('Ingrese primer valor: ');
var_2 = input('Ingrese segundo valor: ');

Suma = var_1 + var_2;

disp(' ');
disp('El valor resultante es: ');
disp(Suma);
```

La función `input` requiere una variable tipo string como entrada, la cual contendrá el mensaje que se quiere que se visualice en la ventana de comandos. Al ejecutarse una línea de código que contenga esta función, el programa se pondrá en pausa, esperando por la entrada de información a través del teclado, la cual se almacenará en una variable, en este caso `var_1` y luego `var_2`.

La variable de salida de `input` se guarda como una matriz de dimensión $n \times n$, dependiendo de la información que ingrese el usuario por teclado. Si se desea almacenar la información introducida como una variable tipo string, el formato es el siguiente:

```
var = input('Mensaje', 's');
```

La función `disp` requiere de una única variable de entrada, y su función es mostrar el contenido de dicha variable en la ventana de comandos.

Otra función de la cual también dispone Matlab es `fprintf`, una función muy conocida para los programadores de C y C++ (`printf`). Cumple el cometido de la función `disp`,

salvo por el hecho que permite manipular con mayor detalle el mensaje a mostrar en la ventana de comandos, así como la cantidad de dígitos enteros y decimales a mostrar. Su funcionamiento es el siguiente:

```
fprintf(FORMAT, A, ...)
```

Donde `FORMAT` es una variable tipo string que contiene el mensaje a exponer y donde también se especifica la precisión de las variables, y `A` es una matriz con los valores a exponer.

Veamos un ejemplo:

```
% Ejemplo

v = [1.23, 43.0034, 45.2345, 56.5678, 234, 1000.01];

fprintf('Los valores expresados como punto flotante con 2
decimales son:\n');
fprintf('%8.2f \n', v);
```

Como se puede observar, la primera línea donde aparece `fprintf` produce el mensaje ingresado en la ventana de comandos, tal como se explicó con la función `disp`, salvo por un detalle: El string ingresado termina con `\n`. Este caracter indica un salto de línea. Pruebe borrarlo para ver las diferencias.

En la segunda línea donde aparece `fprintf`, se ingresa un string con el mensaje y formato, y el vector con datos a mostrar. Con `%8.2f` se está indicando que en ese lugar del mensaje a mostrar en pantalla, se van a utilizar 8 dígitos en total para representar el número, con dos decimales luego del punto, y la `f` es para indicar que es en punto flotante. Esta es la forma como se especifica el formato de salida.

A continuación, se presenta en la tabla 3.2 con los distintos formatos y caracteres para expresarlos:

Tabla 3.2 - Formatos de salida (Documento Matlab)

Tipo	Conversión	Detalle
Entero con signo	<code>%d</code> o <code>%i</code>	Base 10
Entero sin signo	<code>%u</code>	Base 10
	<code>%o</code>	Base 8
	<code>%x</code>	Base 16
	<code>%X</code>	Base 16 (Mayúscula)
Punto flotante	<code>%f</code>	Notación punto flotante

	%e	Notación Exponencial
	%E	Notación Exponencial (Mayúscula)
	%g	Más compacto que %e o %f sin ceros a la derecha
	%G	Idem %g, (%E o %f)
	%bx o %bX %bo %bu	Doble precisión
	%tx o %tX %to %tu	Precisión simple
Caracteres	%c	Caracter simple
	%s	Cadena de caracteres

Funciones definidas por el usuario

El lenguaje de programación Matlab se construye alrededor de funciones. Una función es una pieza de código de computación que acepta un argumento de entrada del usuario y produce salida al programa. Las funciones le ayudan a programar eficientemente, lo que le permite evitar reescribir el código de computación para cálculos que se realizan con frecuencia. Por ejemplo, la mayoría de los programas de cómputo contienen una función que calcula el seno de un número. En Matlab, `sin` es el nombre de la función que se usa para llamar una serie de comandos que realizan los cálculos necesarios. El usuario necesita proporcionar un ángulo y Matlab regresa un resultado (Holly M, 1990).

Como se vio en el comienzo de este capítulo, Matlab dispone de una extensa librería con funciones para utilizar. Pero... ¿Qué ocurre si necesitamos una función específica que no se encuentra en dichas librerías? Matlab permite que el usuario pueda programar una función, es decir, definir una función, guardarla y utilizarla como el resto de las que vienen incorporadas. Para ello, existe una sintaxis que debe ser respetada:

```
function salida = nombre(entrada)
% Línea de código
% Línea de código
% Línea de código
end
```

Esta es la forma con la cual se define una función en Matlab. Al abrir un nuevo m-file se debe comenzar con la palabra `function`, de esta forma se indica que se está definiendo una función. A continuación se procede a definir las variables de salida (**salida**) que tendrá dicha función. Puede ser un escalar, un vector, una matriz, un string, un cell array.

A continuación, se escribe el nombre (**nombre**) el cual se utilizará para invocar la función y, entre paréntesis, la o las variables de entrada de la función (**entrada**), las cuales pueden ser una sola, muchas, un vector, un string, etc. dependiendo del proceso computacional que realice nuestra función.

Para ilustrar, a continuación se presenta un ejemplo muy sencillo, de una función que suma dos valores:

```
function result = suma(numero1, numero2)
% Función que suma dos valores
result = numero1 + numero2;
end
```

Cómo se puede observar, el nombre de nuestra función es: `suma`, depende de las variables `numero1` y `numero2`, y la variable de salida es `result`, que tendrá guardado el resultado de sumar ambos valores ingresados.

Para que esta función sea utilizable, primero hay que guardarla en el directorio de trabajo donde se desea ser empleada. Es **MUY IMPORTANTE** guardar el archivo `.m` con el mismo nombre que el nombre de la función, sino el programa no correrá. En este caso, el nombre deberá ser “**suma.m**”.

También es importante recordar que el nombre de la función debe cumplir con:

- Comenzar con una letra (no puede empezar con un número, al igual que las variables).
- Puede contener letras, números y el guión bajo.
- No se pueden usar nombres reservados (como ‘max’, ‘cos’, etc.).
- Permite cualquier longitud, aunque los nombres largos no son una buena práctica en programación.

Cumpliendo lo mencionado, una vez guardada la función se puede proceder a utilizarse:

```
>> suma(3,6)
ans =

     9
```

Comentarios en las funciones

Como con cualquier programa de cómputo, debe comentar libremente su código de modo que sea más fácil de seguir. Sin embargo, en una función Matlab, los comentarios en la línea inmediatamente siguiente a la primera línea tienen un papel especial. Dichas líneas se recuperan cuando se solicita la función `help` en la ventana de comandos (Holly M, 1990). Si seguimos con nuestra función `suma`, al aplicar el comando mencionado en la ventana de comandos se obtiene (la current folder debe estar en la carpeta donde se ubica la función):

```
>> help suma
Función que suma dos valores
```

Funciones sin entrada o salida

Aunque la mayoría de las funciones necesitan al menos una entrada y regresan al menos un valor de salida, en algunas situaciones no se requieren ni entradas ni salidas. Por ejemplo, considere esta función, que dibuja una estrella en coordenadas polares:

```
function [] = star()
theta = pi/2:0.8*pi:4.8*pi;
r = ones(1, 6);
polar(theta, r);
end
```

Los corchetes de la primera línea indican que la salida de la función es una matriz vacía (es decir: no se regresa valor). Los paréntesis vacíos dicen que no se espera entrada. Si desde la ventana de comandos usted escribe `star`, entonces no regresa valores, sino que se abre una ventana de figura que muestra una estrella dibujada en coordenadas polares (Holly M, 1990), lo cual puede apreciarse en la figura 5.3.

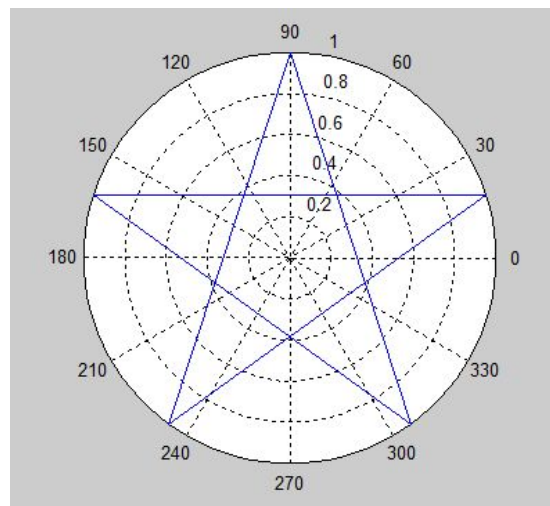


Figura 5.3 - Resultado de emplear la función `star`.

Funciones primarias y secundarias

Un archivo `.m` de una función definida por el usuario puede contener más de una función. Las funciones se definen una a continuación de la otra. La primera función es la primaria y tiene el mismo nombre que el fichero, las otras funciones son secundarias y se denominan subfunciones y pueden estar en cualquier orden dentro del fichero. Solamente se puede llamar a la función primaria en la ventana de comandos o por otras funciones. Las variables de cada función son locales. La ventaja de trabajar de esta manera mantiene el espíritu inicial, hacer un código más estructurado y legible, a partir de subdividir una función

compleja en elementos más sencillos. A Continuación se presenta un ejemplo “sencillo” de aplicación, con el objetivo de resolver una ecuación de segundo grado (ecu. 5.3 y 5.4) y comprobar sus propiedades (ecu. 5.5).

$$ax^2 + bx + c = 0 \quad (5.3)$$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad (5.4)$$

$$x_1 + x_2 = -\frac{b}{a} \quad x_1 \cdot x_2 = \frac{c}{a} \quad (5.5)$$

```
function [x1,x2,r1,r2] = raices_2grad(p)
% RAICES DE UNA ECUACION
% ECUACIONES DE SEGUNDO GRADO (ax^2+bx+c=0)
% Con p=[a b c];
    dis= sqrt(p(2)*p(2)-4*p(1)*p(3));
    x1=calcula_raiz1(p,dis);
    x2=calcula_raiz2(p,dis);
    r1=x1+x2;
    r2=x1*x2;
end
function raiz=calcula_raiz1(p,dis)
    raiz=(-p(2)+dis)/(2*p(1));
end
function raiz = calcula_raiz2(p,dis)
    raiz=(-p(2)-dis)/(2*p(1));
end
```

Este código permite calcular las raíces de una ecuación de segundo grado, como se observa a continuación:

```
>> [x1,x2,r1,r2]=raices_2grad([1 -1 -6])

x1 =
     3
x2 =
    -2
r1 =
     1
r2 =
    -6
```

Funciones específicas - Audio

Como ya se mencionó anteriormente, Matlab dispone de una extensa librería de funciones que pueden ser utilizadas por el programador. Dentro de esa librería, se encuentran algunas destinadas al uso de procesamiento de audio. En esta sección se estudiarán las funciones

que permitirán al lector obtener información de archivos de audio y obtener información de los dispositivos de audio analógicos-digitales asociados a una computadora.

Primero veamos cómo procesar información que se encuentra guardada en un archivo. El registro de una señal sonora puede guardarse en distintos formatos, como pueden ser **.wav**, **.flac**, **.mp3**, **.mp4**, **.m4a** o **.ogg**, y esta información se encuentra guardada en el disco duro o en un dispositivo de almacenamiento externo. Para procesar estos datos en Matlab, necesitamos ubicarlos en memoria, es decir, crear una variable, un vector o una matriz con dicha información.

Para ello, Matlab dispone de dos funciones: `audioread` y `wavread`. La primera está disponible desde la versión 2011 de Matlab y la segunda sigue vigente en las versiones más recientes, aunque al utilizarse aparecerá un mensaje de aviso en la ventana de comandos comunicando que pronto quedará obsoleta. Dada la actualidad de este apunte y las ventajas de `audioread` sobre `wavread`, solo estudiaremos la primer función, aunque cabe aclarar que ambas tienen una forma de funcionamiento muy similar. **La principal ventaja (como habrán deducido por el nombre) es que `wavread` solo es compatible con archivos en formato `.wav` mientras que `audioread` admite todos los formatos mencionados anteriormente.**

Comencemos con un pequeño ejemplo, confiando que el lector del apunte está haciendo uso de las herramientas de ayuda para entender el funcionamiento de estas funciones, mientras lee con mucho detenimiento:

```
%% Audioread

% Nombre del archivo
archivo = 'Globoledit.wav';

% Obtener información
[y, fs] = audioread(archivo);

% Vector tiempo para gráfico
t = (0:length(y)-1)/fs;

% Gráfico
plot(t, y);
title 'Archivo de Audio'
xlabel 'Tiempo [Seg]'
ylabel 'Amplitud normalizada'
grid on
```

Cómo se puede apreciar en el ejemplo, `audioread` tiene una variable de entrada, la cual es una variable tipo string que contiene la información del nombre y ubicación del archivo en la computadora. Esto es importante. En el caso del ejemplo, el archivo se encuentra en la carpeta de trabajo de Matlab, único lugar en el cual se buscará el archivo especificado. Si el archivo se encuentra localizado en cualquier otra carpeta, hay que especificar la ruta

completa del mismo, para evitar recibir un mensaje de error en la ventana de comandos. Por ejemplo:

```
archivo = 'C:\Users\Pepito\Señales&Sistemas\Globoedit.wav' ;
```

Con este dato, `audioread` devuelve dos variables, en el ejemplo nombradas como `y` y `fs`. La primera contiene la información de audio guardada en el archivo, mientras que la segunda contiene la cantidad de muestras por segundos (frecuencia de sampleo) con la que fue grabada, creado o con la cual debe reproducirse el archivo. Entonces `y` es una matriz, la cual contiene en cada columna la información correspondiente a cada canal de nuestro archivo y `fs` es simplemente un escalar.

Con lo visto hasta el momento, el lector de este apunte ya se encuentra en condiciones de cargar un archivo de audio en una matriz y procesar dicha información. Ahora, veamos la forma de reproducir dichos vectores, o de guardar el contenido de una variable en un archivo de audio. Para ello usaremos `sound` y `audiowrite` (Como en el caso anterior, esta última función también está disponible a partir de las versiones más recientes de Matlab y se utiliza de una forma muy similar a la próxima versión obsoleta: `wavwrite`).

Sigamos con nuestro ejemplo:

```
% Reproducir
sound(y, fs);

% Guardar
audiowrite('Hola.wav', y, fs);
```

La función `sound` es la que nos permitirá reproducir desde Matlab la información de una matriz, y sus variables de entrada son: matriz, frecuencia de sampleo y número de bits. Si estos últimos valores no son introducidos, la reproducción se realizará utilizando la información en `y` pero con los valores por defecto de: `fs = 8192` y `nBits = 8`. **Si se desea detener un sound, ingresar `clear sound` en command window**

La función `audiowrite` es la que nos permite crear un archivo (en este caso `.wav`) y sus variables de entrada son: Nombre del archivo, matriz con información y frecuencia de sampleo (También permite guardar algún comentario en la información del archivo, pero no viene al caso). Es importante, como en el caso anterior, especificar toda la ruta del nombre el archivo. Ingresando solamente el nombre, este se guardará en la carpeta de trabajo.

Cómo el ejemplo anterior no es lo suficientemente útil, ya que solo lee la información de un archivo `.wav`, reproduce el contenido y lo vuelve a guardar con el nombre `'Hola.wav'`, a continuación se añade otro ejemplo:

```
%% Ejemplo

% Nombre del archivo
archivo = 'Globoedit.wav' ;
```

```

% Obtener información
[y, fs] = audioread(archivo);

% Vector tiempo
t = (0:length(y)-1)/fs;

% Procesamiento
x = y(1:fix(length(y)/2))/2;

% Vector tiempo
t2 = (0:length(x)-1)/fs;

% Gráficos
subplot(2,1,1)
plot(t, y);
title 'Archivo de audio'
xlabel 'Tiempo [Seg]'
ylabel 'Amplitud normalizada'
subplot(2,1,2)
plot(t2, x);
title 'Audio recortado y con menor nivel'
xlabel 'Tiempo [Seg]'
ylabel 'Amplitud normalizada'

% Guardar
audiowrite('Modificacion.wav', x, fs);

```

En este ejemplo, se busca “recortar” el audio y modificar el nivel de escucha, una práctica muy común que se realiza con editores de audio como plataformas multipistas. Para ello, se crea el vector x que contiene la información de y desde la muestra uno hasta la mitad del valor del total de las muestras, y se divide la amplitud de cada una a la mitad. **La función `fix` redondea el valor resultante de dividir el total de muestras a la mitad, debido a que, si el mismo es impar, el resultado será un número con coma, y no existe la muestra “10.5”. Las muestras son siempre números enteros.** Luego se grafican los resultados para ver los resultados, y se guarda el resultado procesado como `Modificacion.wav`. En la figura 5.4 se observan los resultados obtenidos sobre `Globoedit.wav`:

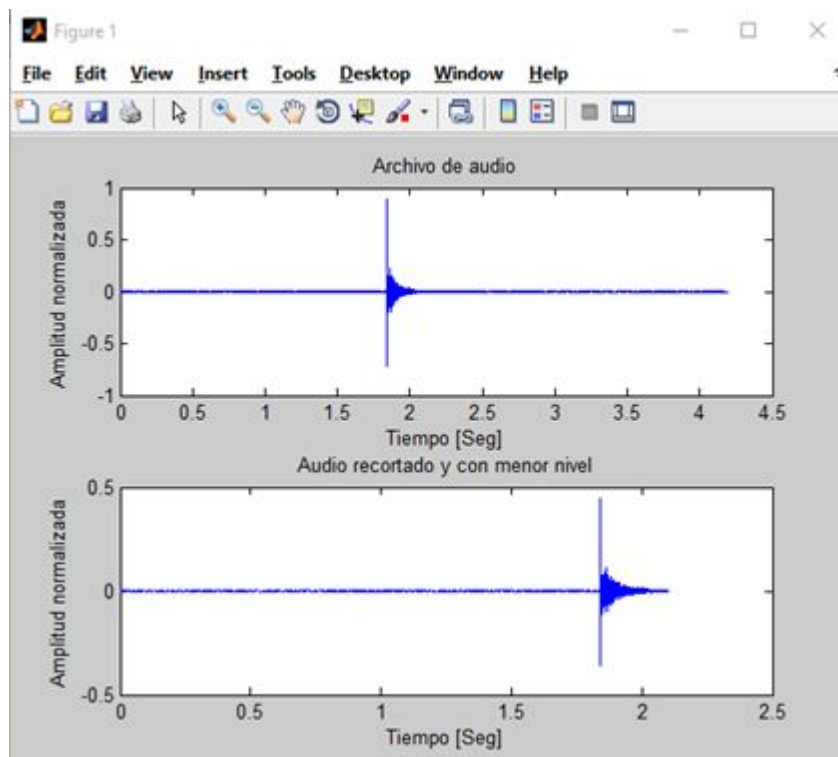


Figura 5.4 - Resultado de procesar el registro de una señal de audio

Como se puede observar en los nombres del eje “y”, la amplitud está normalizada. ¿Que quiere decir esto? La información que tenemos de amplitud no tiene ninguna relación con alguna cantidad o unidad física conocida (por lo menos para este caso). El vector resultante de utilizar `audioread` es un vector cuyos valores están acotados entre -1 y 1 (dependiendo del tipo de datos que se utiliza para representar las muestras del archivo. -1 y 1 corresponden a 32 bits por muestra, tipo single, en este texto siempre se referirá a este formato. Los distintos formatos y rangos se pueden observar en la tabla 5.3).

Tabla 5.3 - Formato de archivo, Bits por muestra, tipos de data y rangos (Documento Matlab)

Formato de Archivo	Bits por muestra	Tipo de datos	Rango
WAVE (.wav)	8	uint8	$0 \leq Y \leq 255$
	16	int16	$-32768 \leq Y \leq 32767$
	24	int32	$-2^{32} \leq Y \leq 2^{32}-1$
	32	int32	$-2^{32} \leq Y \leq 2^{32}-1$
	32	single	$-1.0 \leq Y \leq +1.0$
FLAC (.flac)	8	uint8	$0 \leq Y \leq 255$
	16	int16	$-32768 \leq Y \leq 32767$
	24	int32	$-2^{32} \leq Y \leq 2^{32}-1$
MP3 (.mp3) MPEG-4 (.m4a, .mp4)	N/A	single	$-1.0 \leq Y \leq +1.0$

OGG (.ogg)			
------------	--	--	--

Estos valores representan los valores máximos de los conversores analogico-digitales empleados o a emplearse. ¿Cómo es esto? Nada mejor que un ejemplo ilustrativo. Intente abrir un archivo de audio como en los ejemplos vistos, y aumente la amplitud para que las muestras superen el valor unitario (en vez de dividir por 2, multiplique por un valor conveniente, mientras más muestras superen dicho valor mejor servirá el ejemplo) y utilice la función `audiowrite` para guardar el archivo (tipo single). Luego, utilice un reproductor para escuchar. ¿Qué ocurrió? Dos cosas: Primero escuchó el efecto del saturador que acaba de crear, y segundo, aparece este mensaje en la ventana de comandos:

Warning: Data clipped when writing file.

> In audiowrite>clipInputData at 389

In audiowrite at 167

In Ejemplo at 31

Matlab no guarda valores que superan en módulo al valor 1. Esos son los límites. Si ocurre que, como en este caso, se intentase, los valores que superan la unidad son reemplazados con valores unitarios. Por ejemplo, una muestra con valor 2 se guarda como si fuese 1. Un -1 o un 1 son tomados como el valor máximo de tensión eléctrica límite de salida del conversor digital-analogico. Eso también quiere decir, que el nivel de escucha físico (presión sonora) está condicionado por el conversor y su sistema de sonido. Si el lector pudiera cambiar el conversor de su computadora, es posible que el valor de tensión eléctrica cambie, modificando el nivel de presión sonora resultante. En la figura 5.5 se ejemplifica el efecto de utilizar la función `audiowrite` con una matriz la cual contiene valores que superan los límites establecidos.

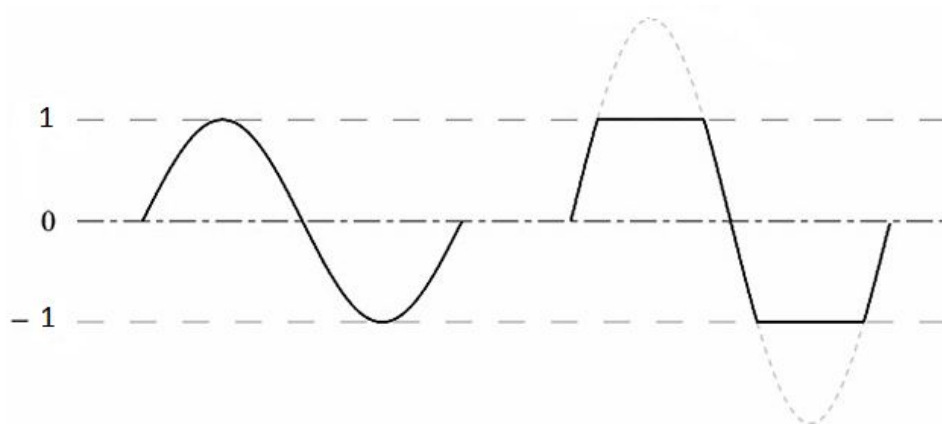


Figura 5.5 - Resultado de utilizar `audiowrite` cuando se supera el rango establecido

Lo mismo ocurre a la hora de grabar audio. Los niveles están limitados por el conversor analogico-digital y eso se refleja cuando se escribe un archivo. Si se sobrepasa el nivel de tensión eléctrica que admite el conversor, el resultado será el valor digital máximo del mismo, para cada valor de tensión superado. Entonces, cuando se obtenga un vector en Matlab con la información del archivo resultante, veremos a la señal “recortada” en amplitud, sin superar nunca, en módulo, el valor 1. En la figura 5.6 se especifica cómo se modifica una

señal tipo senoidal cuando se superan los límites máximos de tensión de entrada de un conversor.

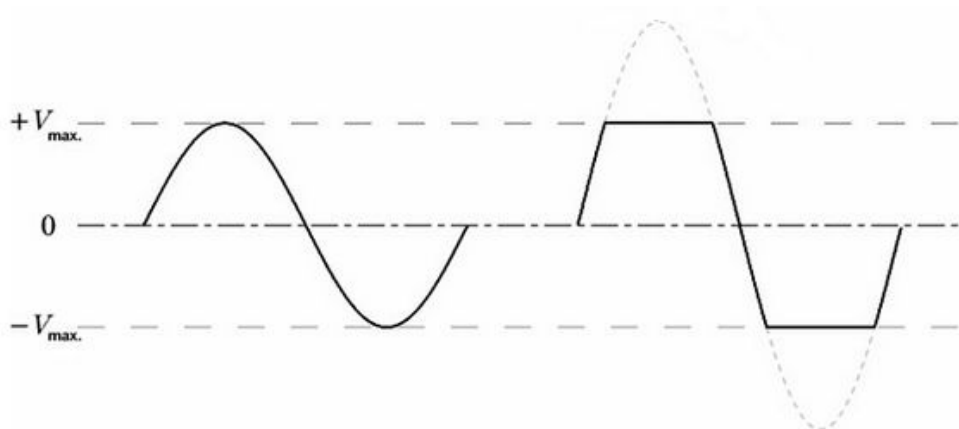


Figura 5.6 - Recorte de una señal cuando se superan valores máximos de un conversor analógico-digital

Es por esto que no se puede saber el valor que representan las muestras guardadas, ni encontrar la relación valor-presión sonora, porque el valor de cada muestra depende del sistema de sonido con el que se realizó el registro del mencionado fenómeno físico.

El único caso, es cuando se conocen todas las condiciones del sistema, como nivel de presión sonora, relación presión-tensión del transductor, configuración del amplificador, y relación tensión-valor digital del conversor. Otra opción (más accesible y utilizable) es tener el registro de una señal que sirva para calibrar el sistema. ¿Cómo? **Si se tuviese un dispositivo capaz de entregar un nivel de presión sonora conocido (94 dB SPL, por ejemplo), y se pudiera realizar un registro con nuestro sistema de dicha señal y observar los valores digitales obtenidos, se podría encontrar la relación presión sonora-valor digital tan deseada, y conocer que representan físicamente los valores de cada muestra obtenida.**

De esta forma, es como se realiza un instrumental de medición digital. Si la configuración no se ve modificada por ningún motivo, y se cuenta con un dispositivo calibrador (dispositivo que produce una magnitud física conocida) que tampoco se ve modificado, es posible establecer la relación analógica-digital con precisión.

Con lo mencionado anteriormente, solo faltaría conocer la forma de adquirir datos del conversor analógico-digital (placa de audio) desde Matlab para almacenar los valores entregados en memoria. Para ello, existe la función `audiorecorder` y otras más, que serán estudiadas a continuación:

Audiorecorder

```
r = audiorecorder(Fs, NBITS, NCHANS, ID)
```

Las variables de entrada que acepta la función `audiorecorder` son `Fs` la cual especifica en un escalar la frecuencia de muestreo, `NBITS` donde se ingresa la resolución, `NCHANS`

donde se indica la cantidad de canales de grabación (máximo dos) y `ID`, la cual va a almacenar un número identificador del dispositivo de audio a utilizarse.

Al correr esta línea de código no se inicializa el proceso de grabación, solamente se crea `r`, un objeto de grabación con la información necesaria. Pero vamos por pasos.

Si no se especifica `ID`, Matlab utiliza la configuración por default, la cual es utilizar la placa onboard de la computadora. Si se tiene otro dispositivo conectado y se desea grabar desde el mismo, hay que utilizar la función `audiodevinfo`, la cual otorgará la información de todos los dispositivos de audio que Matlab detecte conectados a la computadora. Entonces, ejecutamos en la ventana de comandos lo siguiente:

```
>> Info = audiodevinfo
```

Como resultado, se obtiene:

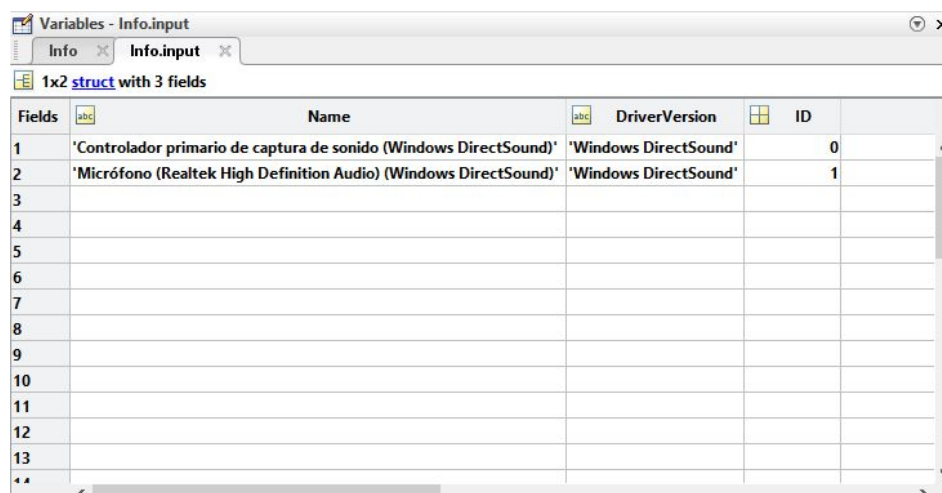
```
>> Info = audiodevinfo

Info =

    input: [1x2 struct]
   output: [1x2 struct]
```

`Info` es una estructura con dos campos, `input` y `output`, las cuales contienen los nombres de todas las entradas y salidas de audio, el driver que utilizan y el `Id` con el cual Matlab las identifica, el cual necesitamos para especificar de donde queremos tomar datos para grabar.

Entonces, se procede a abrir la ventana “Variables” (O como se llame según la versión de Matlab que el lector esté utilizando) haciendo doble click sobre la variable `Info` en la ventana “Workspace”. Una vez hecho esto, tendrían que aparecer los campos `input` y `output`. Se procede a realizar doble click nuevamente sobre `input` para desplegar la información que contiene dicho campo. Como resultado, el lector tendría que poder observar con detalle las diferentes entradas, sus controladores y el `ID` de cada una, cómo se ejemplifica en la figura 5.7:



Fields	Name	DriverVersion	ID
1	'Controlador primario de captura de sonido (Windows DirectSound)'	'Windows DirectSound'	0
2	'Micrófono (Realtek High Definition Audio) (Windows DirectSound)'	'Windows DirectSound'	1
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

Figura 5.7 - Información de todas las entradas de audio.

Otra forma de acceder a esta información, es ingresado en la ventana de comandos:

```
>> Info.input(1)
```

De esta forma se indica que se accede a la estructura `Info` y al primer componente del campo `Input`:

```
>> Info.input(1)

ans =

        Name: 'Controlador primario de captura de sonido
(Windows DirectSound)'
    DriverVersion: 'Windows DirectSound'
           ID: 0
```

Con esta información, se puede seleccionar de donde se desea realizar el proceso de grabación. Nuevamente, se hará uso de un ejemplo para mostrar cómo se adquieren datos de una placa de audio, y luego se explicará el mismo con mayores detalles:

```
%% Ejemplo

% Variables:
Fs = 44100;
NBITS = 16;
NCHANS = 1;
ID = 1;
Tiempo = 5; % En segundos

% Creación de objeto para grabar
r = audiorecorder(Fs, NBITS, NCHANS, ID);

% Inicio proceso de grabación
record(r);
pause(Tiempo);

% Adquirir datos
y = getaudiodata(r);

% Gráfico
figure(1)
plot((0:length(y)-1)/Fs, y)
title 'Adquisición de datos'
xlabel 'Tiempo [Seg]'
ylabel 'Amplitud'
grid on
```

El ejemplo anterior inicia el proceso de grabación desde el dispositivo que Matlab identifica con ID = 1, captura datos durante cinco segundos y almacena esa información en el vector `y`. La función `getaudiodata` es la encargada de obtener la información de audio del objeto obtenido mediante `audiorecorder`. Como resultado de ejecutar este código, se genera la siguiente gráfica (dependiendo de la señal que se capture) mostrada en la figura 5.8.

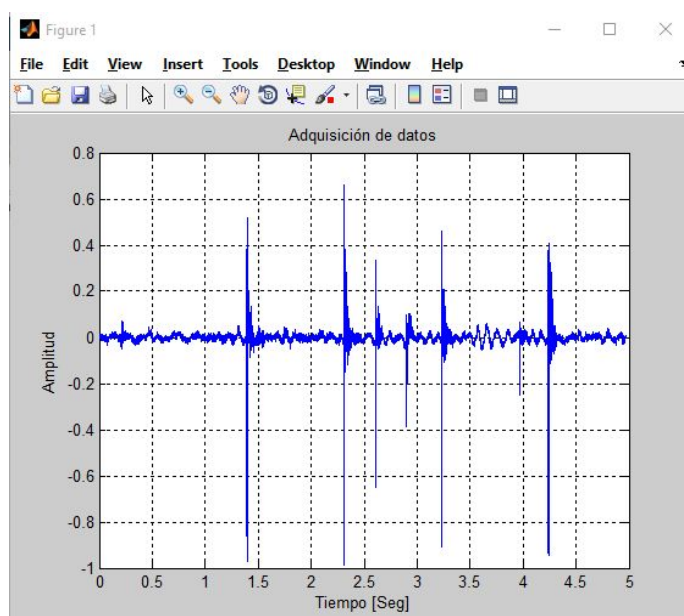


Figura 5.8 - Resultado de adquirir una señal sonora mediante MATLAB

Como no se especifica el formato, por defecto se almacenan como `double`, es por eso que ninguna muestra supera el valor uno. El lector puede cambiar la línea

```
% Adquirir datos
y = getaudiodata(r);
```

por cualquiera de las siguientes, para corroborar lo explicado anteriormente sobre los diferentes rangos y cómo varían los valores límites según el formato del array:

```
% Adquirir datos
y = getaudiodata(r, 'uint8');
y = getaudiodata(r, 'int8');
y = getaudiodata(r, 'int16');
y = getaudiodata(r, 'single');
y = getaudiodata(r, 'double');
```

Comandos, caracteres y funciones usadas en el capítulo

Tabla 5.4 - Comandos, caracteres y funciones usados en el capítulo 3.

Elemento	Definiciones
tan	Función que devuelve la tangente de una matriz
cos	Función que devuelve el coseno de una matriz
sen	Función que devuelve el seno de una matriz
help	Invoca la ayuda
plot	Realiza una gráfica con la información ingresada (ver cap. 5)
polar	Realiza una gráfica en coordenadas polares (ver cap. 5)
input	Permite el ingreso de datos por teclado
audioread	Permite obtener información de un archivo de audio
sound	Reproduce el contenido de una matriz con información de audio
audiowrite	Guarda la información de una matriz en un archivo de audio
audiorecorder	Crea un objeto para grabar
record	Inicia el proceso de grabación
pause	Pausa el programa
audiodevinfo	Brinda información de periféricos como entradas y salidas de audio
getaudiodata	Permite obtener la información de un objeto de audio en una matriz

Comentarios finales

A continuación presentamos una variedad de recomendaciones a la hora de programar en esta plataforma, las mismas son extraídas de la siguiente referencia (García de Jalón J., Rodríguez I. J, Vidal J., 2005):

Las funciones vectoriales de Matlab son mucho más rápidas que los escalares. En la medida de lo posible es muy interesante vectorizar los algoritmos de cálculo, es decir, realizarlos con vectores y matrices, y no con variables escalares dentro de bucles. Aunque los vectores y matrices pueden ir creciendo a medida que se necesita, es mucho más rápido reservarlos toda la memoria necesaria al comienzo del programa. Se puede utilizar para ello la función como `ser zeros` y `ones`. Además de este modo la memoria reservada es contigua³¹. Es importante utilizar el `profile` para conocer en qué sentencias de cada función se gasta la mayor parte del tiempo de cálculo. De esta forma se descubren “cuellos de botella” y se pueden desarrollar aplicaciones mucho más eficientes. Conviene desarrollar los programas incrementalmente, comprobando cada función o componente que se añade. De esta forma siempre se construye sobre algo que ya ha sido comprobado y que funciona: si aparece algún error, lo más probable es que se deba a lo último que se ha añadido, y de esta manera la búsqueda de errores está acotada y es mucho más sencilla. El debugger es una herramienta muy útil a la hora de acortar el tiempo de evaluación del código. Otro objetivo de la programación debe ser mantener el código lo más sencillo y ordenado posible. Al pensar en cómo hacer un programa o en cómo realizar determinada tarea es conveniente pensar siempre primero en la solución más sencilla, y luego plantearse otras cuestiones como la eficiencia. Finalmente, el código debe ser escrito de una manera clara y ordenada, introduciendo comentarios, utilizando líneas en blanco para separar las distintas partes del programa, sangrando las líneas para ver claramente el rango de las bifurcaciones y bucles, utilizando nombres de variables que recuerden al significado de la magnitud física correspondientes, etc. Este primer volumen finaliza dejando al lector algunos toolbox de acústica, ultrasonido y procesamiento de audio muy potentes. Con los conocimientos adquiridos hasta el momento, es posible aprovechar casi en su totalidad esta herramienta open source.

- [K-wave](#)
- [ITA](#)
- [Procesamiento de audio en tiempo real](#)
- [Procesamiento de audio con hardware](#)

Toolbox de interés

- [Data Acquisition Toolbox](#)
- [Signal Processing Toolbox](#)
- [DSP System Toolbox](#)

³¹ Una explicación más exhaustiva de esto se encuentra en el siguiente link:

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_din%C3%A1mico_de_memoria. Este concepto se aplica de igual manera en Matlab.

Referencias

- Moore H. (1990), **“MATLAB para ingenieros”**, 1ed. Pearson Prentice Hall.
- McLoughlin I.(2009), **“Applied speech and audio processing”**, 1ed. Cambridge University Press.
- Möser M., Barros J.L (2009), **“Ingeniería acustica - Teoría y aplicaciones”**, 2ed. Springer.
- García de Jalón J., Rodríguez I. J, Vidal J. (2005), **“Aprenda Matlab 7.0 como si estuviera en primero”**, Escuela Técnica Superior de Ingenieros Industriales, Universidad Politécnica de Madrid, Madrid, España.
- Floyd T. (2006), **“Fundamentos de sistemas digitales”**, 9ed. Pearson Prentice Hall.
- Etter D. M. (1998), **“Soluciones de problemas de ingeniería con Matlab”**, A Simon & Schuster Company.
- Chaparro L. F. (2011), **“Signals and Systems. Using Matlab”**, Elsevier.

Páginas Web de interés

- Página Web en español con una extensa explicaciones de las principales herramientas de Matlab y aplicaciones:

<http://www.sc.ehu.es/sbweb/energias-renovables/MATLAB/intro.html>