Introducción al uso de MATLAB y Octave

1. Introducción

En esta parte introduciremos los elementos esenciales de MATLAB® y GNU Octave (en adelante, normalmente diremos solo Octave), que son don entornos de programación matemática. Cuando digamos MATLAB® u Octave, podremos referirnos tanto al programa informático en sí (al *software*), como al lenguaje de programación que utiliza internamente cada uno de estos programas. Como lenguajes de programación, MATLAB® y Octave son esencialmente iguales, con algunos pequeños matices que se mencionarán cuando sea necesario. Salvo que se indique lo contrario, todo el código mostrado será válido tanto para MATLAB® como para Octave. Para la versión final de este documento se han utilizado las versiones R2024a (24.1.0.2689473) de MATLAB® , y 9.3.0 de GNU Octave, ambas para Windows 10 de 64 bits.

MATLAB® (en origen, abreviatura de *MATrix LABoratory*, "laboratorio de matrices") es un sistema informático de cálculo numérico que ofrece un entorno de desarrollo integrado (*IDE*, del inglés, *integrated development environment*), y desarrollado por MathWorks®, una compañía multinacional con sede principal en Natick, Massachusetts (Estados Unidos de América). Por tanto, se trata de un software propietario o privativo.

GNU Octave u Octave es usualmente considerado como la versión libre de MATLAB®, y, como su nombre indica, es parte del proyecto GNU de software libre (más información sobre el proyecto GNU en: `https://www.gnu.org/`). El software Octave también proporciona un IDE, algo más sencillo que el de MATLAB® pero con la misma funcionalidad básica.

Ambos entornos y lenguajes de programación son muy usados en el ámbito científico y de la ingeniería, y especialmente en universidades y centros de investigación y desarrollo. Recientemente han aumentado su popularidad con el auge de las técnicas de inteligencia artificial y *machine learning*. De acuerdo con MathWorks®, en 2023 había más de 5 millones de usuarios de MATLAB® a nivel mundial[1], siendo usado por 100.000 empresas, universidades y entidades gubernamentales, de las que más de 6.500 son universidades o facultades.

Para el propósito de este manual, podemos considerar que ambos entornos y lenguajes, MATLAB y Octave, son equivalentes. El código escrito en uno será directamente, o con muy leves modificaciones, ejecutable en el otro, y los resultados producidos, tanto gráficos como numéricos, prácticamente idénticos.

2. Instalación de MATLAB® y Octave

El primer requisito para utilizar código de MATLAB® o de Octave, es tener alguno de estos programas instalados en el ordenador. En realidad, esto no es estrictamente necesario; se podría usar alguna versión en la nube de los mismos como `https://matlab.mathworks.com/` o `https://octave-online.net/`, pero, por sus beneficios, recomendamos usarlos de forma local con el programa correspondiente instalado en nuestro ordenador siempre que sea posible.

[1] Según el *factsheet* publicado en: `https://www.mathworks.com/content/dam/mathworks/fact-sheet/2023-company-factsheet-8-5x11-8282v23.pdf`

La instalación de MATLAB® requiere de una licencia, y la mayoría de universidades proporcionan licencias de uso académico a sus docentes o estudiantes. Para obtener el software o una licencia, así como las instrucciones de descarga e instalación, recomendamos consultar en la página web de MathWorks® la sección del producto MATLAB®: `https://es.mathworks.com/products/matlab.html`, o bien consultar al soporte técnico de su universidad o centro de trabajo, cuando esto sea posible.

Como alternativa, también es posible utilizar MATLAB® directamente desde un navegador web a través del portal `https://matlab.mathworks.com/`, con una interfaz y utilización muy similares a las del software MATLAB® instalado localmente, aunque esa aplicación web también requiere disponer de algún tipo de licencia.

Para la instalación de Octave, que al tratarse de software libre no requiere disponer de ninguna licencia de pago, recomendamos consultar su página web: `https://www.gnu.org/software/octave/`, donde se encuentran las instrucciones y los archivos necesarios para la descarga e instalación en cualquier sistema operativo

Se pueden encontrar manuales muy completos para el uso de MATLAB® en la página web de MathWorks®: `https://es.mathworks.com/help/matlab/`, y también para Octave en la web: `https://docs.octave.org/latest/`. La mayoría de la información de estos manuales también está disponible dentro de la propia ayuda y de la documentación accesible desde ambos entornos de programación cuando se tienen instalados localmente.

3. Interfaz gráfica de MATLAB®

Comenzamos explicando los elementos del entorno gráfico de MATLAB®. Como complemento (o alternativa) a la explicación introductoria que hay a continuación, se ha elaborado un vídeo explicativo sobre los elementos fundamentales del programa informático MATLAB, que está disponible para su consulta en la web: `https://youtu.be/MwOYPVWDv-Q`

Al abrir el programa MATLAB, se muestra una ventana principal que está dividida en varias secciones. La Figura 1.1 ilustra una disposición inicial típica al abrir MATLAB. La colocación de estas partes puede cambiar según la versión o la personalización previa por parte del usuario, pudiendo estas algunas de ellas minimizadas u ocultas, y aparecer otras que no están visibles en la citada imagen.

En el ejemplo de la Figura 1.1 aparecen varias partes diferenciadas: el menú de herramientas superior 1 , la barra para configuración del directorio de trabajo 2, la sección *Current folder* con el contenido del directorio de trabajo 3, la ventana de comandos (*Command window*) 4 y la sección *Workspace* donde aparecerán las distintas variables que se vayan utilizado 5. El uso de la interfaz gráfica de MATLAB® es bastante intuitivo, y a continuación explicamos brevemente el propósito de cada una de estas partes:

1 **Menú de herramientas**: se muestran todas las acciones que el programa permite realizar, agrupadas en pestañas (*Home, Plots, Apps*, ...). Es una barra dinámica, cambiará según las ventanas que se tengan abiertas, o la que esté activa en un momento determinado. Dos herramientas a destacar en este punto son: *Layout*, para configurar qué ventanas se muestran y cómo se colocan (desde aquí se puede volver a mostrar alguna ventana que fuera cerrada previamente), y *Preferences*, con la configuración general y personalización del programa y del entorno gráfico (tipos de letra, comportamiento, colores, etc). Ambas

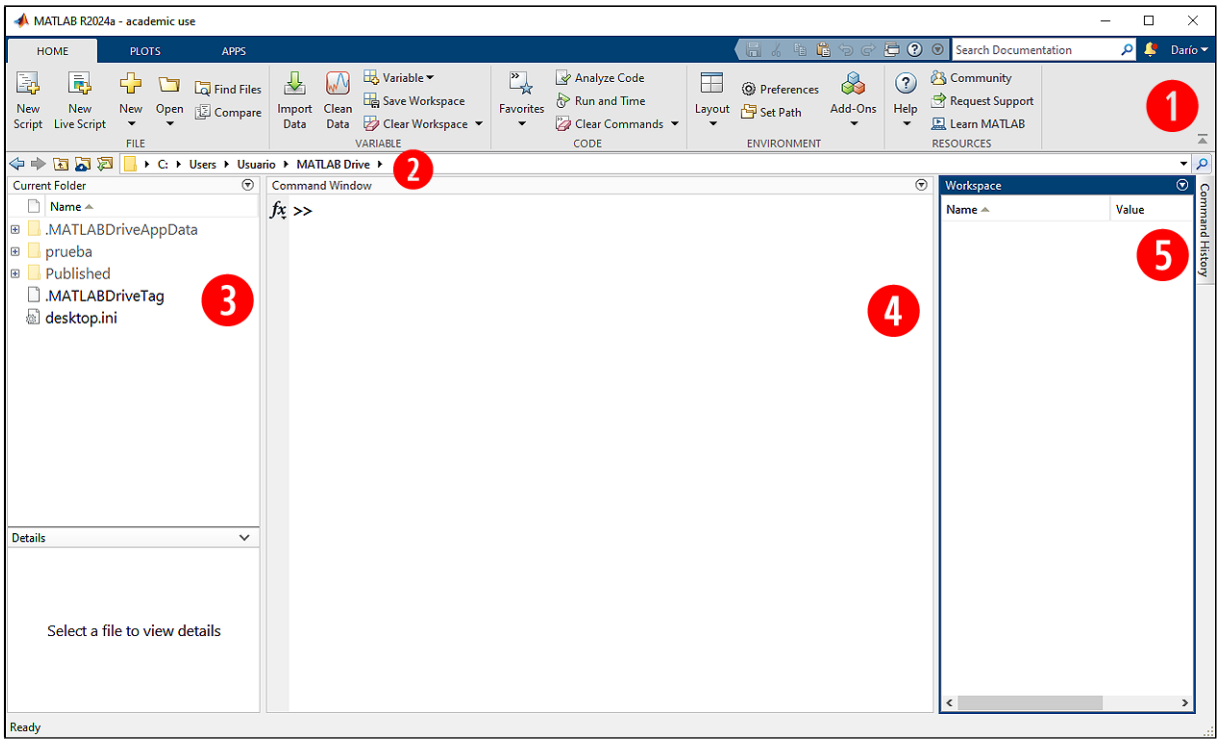

Figura 1.1: Entorno gráfico de MATLAB®, dividido en varias secciones.

herramientas, *Layout* y *Preferences*, están dentro del bloque *Environment* de la pestaña *Home*. También será de interés la ayuda y demás documentación, accesibles en *Help*, dentro del bloque *Resources* en la misma pestaña *Home*.

2 **Directorio de trabajo**: en esta barra se muestra el directorio activo o de trabajo, y permite modificarlo mediante los botones de la izquierda. Es la carpeta donde MATLAB® buscará los archivos necesarios al ejecutar cualquier programa. Si los archivos no se encuentran en esta carpeta, es probable que la ejecución falle. Por defecto, MATLAB® no encontrará los archivos que no estén en el directorio de trabajo (ni siquiera en subcapetas), salvo que la ruta donde estén dichos archivos se haya incluido en el *Path* de búsqueda de MATLAB®, mediante la opción *Set path* de la barra de herramientas superior. En general, para proyectos o prácticas pequeñas, se recomienda colocar todos los archivos necesarios en la misma carpeta (sin usar subcarpetas), y que ésta sea el directorio de trabajo activo.

3 **Directorio actual o *Current folder***: se muestran los archivos y subcarpetas del directorio actual. Por defecto, al ejecutar código MATLAB® buscará los archivos necesarios en esta misma carpeta (no en sus subcarpetas). Ver punto 2 y nota al pie para más información. Al seleccionar alguno de los elementos del directorio activo, se muestra información sobre dicho elemento en *Details*, en la parte inferior de *Current folder*.

4 **Ventana de comandos o *Command window***: en esta parte se pueden escribir los comandos o funciones de MATLAB® que queramos ejecutar, y tras pulsar ↵ (*Enter* o *Intro*) en el teclado, se ejecutarán estos comandos y se mostrarán los resultado en la misma ventana. Para un código de cierta complejidad, no es recomendable usar la *Command window* para introducir los comandos, excepto si es para alguna pequeña prueba o comprobación elemental. En su lugar, es preferible trabajar con el *Editor* de archivos y con *scripts* (ver Apartado 3.1), que se abre directamente al pinchar, por ejemplo, en *New*

script en la barra 1.

5 **Espacio de trabajo o *Workspace***: en esta ventana, inicialmente vacía, se irán mostrando automáticamente las distintas variables que se hayan creado durante la sesión de uso de MATLAB® , así como sus valores o tipo de datos. Haciendo doble click sobre cualquiera de las variables, se abre otra ventana que muestra información adicional sobre su contenido.

Las secciones 3, 4 y 5 se pueden mover y recolocar de varias formas, pinchando y arrastrando con el ratón en la barra azul oscura del título de cada una. También se pueden minimizar, maximizar, desacoplar (*undock*, separar de la ventana principal), o acoplar (*dock)*, si ya se ha separado previamente) pinchando en el menú de la esquina superior derecha (triángulo dentro de un círculo) de cada una de ellas. Normalmente, no es necesario tener siempre visibles las secciones 3 y 5 y se pueden minimizar para dejar más espacio a las demás, especialmente cuando se usa el *Editor* que a continuación se explica.

3.1. Editor y uso de *scripts*

Como se decía en el punto 4 anterior, no se recomienda ejecutar un programa de MATLAB® de cierta envergadura a través de la *Command window*. Esto supondría tener que ir escribiéndolo y ejecutándolo línea a línea, o bien copiando y pegando desde algún editor externo. En vez de eso, MATLAB® incluye un Editor de archivos que nos permitirá ir guardando todos los comandos en uno o varios archivos en el formato de '.m' o '.mlx' de MATLAB® . Para abrir el editor basta con pinchar sobre la barra 1 en *New script*, o bien en *New -> Script*, o en *Open* y elegir un script en formato MATLAB® que ya exista previamente. Al hacerlo, aparece una nueva ventana, el Editor de archivos 6. Tras minimizar las ventanas de *Current folder* 3 y *Workspace* 5 (una vez minimizadas, para abrirlas, basta pinchar sobre su nombre en los laterales), y después de reordenar las ventanas, el aspecto sería algo parecido al de la Figura 1.2.

Ahora, además de las ventanas iniciales, tenemos una nueva:

6 **Editor**: en esta ventana podemos abrir los ficheros en formato de MATLAB® : *scripts* o funciones, de tipo '.m', y *live scripts* o *live functions* de tipo '.mlx'. Los archivos se abren en pestañas, de forma que es posible tener varios abiertos simultáneamente. Trabajaremos fundamentalmente con *scripts* '.m'. Un script o guion contiene una serie de comandos de MATLAB® que se irán ejecutando de uno en uno, por orden. Al agrupar muchos comandos en un solo script, podemos ejecutarlos todos de una vez, simplemente ejecutando el script. Para ejecutar un script completo, podemos pinchar en *Run* del menú 1, pulsar la tecla F5 en el teclado, o escribir el nombre del script en la *Command window* y pulsar ↵ (*Intro*) en el teclado. Conforme se va ejecutando el script, los resultados que se produzcan se mostrarán en la *Command window*.

4. Interfaz gráfica de Octave

Al igual que en la sección anterior, como complemento o alternativa a la explicación escrita que hay a continuación, se ha elaborado un vídeo explicativo sobre los elementos fundamentales y la interfaz gráfica del programa informático Octave, que se puede consultar en la siguiente dirección web: `https://youtu.be/Y5cVjbHnAq0`

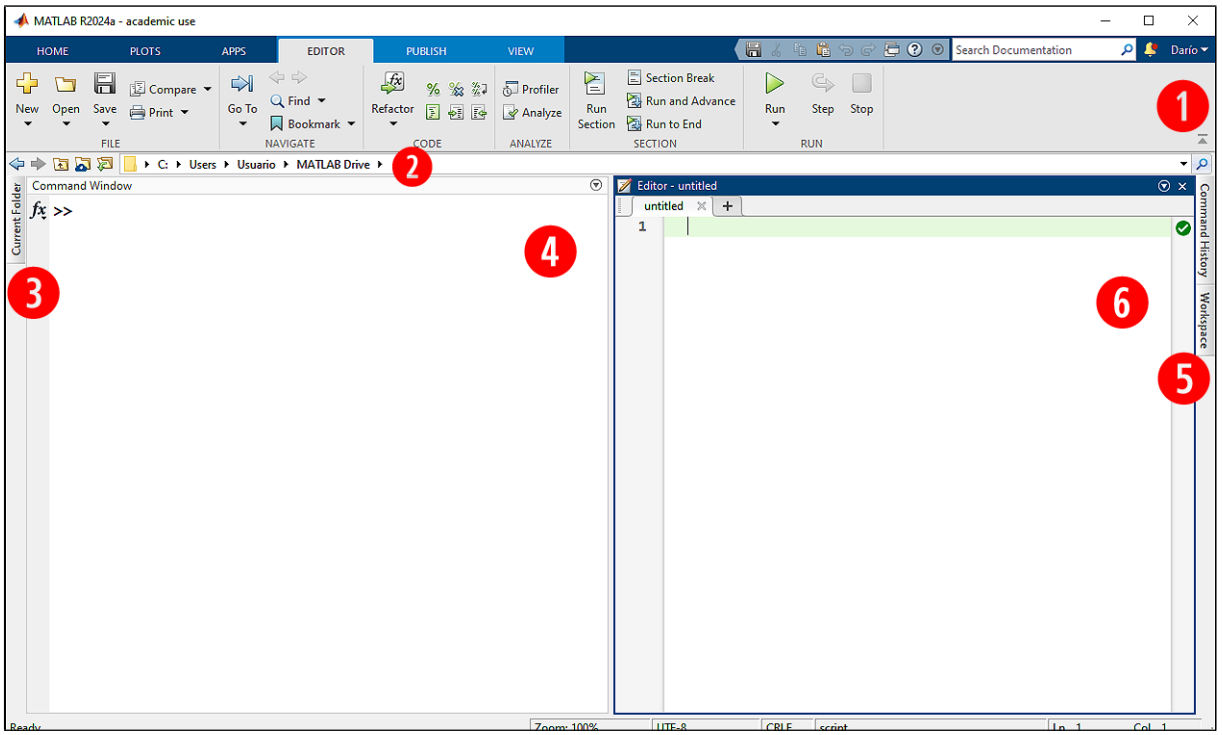

Figura 1.2: Entorno gráfico de MATLAB® tras reorganizar las ventanas y abrir el Editor.

Tras la instalación de Octave, tenemos dos formas muy diferentes de arrancarlo: mediante "Octave (GUI)" o con "Octave (CLI)". GUI son las siglas en inglés de *Graphical User Interface*, o interfaz gráfica de usuario, en contraposición de CLI, que viene de *Command Line Interface*, intefaz de línea de comandos). Salvo para un usuario avanzado, es preferible usar Octave a través de la interfaz gráfica, así que abriremos "Octave (GUI)".

El aspecto inicial del entorno de Octave (Figura 1.3) es similar al ya comentado para MATLAB® (Figura 1.1), con algunas diferencias. En Octave, inicialmente aparecen más secciones o subventanas abiertas, que son esencialmente las que ya habíamos visto en MATLAB®, incluyendo el Editor, y alguna más.

El menú de herramientas, sección 1, está separado entre las herramientas generales del programa Octave (parte superior izquierda) y las herramientas específicas del Editor (parte superior derecha, dentro de la ventana 6). La cantidad y tipo de herramientas, así como su ubicación difiere de las disponibles en MATLAB®, aunque la funcionalidad básica es muy similar.

Las secciones o ventanas desde la 2 hasta la 6 tienen prácticamente las mismas funciones que sus análogas en MATLAB® , es decir: *Current directory* o Directorio actual 2, *File browser* o Navegador de archivos 3 (*Current folder* en MATLAB®), *Command window* o Ventana de comandos 4, *Workspace* o Espacio de trabajo 5, y Editor 6.

Para una descripción de cada una de estas ventanas, recomendamos consultar el Apartado 3. Al igual que en MATLAB®, el uso de scripts mediante el Editor, explicado en el Apartado 3.1, está altamente recomendado. El usuario de Octave debería leer en detalle ambos Apartados, pues la mayor parte de su contenido es aplicable tanto a MATLAB® como a Octave.

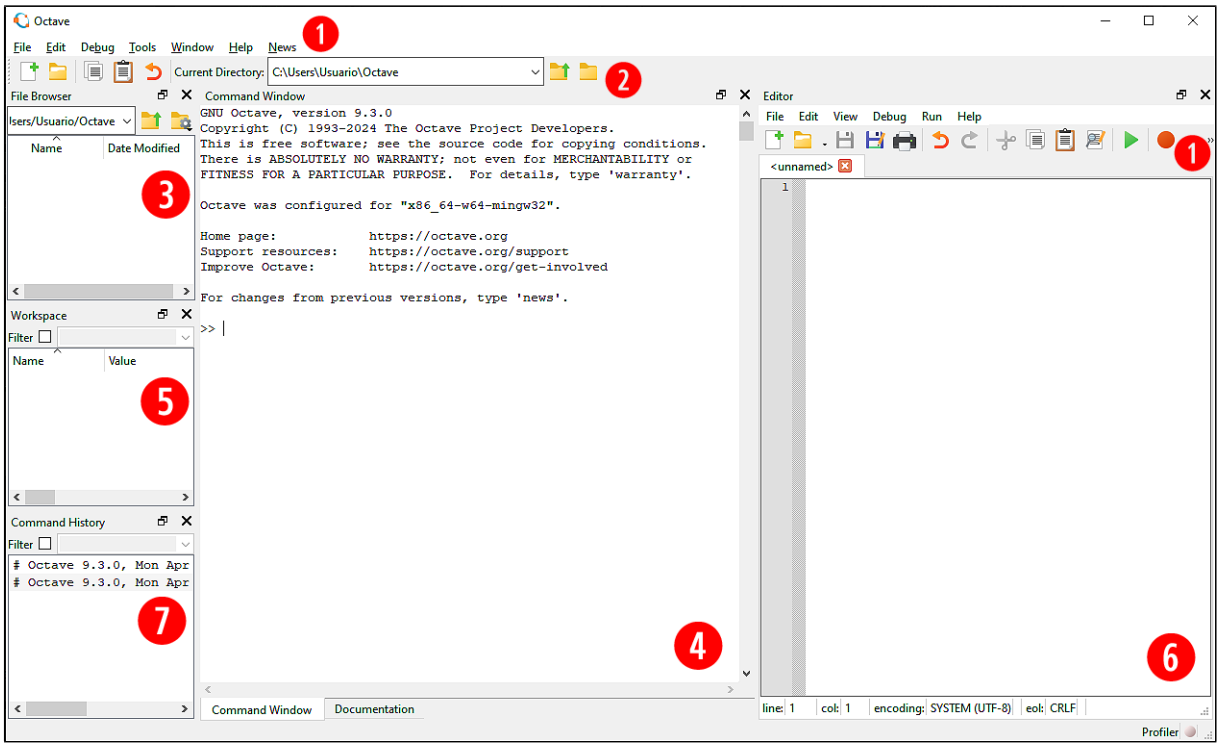

Figura 1.3: Entorno gráfico de Octave (GUI), dividido en varias secciones.

La ventana 7 de Octave, ***Command history* o Historial de comandos**, no aparecía por defecto en MATLAB®, aunque también existe y se puede mostrar. En esta ventana aparece un listado con todos los comandos que se hayan ejecutado en Octave, incluso en sesiones previas (cuando se cierra y se vuelve a abrir Octave, no se borra esta lista). De esta forma, es fácil volver a acceder a los comandos ya ejecutados previamente, para consultarlos, repetir su ejecución o corregirlos.

En Octave las distintas subventanas (*Workspace*, *Command history*, etc.) no se pueden minimizar al estilo de MATLAB® , pero sí se pueden cerrar o desacoplar con los iconos usuales, arriba a la derecha de cada una. Se pueden volver a abrir o restaurar la vista por defecto desde el menú *Window* en 1.

El acceso a la ayuda/documentación de Octave se puede hacer desde el menú *Help* en 1, o también desde la parte inferior de la *Command window*, en la pestaña *Documentation*.

5. Comparativa MATLAB® - Octave para la docencia

Aunque los entornos gráficos de MATLAB® y Octave, así como sus lenguajes de programación (como se verá más adelante) son muy similares, y en muchos aspectos equivalentes, hay algunas diferencias importantes, especialmente desde el punto de vista de la docencia.

La ayuda es una de las grandes diferencias entre MATLAB® y Octave. La ayuda de MATLAB®, en general, suele ser mucho más exhaustiva, incluyendo ejemplos, fundamentos matemáticos, y mucha otra información relacionada. En cambio, la de Octave es mucho más sucinta, especialmente en la documentación de funciones, que suele limitarse a una breve explicación del

objetivo de la función y sus parámetros, generalmente sin ejemplos de uso.

Otra diferencia importante entre ambos entornos se da cuando se producen errores de ejecución. Al intentar ejecutar algún comando o script, es frecuente que se produzcan errores por distintas causas. En estos casos, tanto en MATLAB® como en Octave, la ejecución se interrumpirá debido al error, y se mostrarán uno o más mensajes describiendo el problema. Los mensajes de error que proporciona MATLAB® suelen ser muy detallados, dando tanto una explicación (en inglés) del motivo del error, como su localización e incluso, en ocasiones, ofrece posibles soluciones. En cambio, los mensajes de error de Octave son mucho más escuetos, ofreciendo solo una breve descripción del problema, lo que dificulta la corrección de los errores que surjan.

Las diferencias mencionadas hacen que, desde el punto de vista del autor, MATLAB® sea más recomendable para la enseñanza-aprendizaje que Octave, al ser la ayuda del primero mucho más completa y didáctica, y también la información que proporciona en el caso de que haya errores (y durante el proceso de aprendizaje, suele haber muchos). Una vez adquirido el dominio suficiente, un usuario avanzado podría utilizar Octave para sustituir totalmente a MATLAB®. En cambio, el uso de Octave por parte de un usuario inexperto puede ser complicado, dificultando su aprendizaje o la resolución de los problemas que aparezcan.

6. Elementos básicos del lenguaje de MATLAB® y Octave

En esta sección explicamos los comandos y funciones elementales de MATLAB® y Octave que utilizaremos durante las prácticas propuestas. No pretende ser una documentación exhaustiva de los lenguajes de programación de MATLAB® y Octave, pero sí es algo más que una introducción inicial para principiantes. Se incluyen contenidos que, si bien no son estrictamente necesarios como parte de una introducción elemental, es útil o recomendable conocer para tener más soltura. El objetivo de esta guía es que pueda servir a aprendices en sus primeros pasos con el lenguaje de MATLAB® y Octave, como referencia de consulta posterior, y para usuarios algo más avanzados o que ya tengan experiencia previa.

Dividiremos la explicación en varias subsecciones, según la temática de los elementos expuestos, de forma que sea más fácil consultar directamente ciertas partes del contenido. Como complemento o alternativa a este material, se proporciona un vídeo explicativo sobre estos contenidos, al que se puede acceder desde el enlace web: `https://youtu.be/prU-gvFwHkw`

Junto al texto explicativo, se mostrarán fragmentos de código ejecutable. Este código, que se puede introducir en un script o función, o bien directamente en la Ventana de Comandos, aparecerá recuadrado y con fondo amarillo pálido, como aquí:

```
ejemplo de código ejecutable en MATLAB/Octave
```

En muchos casos también se mostrará el resultado que se muestra por pantalla, en la Ventana de Comandos, una vez que el código se ha ejecutado. En este caso, estos resultados (que no son código ejecutable), se mostrarán en cuadros con fondo blanco, como este:

```
ejemplo de resultados obtenidos en MATLAB/Octave
```

Además, se incluirán comentarios o explicaciones de un nivel algo superior al elemental, con el objetivo de que sirva a lectores que ya tengan cierta experiencia previa o quieran profundizar

en los contenidos. Para un lector que sea nuevo en MATLAB® , Octave, o en general en la programación, estas partes se pueden omitir en una primera lectura. Este contenido de ampliación se encontrará en cuadros con fondo gris claro y esquinas redondeadas, como la siguiente:

Ampliación: Para ampliar

En estos cuadros de texto encontrarás contenido de nivel algo más avanzado.

Todo el texto que corresponda a código ejecutable, o bien a resultados de una ejecución, se mostrarán con un tipo de letra distinto al texto normal, usando fuente tipográfica monoespaciada, como esta: `código ejecutable`.

6.1. Variables y operaciones elementales

Como en cualquier lenguaje de programación, en MATLAB® se pueden manejar datos numéricos de distinto tipo, vectores, matrices, cadenas de texto, funciones, etc. Todos estos tipos de datos se pueden guardar directamente en una variable sin necesidad de declararla antes de su uso, ni de convertirla de un tipo a otro.

Para asignar un valor (del tipo que sea) a una variable, se usa el operador `=` , poniendo a su izquierda el nombre de la variable y a su derecha el valor a asignar. Los nombres de variables en MATLAB® tienen que empezar por una letra, pueden contener letras, números y guiones bajos (_), pero ningún carácter de otro tipo. En Octave, se aplican las mismas reglas, excepto que sí se admite que el nombre empiece por guión bajo. Un error frecuente, que hay que evitar, es el uso en el nombre de variables de guiones comunes (-), paréntesis o corchetes. Una vez definida una variable, se puede utilizar su valor escribiendo su nombre, como se verá en los ejemplos a continuación.

Constantes numéricas Para números no enteros, el separador decimal es el punto (.), y se puede utilizar notación científica para introducir datos, en el formato `MeP`, donde `M` es la mantisa, `e` es un carácter literal y `P` el exponente de 10, es decir, se trata del número $M \times 10^P$. Por ejemplo, `1.062e2` representa al 106.2. Este formato también es utilizado por MATLAB® y Octave al mostrar los resultados.

Igualmente, se pueden introducir números complejos usando `a + b*1i` ó `a + b*1j` donde `a` es la parte real y `b` la parte imaginaria. Tanto `1i` como `1j` se interpretan como la unidad imaginaria.

La constante matemática π se puede utilizar tanto en MATLAB® como en Octave escribiendo `pi`. En cambio, mientras que el número e se puede utilizar directamente en Octave poniendo `e`, en MATLAB® no está disponible por defecto y en caso de necesitar su valor se puede obtener mediante la función exponencial e^x evaluada en $x = 1$, lo que se hace con `exp(1)`. Rara vez será necesario obtener el valor de e como tal, y bastará evaluar la función `exp(x)` en un valor adecuado de `x` (además, usar `exp(x)` dará siempre resultados más precisos que calcular e y luego elevarlo a x; no son operaciones equivalentes computacionalmente[2]).

Las variables numéricas se pueden operar entre ellas con los operadores usuales de suma, resta, producto, división o exponenciación: `+`, `-`, `*`, `/`, `^`, respectivamente. En una expresión con

[2]Esto se debe a la forma optimizada en que está implementada la función `exp(x)`, e^x, y también a la *aritmética de la máquina*, que es la forma en la que se representan números y se opera con ellos en un ordenador (se tratará este tema más adelante).

varias operaciones, se respeta la jerarquía usual de las operaciones, es decir, primero se aplican funciones, después potencias, luego multiplicaciones y divisiones, y finalmente sumas y restas. El orden de estas operaciones se puede modificar usando paréntesis, cuando sea necesario (es conveniente no abusar de los paréntesis, evitando su uso si no son imprescindibles).

Se muestran a continuación algunas asignaciones de variables y operaciones elementales:

```
a = 2            % Asignamos a la variable 'a' el valor 2
b = pi           % Se puede usar la constante pi directamente
c = a*b          % Producto de 'a' por 'b'
e                % Error en Matlab, funciona en Octave
exp(1)           % Recomendable usar exp(x), y no e^x, en Matlab y en Octave
3*a^3-6          % Se respeta el orden habitual de las operaciones
3*(a^3-6)        % Se pueden usar paréntesis para alterar el orden usual
```

Al ejecutar los comandos anteriores en MATLAB® , el resultado o salida por pantalla se muestra en la *Command window* o Ventana de Comandos:

```
a =
    2
b =
    3.1416
c =
    6.2832
Unrecognized function or variable 'e'.
ans =
    2.7183
x =
    18
y =
    6
```

Comprobamos que se obtienen los resultados esperados, pero sale un error al ejecutar la cuarta línea (`e`) en MATLAB® , aunque esta línea de código sí funcionaría en Octave. Observa que, cuando no se asigna el resultado a una variable, como en las últimas líneas, este se asigna automáticamente a la variable `ans` (iniciales de *answer*, respuesta, en inglés), de forma que la variable `ans` siempre contiene el último resultado que no haya sido asignado a otra variable.

En ocasiones, como resultado de ciertas operaciones con números, se obtienen resultados que no son numéricos. Es lo que sucede cuando dividimos por cero (`3/0`), o si una operación da como resultado un número extremadamente grande (`10^500`); en estos casos obtenemos `Inf` ó `-Inf` ("infinito" ∞, o "menos infinito" $-\infty$, respectivamente). Naturalmente, `Inf` no es un número corriente, pero se pueden hacer ciertas operaciones aritméticas con él. Ejemplos: `2*Inf` da como resultado `Inf`, y con `7/Inf` se obtiene `0`. Con algunas indeterminaciones, el resultado no será numérico ni tampoco $\pm\infty$. En estos casos, obtendremos `NaN` (siglas de *Not-a-Number*, no es un número, en inglés). Ejemplos: se produce un `NaN` como resultado de `0/0`, de `Inf-Inf`, o de `Inf/Inf`, pero no con `Inf/0` ni con `0/Inf`, que dan, respectivamente, `Inf` y `0`.

6.2. Definición de vectores y matrices

Se pueden crear vectores o matrices, o en general *arrays*, haciendo uso de los corchetes: `[ ]`. Veamos las particularidades de cada uno.

Vectores:

Para vectores fila, basta escribir sus componentes separadas por comas (`,`) o espacios en blanco, entre los corchetes (se recomienda usar la coma preferiblemente). Ejemplo: `v = [1,2,3,4,5]`. Para un vector columna, el separador debe ser el punto y coma (`;`). Ejemplo: `[1;2;3]`.

Para crear vectores equiespaciados o linealmente espaciados, en los que cada elemento del vector es obtiene sumando un paso fijo al elemento anterior, se pueden utilizar estas dos formas:

- Operadores `:` y `::` — el operador `:` (*colon*, en inglés) genera una secuencia de números equiespaciados. Podemos poner `a:b`, que devolverá un vector con los números desde `a` hasta `b`, con un paso o salto de 1 (distancia entre cada dos números consecutivos). Otra opción es utilizar `a:h:b`, que hará esencialmente lo mismo, pero con un paso o salto de valor `h` especificado. Dependiendo de los valores de `a`, `h` y `b`, es posible que `b` no esté incluido en la secuencia obtenida.

- Comando `linspace` — alternativamente, se puede utilizar la función `linspace(a,b,n)`, que devolverá una secuencia de `n` números equiespaciados entre `a` y `b`. También se puede usar `linspace(a,b`, que por defecto devuelve una secuencia de 100 números entre `a` y `b`. La secuencia obtenida con `linspace` siempre incluye a los extremos `a` y `b`.

Mientras que con el operador `:` se puede indicar el paso `h`, con `linspace` lo que se indica es la cantidad de números que se quiere obtener. Estas dos formas de crear vectores producen vectores fila.

Para trasponer un vector, por ejemplo para transformar un vector fila en uno columna, se pone después de su nombre el operador apóstrofe (`'`), de forma que el código `v'` equivale a la operación matemática v^T, si v es un vector con coeficientes reales.

Ejemplos de definición de vectores:

```
v = [3,4,5]          % Vector fila indicando todos los valores
v = 3:5              % Secuencia en un rango (inicio y fin) con paso 1
w = 1:0.5:3          % Secuencia en un rango con paso determinado
w = linspace(1,3,5) % Secuencia equivalente con linspace
x = [1:3]'           % Vector columna (1,2,3)
```

```
v =
    3     4     5
v =
    3     4     5
w =
    1.0000    1.5000    2.0000    2.5000    3.0000
w =
    1.0000    1.5000    2.0000    2.5000    3.0000
x =
    1
    2
    3
```

Como se ve, las dos primeras líneas de este bloque son equivalentes entre sí (producen el mismo resultado), al igual que las dos siguientes.

Matrices:

Las matrices se pueden introducir de forma similar, indicando sus componentes por filas, y separando cada fila de la siguiente con punto y coma (`;`). Ejemplo: `M = [1,2;-4,3]` será la matriz $M = \begin{pmatrix} 1 & 2 \\ -4 & 3 \end{pmatrix}$. Además, se pueden construir matrices elementales con los comandos:

- `ones(n,m)`: crea una matriz de unos (todos los elementos son 1) con n filas y m columnas. Se puede usar con un solo parámetro, creando una matriz cuadrada $n \times n$. También sirve para crear vectores con $n = 1$ ó $m = 1$, según sea fila o columna.
- `zeros(n,m)`: crea una matriz de ceros con n filas y m columnas. Se puede usar con un solo parámetro, creando una matriz cuadrada $n \times n$. También sirve para crear vectores con $n = 1$ ó $m = 1$, según sea fila o columna.
- `eye(n)`: crea la matriz identidad de orden n, usualmente denotada I_n.

Al igual que para vectores, se puede obtener la traspuesta de una matriz con coeficientes reales mediante el operador apóstrofe (`'`). Es decir, `M'` es el código para hacer la operación matemática M^T, si M es una matriz de coeficientes reales.

Ejemplos de creación de matrices:

```
M = [1,2;-4,3]          % Matriz 2x2
N = [2,1,-3;-1,0,3]     % Matriz 2x3
I = eye(3)              % Matriz 3x3
O = ones(2)             % Matriz 2x2
Z = zeros(2,3)          % Matriz 2x3
M'                      % Matriz 2x2
```

```
M =
     1     2
    -4     3
N =
     2     1    -3
    -1     0     3
I =
    1     0     0
    0     1     0
    0     0     1
O =
    1     1
    1     1
Z =
    0     0     0
    0     0     0
ans =
    1    -4
    2     3
```

Ampliación: Matrices complejas y otros comandos para matrices

El operador apóstrofe (`'`), visto anteriormente, sirve para trasponer vectores o matrices con coeficientes reales. Sin embargo, cuando tienen coeficientes complejos, lo que devuelve dicho operador es la traspuesta conjugada o traspuesta Hermitiana, en la que, además de trasponer el vector o la matriz original, se sustituye cada elemento $a + bi$ por su complejo

conjugado $a - bi$. En matemáticas, para una matriz A, esta operación se suele denotar por A^H o A^*. Para hacer la trasposición usual, sin conjugación, de una matriz o vector complejo `C`, se puede hacer anteponiendo un punto al operador apóstrofe (`C.'`). En el caso de coeficientes reales, ambos operadores son equivalentes.

Otras funciones algo más avanzadas y muy útiles para la creación de matrices, son:

- `diag(v)`: crea una matriz diagonal con los elementos del vector `v` en la diagonal principal. También se puede utilizar para extraer una diagonal de una matriz dada. Esta función es muy flexible y se puede utilizar de otras formas, consulta la ayuda de MATLAB® para más información.
- `rand(n,m)`: crea una matriz de `n` filas y `m` columnas, cuyas entradas son números aleatorios uniformemente distribuidos en el intervalo $(0, 1)$.
- `gallery`: esta función genera matrices *test* o de prueba, de distintos tipos y con distintas propiedades (matrices mal condicionadas numéricamente, cíclicas, simétricas y definidas positivas, etc.). Remitimos a la ayuda de MATLAB® para una referencia completa de sus posibilidades.
- `vander(v)`: sirve para obtener matrices de Vandermonde, cuyas columnas son las potencias de un vector `v` dado.

6.3. Operaciones con vectores y matrices

Se puede operar combinando números, vectores o matrices directamente mediante los operadores usuales: `+`, `-`, `*`, `/`, `^`; todas estas operaciones se interpretan de forma matricial o vectorial cuando no sean entre escalares. Es decir, las dimensiones de las matrices o vectores involucrados en las operaciones deben ser las necesarias para que la operación pueda realizarse. Por ejemplo, si `A` y `B` son matrices, `A*B` obtiene el producto matricial de ellas, siempre que sea posible calcularlo dadas sus dimensiones (el número de columnas de `A` debe coincidir con el número de filas de `B`).

Para conocer las dimensiones de un vector o matriz utilizaremos el comando `size(X)`, que devolverá un vector de dos componentes con su número de filas y número de columnas respectivamente. En el caso de vectores, podemos utilizar `length(x)`, que nos dará un sólo número, la cantidad de elementos del vector (en este caso, no sabremos si es un vector fila o columna).

Utilizando las matrices y vectores definidos en la sección previa, esto es, las matrices $\mathtt{M}_{2\times2}$, $\mathtt{N}_{2\times3}$, $\mathtt{I}_{3\times3}$, $\mathtt{O}_{2\times2}$, $\mathtt{Z}_{2\times3}$, y los vectores $\mathtt{v}_{1\times3}$ y $\mathtt{x}_{3\times1}$. Teniendo en cuenta sus dimensiones, se podrían hacer estas operaciones (analice cada operación y razone si efectivamente se pueden realizar):

- `2*M,  5+Z,  2+x`
- `M+O,  N-Z,  v+x'`
- `M*O,  M*Z,  (N+Z+1)*x,  2*v*I`
- `M^4,  M^2+O^7`

En cambio, obtendríamos un error al hacer estas otras operaciones:

- `M+Z`, `N+O`, `v+x`
- `Z*O`, `Z*M`, `M*x`, `N*v`
- `N^2`, `Z^5`, `x^2`, `v^3`

En particular, en el último punto tenemos la operación `x^2`, sobre el vector `x` de tamaño 3×1. El cuadrado de x, visto como matriz, corresponde a multiplicar el vector x por sí mismo: $x * x$. Sin embargo, para que esta operación pueda efectuarse, el número de columnas del primer factor (1) debería coincidir con el número de filas del segundo factor (3), por lo que esta operación no puede hacerse. En general, podemos decir que, para cualquier vector x (con más de un elemento) o cualquier matriz M no cuadrada, no se puede ejecutar el código `x^n` ni `M^n`.

Operaciones elemento a elemento:

Existe una modificación de las operaciones de multiplicación, división y exponenciación para vectores o matrices. Son las operaciones denominadas componente a componente, o elemento a elemento (*element-wise*), en las que la operación indicada se aplica individualmente a cada elemento del vector o matriz. Se hace con los operadores `.*`, `./`, `.^` (anteponiendo un punto al operador matricial correspondiente). Para sumas y restas no existe esta modificación, pues siempre son operaciones componente a componente. Para operaciones entre un escalar y un vector o matriz no hay que indicar explícitamente que se hace elemento a elemento (se asume). En el caso de multiplicaciones o divisiones entre vectores o matrices, ambos factores deben tener las mismas dimensiones (igual número de filas y de columnas). En particular, se pueden calcular las potencias elemento a elemento de cualquier vector o matriz. Utilizando estas operaciones, podemos ejecutar con éxito los siguientes comandos:

- `M.*O`, `Z./N`, `x./v'`
- `N.^2`, `Z.^5`, `x.^2`, `v.^3`

Aunque estas operaciones aplicadas elemento a elemento pueden ser útiles en ocasiones, para simplificar y optimizar el código, o evitar bucles, debe quedar claro que no corresponden con los productos usuales de vectores y matrices utilizados en el álgebra lineal.

Ampliación: "División" matricial

Como se puede comprobar en los ejemplos previos, excepto en las operaciones elemento a elemento, no se han realizado divisiones entre vectores o matrices. La división de un vector entre otro, de una matriz entre otra, o de una matriz entre un vector, no son operaciones definidas en matemáticas.

En MATLAB® y Octave, sin embargo, en ocasiones es posible ejecutar el código `M/v`, donde `M` es una matriz y `v` es un vector, y que esta sea una operación válida y tenga sentido. Incluso, existe una operación de "división a izquierda", que se hace con la barra invertida: `M\`. Como veremos a continuación, estas dos operaciones no son divisiones como tal, sino que se realizan otras operaciones.

La operación `M/v` (o `mrdivide`, de *matrix right division*) se interpreta como la solución del sistema de ecuaciones lineales $x * M = v$, es decir, lo que se obtiene es la solución x del sistema. Grosso modo, si M es cuadrada, lo que se obtiene con este comando es $x = v * M^{-1}$, aunque se calcula de una forma muy distinta, sin utilizar la inversa. La

operación `M\v` (o `mldivide`, de *matrix left division*) es completamente análoga, pero para el sistema de ecuaciones lineales $M * x = v$. En este caso, si M es cuadrada, lo que se obtiene es, aproximadamente, $x = M^{-1} * v$.

Para que puedan utilizase estos comandos, las dimensiones de M y v deben ser coherentes (son comandos muy flexibles, y remitimos a la ayuda de MATLAB® y Octave para una explicación exhaustiva de todos los casos posibles; incluso es posible tomar como v otra matriz).

Además, las órdenes anteriores pueden devolver un resultado si el sistema de ecuaciones es incompatible, y en este caso, lo que se obtiene es la solución en el sentido de *mínimos cuadrados*, concepto que se estudiará mas adelante.

6.4. Indexación de vectores y matrices

La indexación o indización de vectores y matrices se refiere a cómo podemos acceder, seleccionar o extraer sus distintos elementos o subconjuntos, y es un aspecto esencial.

Para acceder a un elemento de un vector, basta poner el nombre del vector y entre paréntesis el índice o posición que ocupa dicho elemento en el vector: `v(indices)`. En MATLAB® y Octave, estas posiciones empiezan en `1` y llegan hasta la longitud del vector, `length(v)`. Se puede utilizar `end` para acceder al último elemento. Ejemplos: `v(1)`, `v(4)`, `v(end)` Se puede acceder simultáneamente a un conjunto de elementos de un vector, utilizando entre los paréntesis un vector que contenga los índices de los elementos deseados. En el resultado, obtendremos un vector de tantos elementos como índices hayamos utilizado, y en el mismo orden. Ejemplos: `w(1:3)`, `w(4:end)`, `w([1,4,5])`; en el primer caso se obtienen los elementos de las posiciones 1, 2 y 3; en el segundo caso, desde la posición 4 hasta el final del vector, y en el tercer caso, sólo los elementos 1, 4 y 5. Veamos lo que obtenemos en cada caso:

```
w = 1:0.5:3          % Vector w
w(1:3)               % Elementos de w hasta la posición 3 (incluida)
w(4:end)             % Elementos de w a partir de la posición 4 (incluida)
w([1,4,5])           % Elementos de w de las posiciones 1, 4 y 5
```

```
w =
    1.0000    1.5000    2.0000    2.5000    3.0000
ans =
    1.0000    1.5000    2.0000
ans =
    2.5000    3.0000
ans =
    1.0000    2.5000    3.0000
```

Para matrices, la indexación funciona de forma similar, excepto que hay que indicar cuáles son las filas y columnas a las que se quiere acceder, separando con una coma los índices. Para una matriz `M`, poniendo `M(indF, indC)` accedemos a los elementos de las filas cuyos índices son `indF` y de las columnas cuyos índices son `indC`. Ejemplos: `M(2,3)` devolverá el elemento de la 2ª fila y 3ª columna; mientras que `M(1:2,1:2)` devolverá los elementos de las dos primeras filas y columnas (es decir, la submatriz principal de orden 2). Para una matriz `M`, para acceder a las filas completas con índices `indF`, se utiliza `M(indF,:)` (se toman los elementos de todas las columnas, es decir, las filas completas). Análogamente, para acceder a las columnas de índices `indC`, ponemos `M(:,indC)`. Lo comprobaremos con algunos ejemplos:

```
M = [2,1,-3;-1,0,3]  % Matriz M
M(2,3)               % Elemento de la 2ª fila y 3ª columna
M(1:2,1:2)           % Submatriz con las dos primeras filas y columnas
M(2,:)               % Se toma la segunda fila completa
M(:,end)             % Se accede a la última (3ª) columna completa
```

```
M =
     2     1    -3
    -1     0     3

ans =
     3
ans =
     2     1
    -1     0
ans =
    -1     0     3
ans =
    -3
     3
```

6.5. Comentarios y secciones

Como se ha visto los ejemplos anteriores, se pueden añadir textos explicativos intercalados con el código, llamados comentarios. En los lenguajes de MATLAB® y Octave se pueden incluir comentarios en cualquier línea (tenga o no código ejecutable delante), escribiéndolos detrás del símbolo de tanto por ciento (`%`). Todo lo que se escriba a la derecha de este símbolo, será ignorado al ejecutar el comando. En Octave, también se puede usar el símbolo almohadilla (`#`) para escribir comentarios.

Además, en MATLAB®, el código de un script se puede dividir en secciones, de forma que es más sencillo visualizar lar partes del código, o ejecutar cada una de las secciones por separado (esta funcionalidad no está disponible en Octave). Para crear una nueva sección dentro del script basta añadir una línea que empiece por un doble símbolo de tanto por ciento (`%%`), y si se quiere indicar un nombre para la sección se escribe a la derecha, dejando un espacio en blanco. Al crear secciones, la parte del archivo antes de la primera sección también se considera directamente otra sección, aunque no se indique el símbolo `%%`. Al crear secciones, aparecen separadas en el Editor de MATLAB® con una línea separadora horizontal.

A continuación se muestra un fragmento de código con comentarios y secciones. En este ejemplo, las secciones tienen un solo comando cada una, pero generalmente contendrán varias órdenes relacionadas entre sí:

```
x = pi/4;
%% SECCIÓN 2
y = cos(2*x);  % Calculamos el coseno del ángulo doble
%% SECCIÓN 3
z = acos(y)/2  % Recuperamos x mediante la función inversa
```

Una vez estructurado el código en secciones, se puede ejecutar un script completo pinchando en *Run* en el menú de herramientas, o F5 en el teclado, o bien ejecutar solo la sección activa en ese momento pinchando en *Run section* o las teclas Ctrl+↵ (*Enter* o *Intro*). También hay

otras variantes de estas opciones, que permiten ejecutar una sección y avanzar a la siguiente (*Run and Advance*), o ejecutar todo el código a partir de la sección actual (*Run to End*).

6.6. Funciones matemáticas usuales

Tanto en MATLAB® como en Octave, están predefinidas y se pueden usar de forma directa las funciones matemáticas usuales, y también otras menos comunes. Estas funciones se pueden aplicar directamente tanto a números reales como a complejos (siempre que tenga sentido matemático hacerlo), y tanto a escalares como a vectores o matrices (en este caso, se aplican elemento a elemento sobre los valores del vector o matriz, salvo que se indique lo contrario en la propia función). A continuación, enumeramos algunas de las funciones más importantes:

- Función exponencial: La función e^x se calcula como `exp(x)`. En el caso de que `x` fuera una matriz o vector, se aplica la exponencial elemento a elemento, no se calcula la exponencial matricial (esto se haría con `expm(x)`).
- Logaritmos: El logaritmo natural o neperiano, $\ln(x)$, se obtiene con `log(x)` (cuidado: no existe `ln(x)`, y `log(x)` calcula el logaritmo neperiano, no el logaritmo en base 10). Otros logaritmos, con base 2 o 10 se pueden obtener con `log2(x)` o `log10(x)` respectivamente.
- Funciones trigonométricas (todas operan en radianes):
  Las habituales funciones trigonométricas se pueden usar con las órdenes: `sin(x)` (cuidado: es `sin` con i, no `sen` con e; los nombres de las funciones están en inglés), `cos(x)`, `tan(x)` (esta es la tangente, no existe `tg(x)`).
  Sus inversas, las funciones arco, se utilizan anteponiendo una `a` a las funciones anteriores: `asin(x)`, `acos(x)`, `atan(x)`.
- Valor absoluto: Se puede obtener con `abs(x)`, y también sirve para el módulo si `x` es un número complejo. Sobre vectores o matrices, esta función calcula el valor absoluto o módulo de cada componente.
- Inversa y determinante: La inversa de una matriz `M` se puede obtener con `inv(M)` (aunque no es recomendable hacerlo, como veremos en algunas de las prácticas siguientes). El determinante se puede obtener con `det(M)`.

Ampliación: Funciones más avanzadas

Además de las anteriores, incluimos otras funciones muy útiles en matemáticas, algunas de las cuales se utilizarán más adelante:

- Funciones hiperbólicas: de forma análoga a las funciones trigonométricas usuales, existen las funciones hiperbólicas: `sinh(x)`, `cosh(x)`, `tanh(x)`, seno, coseno y tangente hiperbólicas respectivamente; y sus inversas, `asinh(x)`, `acosh(x)` y `atanh(x)`.
- Normas vectoriales y matriciales: Se puede calcular la norma euclídea (norma-2) de un vector o matriz con `norm(v)`. Para otras normas, si `v` es un vector, se puede usar `norm(v,p)` (norma-p), donde `p` puede ser un valor real, o bien `Inf` (norma-∞) o `'fro'` (norma Frobenius). Si `M` es una matriz y se calcula `norm(M,p)`, entonces `p` debe ser `1`, `2`, `Inf`, o `'fro'`.
- Número de condición: Se puede obtener el número de condición de una matriz `M` para la inversión mediante `cond(M)`. Igual que con `norm`, es posible especificar otras normas;

por defecto se usa la euclídea.

- Autovalores y valores singulares: Los autovalores y autovectores (o valores propios y vectores propios) de una matriz `M` se pueden obtener con el comando `eig(M)` (del inglés, *eigenvalues* y *eigenvectors*), que admite varias posibilidades de uso. De forma análoga, se puede obtener la descomposición en valores singulares de la matriz `M` con `svd(M)` (iniciales, en inglés, de *singular value decomposition*). Para más información, se recomienda consultar la ayuda de estas funciones.

Hay muchas más funciones matemáticas predefinidas, tanto en MATLAB® como en Octave, incluyendo funciones especiales como la Gamma $\Gamma(x)$, la Beta $B(x, y)$, las de Bessel de primera especie $J_\alpha(x)$ y de segunda especie $Y_\alpha(x)$, la de Airy $Ai(x)$, la Zeta de Riemann $\zeta(z)$, y muchas más. Se sugiere consultar la ayuda de cada entorno para más información sobre cualquiera de estas, o para localizar otras funciones predefinidas.

6.7. Definición de nuevas funciones

Para la resolución de prácticamente cualquier problema matemático con un entorno de programación, necesitaremos definir nuestras propias funciones, y esto engloba tanto lo que normalmente entendemos por funciones matemáticas, como algoritmos o métodos más complejos. Hay varias formas de definir funciones, distinguiremos dos, las funciones anónimas, definidas dentro del script donde se necesitan usar, y las funciones creadas en archivos dedicados. En cualquier caso, es recomendable que las funciones que definamos no sobrecarguen a las funciones estándar definidas en MATLAB® u Octave, es decir, hay que evitar llamar a nuestras propias funciones (también a los scripts que utilicemos) con nombres como: `abs`, `exp`, `cos`, `plot`, etc. A continuación explicaremos las dos formas fundamentales que hay para definir funciones.

Funciones anónimas Para definir nuevas funciones, cuando sean expresiones u operaciones relativamente simples, la mejor manera es usar lo que se llama una función anónima. Para crear una función anónima, se escribe el operador arroba (`@`); tras él, entre paréntesis, la o las variables de dicha función (separadas por comas); y finalmente, tras las variables, sin ningún carácter separador (u opcionalmente, espacios en blanco), se escribe la expresión matemática de dicha función. Ejemplo: `@(x) cos(2*x)` . Si se necesita guardar, se puede asignar su valor como cualquier otra variable (que será de tipo *function handle* o identificador de función). Ejemplo: `f = @(x) cos(2*x)` . Para aplicar una función, una vez definida, se pone su nombre y entre paréntesis los valores que tienen sus variables (que pueden ser otras variables que ya existan). Ejemplo: `f(pi)` .

Las funciones anónimas son tratadas como funciones matemáticas, es decir, no se calcula nada en el momento de definirlas, ni se aplican, incluso aunque las variables de la función ya tenga valores asignados en el *Workspace* (las variables de la función se consideran locales). Una función anónima es simplemente una expresión que se puede aplicar posteriormente sobre los valores que interesen. En las funciones, las variables se tratan de forma abstracta. Se puede consultar la ayuda sobre "Anonymous functions" para más información.

Veamos este sencillo ejemplo para funciones de una y dos variables:

```
x = pi/2;
y = pi;
funcion1 = @(x) cos(2*x)
funcion2 = @(x,y) -cos(y).*sin(x)
```

```
% Gracias al operador .*, funcion2 se puede aplicar a vectores y matrices
funcion1(x)
funcion2(x,y)
```

Al ejecutar el código anterior, se generan estos resultados en MATLAB® :

```
funcion1 =
    function_handle with value:
      @(x)cos(2*x)
funcion2 =
    function_handle with value:
      @(x,y)-cos(y)*sin(x)
ans =
    -1
ans =
    1
```

En Octave los resultados serían casi iguales, aunque no se indica explícitamente que `funcion1` y `funcion2` son identificadores de función (`function_handle`), sino que se deduce de la presencia del operador `@(.)` en las definiciones.

Observa que en las líneas 1 y 2 se definen `x` e `y`, pero cuando en las líneas 3 y 4 se crean las funciones `funcion1` y `funcion2`, estas no se aplican a ningún valor concreto, sino que simplemente se define su expresión matemática y se crea el identificador de dichas funciones. Es en las líneas 6 y 7 cuando realmente se aplican las funciones, a los valores de `x` e `y` definidos previamente. En la definición de `funcion2` se ha utilizado el operador `.*` para que la función pueda posteriormente aplicarse, elemento a elemento, sobre vectores y matrices.

Aún hay otra forma más de definir funciones, que es mediante le comando `inline`, al que se le pasa como argumento una cadena de texto con la expresión de la función. Sin embargo, desde hace bastante tiempo, el uso de las funciones `inline` está desaconsejado, y se recomienda utilizar siempre funciones anónimas en su lugar.

Funciones en ficheros dedicados Si queremos implementar una función matemática que tenga una expresión muy compleja, o que requiera calcular distintos pasos, como un algoritmo o un método numérico, podemos hacerlo en un fichero `.m` dedicado. Aunque de esta forma se pueden definir también funciones dentro de un script, colocándolas al final, lo más recomendable en general es separarlas y ponerlas en un fichero dedicado.

Para crear una función de este tipo en MATLAB® u Octave, podemos crear un nuevo script vacío y escribir la estructura que se verá a continuación, o bien crear un nuevo archivo de función, en cuyo caso nos aparece una plantilla con los elementos básicos que debe tener la función. Por ejemplo, al crear un nuevo archivo de función en MATLAB® , nos aparece el archivo con el código (en Octave aparecen otros nombres, pero la estructura es la misma):

```
1 function [outputArg1,outputArg2] = untitled(inputArg1,inputArg2)
2 %UNTITLED Summary of this function goes here
3 %   Detailed explanation goes here
4 outputArg1 = inputArg1;
5 outputArg2 = inputArg2;
6 end
```

Vemos que la línea 1 comienza con la palabra clave `function`, seguida de los parámetros de salida `outputArg1` y `outputArg2`, entre corchetes. Después viene el nombre de la función (por

ahora, se llama `untitled`) y tras este, entre paréntesis, los parámetros o argumentos de entrada de la función `inputArg1` e `inputArg2`.

Debajo, en las líneas 2 y 3, se incluye un espacio para comentarios explicativos acerca de la función. Idealmente, en el primero se escribe un breve resumen o explicación corta, y a continuación se puede incluir una explicación más detallada.

Las líneas 4 y 5 representan los cálculos que debe hacer la función internamente para, a partir de las variables de entrada, `inputArg1` e `inputArg2`, calcular las variables de salida `outputArg1` y `outputArg2`. Obviamente, tendremos que sustituirlas por los cálculos que sean necesarios para nuestra función.

La última línea de la plantilla, la 6, que indica el final de la función, es `end` en MATLAB® y `endfunction` en Octave. Una función que acabe con `end` se puede usar en Octave, pero una que acabe en `endfunction` dará error en MATLAB® .

Sobre la plantilla anterior se puede modificar todo lo que sea necesario. Por ejemplo, se puede cambiar el nombre de la función, teniendo en cuenta las mismas reglas que para los nombres de variables, debe empezar por una letra y contener solo letras, números o guiones bajos.
Importante: al guardar la función, el archivo `.m` debe tener el mismo nombre que la función que estamos definiendo dentro (en este caso, el archivo debería llamarse `untitled.m`, y es el nombre que MATLAB® sugiere si se intenta guardar el archivo).
También es posible cambiar de nombre, añadir o quitar parámetros de entrada o de salida, según nuestras necesidades. Incluso, es posible tener una función que no tenga ningún parámetro de entrada o de salida, y también se pueden manejar argumentos opcionales.

A modo de ejemplo, lo ilustramos con una función bastante trivial: dada la base b y altura h de un rectángulo, se calculan y devuelven su área y su perímetro. Esta función, `rectangulo`, se debe guardar en un fichero del mismo nombre, `rectangulo.m`:

Fichero rectangulo.m

```
function [area,perimetro] = rectangulo(b,h)
    %RECTANGULO   Se calcula el área y el perimetro de un rectángulo
    %   Dadas la base b y la altura h de un rectángulo, se calculan
    %   y se devuelven los valores de su área y su perímetro
    area = b*h;
    perimetro = 2*(b+h);
end
```

A la hora de utilizar esta función, habría que escribir, desde un script o la línea de comandos lo siguiente: `rectangulo(b, h)`, habiendo asignado previamente valores a estos parámetros de entrada, o directamente poniendo los valores numéricos en lugar de `b` y `h`. La función, después de realizar los cálculos necesarios, devolverá los valores de los parámetros de salida. Para funciones definidas de esta forma, podemos asignar uno, ninguno, o varios de los parámetros de salida, según nos interese. Además, al usar la función, los nombres de los parámetros de entrada y de salida no tienen por qué coincidir con los que se hayan usado internamente. Veamos algunos ejemplos.

```
L1 = 2;
L2 = 4;
[a,p] = rectangulo(L1,L2)   % Se guarda el área en a y el perímetro en p
[a] = rectangulo(L1,L2)     % Solo se guardan el área, en la variable a
[~,p] = rectangulo(L1,L2)   % Solo se guardan el perímetro, en la variable p
```

```
rectangulo(L1,L2)          % Se aplica la función, pero no se guardan resultados
% En este último caso, se muestra en pantalla solo el primer resultado (el área).
```

En la penúltima línea se ha utilizado el símbolo ~ para ignorar el primer argumento de salida. Este símbolo se llama en español virgulilla, o informalmente, *gusanillo*, y en inglés *tilde*. Se escribe en Windows con las combinaciones de teclas AltGr+4 y después Espacio, o bien Alt +1 2 6 en el teclado numérico. Este símbolo tiene varios usos en MATLAB® y Octave, como se verá más adelante en la sección Apartado 6.9.

Al utilizar una función definida en un archivo dedicado, es importante que dicho archivo esté guardado dentro del directorio actual de trabajo, o bien en algún directorio del *Search path* (lo más sencillo, en proyectos o prácticas pequeños, es guardar todos los archivos en el mismo directorio). Si no es así, al intentar usarla, el programa no encontrará el archivo donde está definida, y se lanzará un error.

6.8. Representación gráfica

Hay múltiples comandos en MATLAB® y en Octave que nos permiten realizar gráficas de funciones en dos y tres dimensiones, gráficas de dispersión de conjuntos de datos, y todo tipo de gráficos estadísticos. Veremos algunas de las posibilidades que hay, sobre todo para representar curvas y superficies definidas por funciones.

Comando plot Uno de los gráficos más básicos que podemos hacer, es un el de una línea, dadas las coordenadas de ciertos puntos sobre la línea. Esto se puede hacer con el comando `plot` para 2 dimensiones (coordenadas x e y), y con `plot3` para 3 dimensiones (x, y, z), con un uso casi idéntico. La forma más elemental es tener dos vectores, digamos `x` con las coordenadas x de los puntos que se quieren representar, y el vector `y` de la misma longitud con sus coordenadas y. La orden `plot(x,y)` dibuja en el plano xy las coordenadas de estos puntos, y las une mediante segmentos de recta con línea continua. La gráfica que se muestra, por tanto, es una línea poligonal. Cuando se utilizan muchos puntos, da la sensación de ser una línea continua, pero no lo es.

Este gráfico se puede personalizar de muchas formas, añadiendo parámetros opcionales a `plot`. Por ejemplo, si después de las coordenadas de los puntos añadimos `':'`, se dibujará una línea discontinua de puntos, en lugar de la línea continua que aparece por defecto, o poniendo `'or'` se marcarán los puntos dibujados con un pequeño circulo (`'o'`) de color rojo (`'r'`, de *red*, rojo en inglés).

```
x = [0, 1, 2, 3, 4, 5];          % Vector con las coordenadas x de los puntos
y = [1, 1.2, 1.5, 2, 1.4, 1];    % Vector con las coordenadas y de los puntos
plot(x,y)                        % Se dibuja y frente a x, uniendo los puntos
figure
plot(x,y,'or')                   % Cambiamos el marcador y el color
```

Al dibujar un gráfico en MATLAB® u Octave, se abre una nueva ventana que lo muestra (a menos que se haya elegido acoplar las ventanas de gráficos al entorno, usando la opción *Dock*). Si se realizan varios gráficos, cada uno sobrescribe al anterior, borrándolo, de forma que al final solo habrá una ventana abierta que mostrará el último de ellos. Si se desea dibujar cada gráfico en una ventana distinta, por ejemplo, para poder verlos a la vez y compararlos, hay que utilizar la orden `figure` entre un comando `plot` y el siguiente. Además, si se quieren superponer dos o

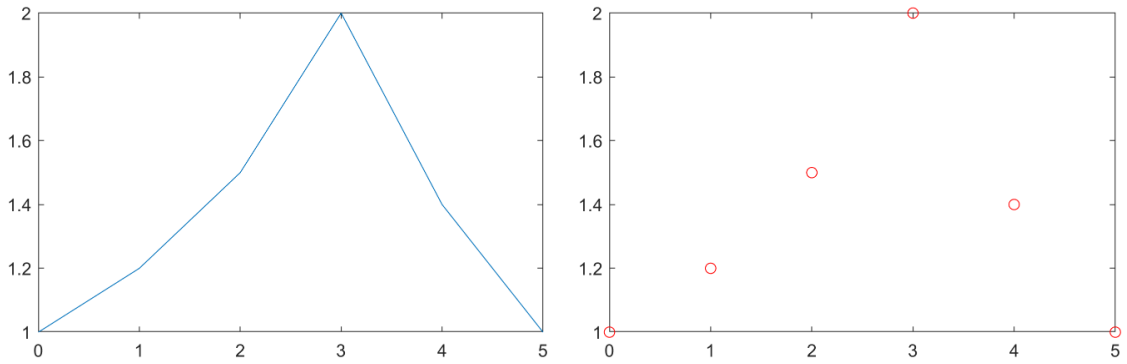

Figura 1.4: Ejemplos de gráficos generados usando el comando `plot`.

más gráficos para dibujarlos conjuntamente en los mismos ejes, hay que incluir `hold on` después del primer gráfico y `hold off` después del último. El uso de `figure` y `hold` es válido tanto para los gráficos hechos con `plot` como para los otros comandos que se verán más adelante.

El gráfico obtenido se puede hacer más informativo aplicando otros comandos posteriormente al `plot`, como por ejemplo con `xlabel` (etiqueta en el eje x), `ylabel` (etiqueta en el eje y), `title` (título del gráfico), `legend` (añade una leyenda), etc. Para una referencia más completa, se recomienda consultar la ayuda sobre la función `plot` y especialmente su parámetro `LineSpec`, que permite elegir tipos de línea, marcadores (símbolo con el que se representa cada punto) y colores, así como de las demás funciones relacionadas que se han mencionado.

Gráficas de funciones en 2D Para representar una función de una variable $f(x) : \mathbb{R} \to \mathbb{R}$, cuya gráfica se dibuja en un plano, hay dos estrategias principalmente. Una es utilizar `plot` con coordenadas de puntos que estén sobre la gráfica, y la otra es utilizar el comando `fplot`, específico para funciones.

Para hacerlo con `plot`, hay que generar un vector de coordenadas x en el rango donde se quiere dibujar la función. Si se desea dibujar en un intervalo $[a, b]$, se puede optar por generar un vector `x` de puntos equiespaciados en ese intervalo, usando `linspace` o el operador `::`. Ejemplo: `x = linspace(0, 1, 100)` generará 100 puntos equidistantes entre 0 y 1. Una vez hecho esto, se construye otro vector `y` con los valores de $f(x)$ sobre estos puntos. Si `f` ya existe como función (y suponiendo que se puede aplicar a vectores), podemos hacer simplemente `y = f(x)`, y si no, `y = ...` sustituyendo los puntos suspensivos por la expresión matemática de la función $f(x)$, que utilizará los valores almacenados en el vector `x`. Ejemplo: `y = 1/2*x.^2 + 5*x - 1` aplica la función $f(x) = \frac{x^2}{2} + 5x - 1$ (el punto `.` delante del operador potencia `^` permite que se pueda realizar esta operación elemento a elemento sobre el vector `x`; si no estuviera, se calcularía el cuadrado del vector, lanzando un error por incompatibilidad de dimensiones).

La segunda forma de dibujar la gráfica, directamente a partir de la función, es usar `fplot`. Esta orden tiene dos parámetros, la función `f` que se quiere dibujar y el intervalo en el cuál representarla, `I=[a,b]`, dado como un vector de dos elementos. Por ejemplo, con el comando: `fplot(@(x)sin(4*x),[0,2*pi])`. El segundo parámetro es opcional, y si se omite, la gráfica se hará en el intervalo por defecto, `[-5,5]`. La representación gráfica con `fplot` puede ser más conveniente que `plot` para funciones con asíntotas verticales, con discontinuidades, o que no estén definidas en alguna parte del dominio.

Esta función también permite dibujar curvas paramétricas con `fplot(fx,fy)` siendo `fx` y `fy` las componentes x e y de la curva, como funciones de un parámetro t. Además, existe

`fplot3` para curvas paramétricas en tres dimensiones. Las gráficas de este tipo también se pueden personalizar con opciones similares a las de `plot` (ver sección anterior).

Como ejemplo, dibujaremos la función `funcion1` que fue definida anteriormente (`funcion1 = @(x) cos(2*x)`), de las dos maneras explicadas:

```
x = linspace(0,2*pi,20);   % Generamos una partición en [0,2pi]
y = funcion1(x);           % Se evalúa la función en los puntos
plot(x,y)                  % Se representa la función con plot
figure
fplot(funcion1,[0,2*pi])   % Se representa la función con fplot
```

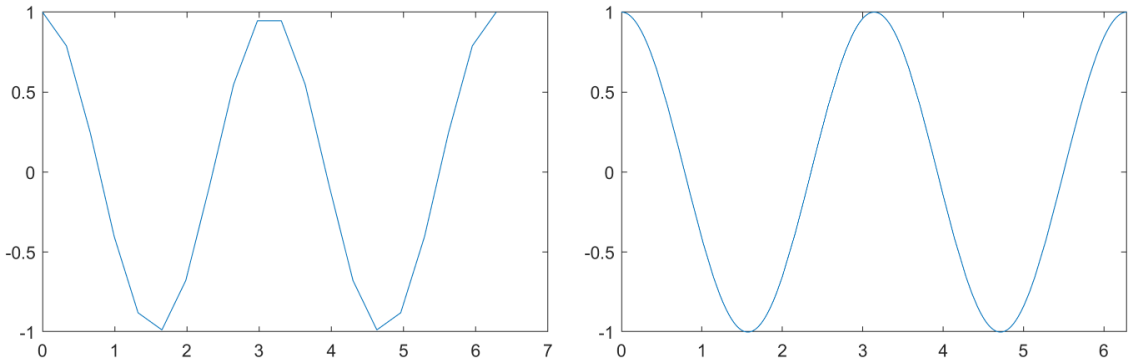

Figura 1.5: Gráficas de `funcion1` en $[0, 2\pi]$, generadas con `plot` (izquierda) y con `fplot` (derecha).

Para la primera gráfica (Figura 1.5, izquierda), deliberadamente se ha utilizado un vector `x` con muy pocos puntos (20). Al hacer esto, la gráfica, que siempre se construye como una poligonal uniendo esos puntos, da la sensación de no derivabilidad en los puntos de unión. En representaciones gráficas de tipo, es recomendable usar más puntos para obtener una representación más suave. Con usar 500 o 1000 puntos, generalmente es suficiente, y la gráfica se obtendrá muy rápido con estas cantidades. La segunda gráfica, con `fplot` (Figura 1.5, derecha), en realidad se construye internamente de forma similar, pero no tenemos control sobre cuántos puntos se están dibujando sobre la curva.

Gráficas de funciones en 3D Como se decía en los apartados previos, existen las variantes `plot3` y `fplot3` para poder representar curvas paramétricas en tres dimensiones. Su uso es totalmente análogo a las funciones `plot` y `fplot` para dos dimensiones ya explicadas. Así pues, nos centraremos en este punto en la representación gráfica de funciones de dos variables, $f(x, y) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, cuya gráfica, en 3D, es lo que llamamos una superficie en el espacio. Como en los casos previos, hay varias alternativas. La gráfica puede hacerse a partir de puntos de la superficie, usando `surf` o `mesh`, o directamente a partir de la expresión de la función, con `fsurf`.

Para usar `surf` o `mesh` (será la única opción cuando no esté disponible la expresión analítica de la función $f(x, y)$ a dibujar, sino solo sus valores), hay que construir lo que se denomina una malla o cuadrícula bidimensional, en el plano xy. Supongamos que se desea dibujar $f(x, y)$ en el dominio $A = I_x \times I_y$, siendo I_x e I_y intervalos para x e y respectivamente. Una malla en A se obtiene realizando una partición en I_x, otra en I_y, y haciendo el producto cartesiano de ambas, de forma que todas las coordenadas de x se combinan con todas las de y. Así, se obtiene una distribución de puntos sobre el dominio bidimensional A. Después, hay que evaluar la función $f(x, y)$ sobre estos puntos, y finalmente se hace la gráfica usando el comando correspondiente. Veamos un ejemplo, para dibujar la gráfica de `funcion2` definida anteriormente

(`funcion2 = @(x,y) -cos(y)*sin(x)`):

```
[X,Y] = meshgrid(linspace(-pi,pi,50));   % Se genera la partición bidimensional
Z = funcion2(X,Y);                       % Se evalúa la función
surf(X,Y,Z)                              % Se representa como superficie continua
figure
mesh(X,Y,Z)                              % Se representa como una red o malla
```

La primera línea de este bloque genera una partición bidimensional, con valores de x e y en el intervalo $[-\pi, \pi]$ (ambas). El comando `meshgrid` permite elegir intervalos distintos, usándolo dos dos parámetros en lugar de solo uno. Para superficies, el uso de un número excesivo de puntos puede ser contraproducente. En este caso se han utilizado 50 por variable, que son 2500 en total. Cantidades de puntos mucho mayores podrían dar problemas de rendimiento o de visualización. Las dos gráficas generadas al ejecutar este código se muestran en la Figura 1.6.

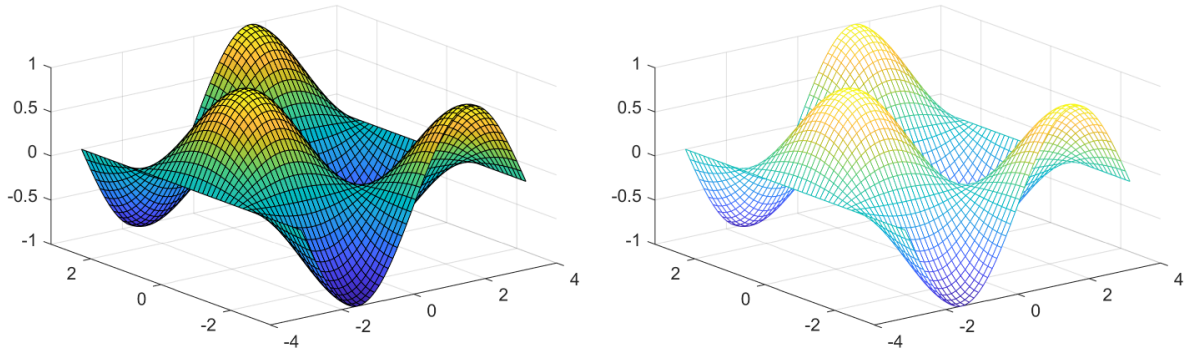

Figura 1.6: Gráficas de `funcion2`, $f(x, y) = -\cos(y)\sin(x)$, en $[-\pi, \pi] \times [-\pi, \pi]$, generadas con `surf` (izquierda) y `mesh` (derecha).

El uso del comando `fsurf`, que se aplica directamente a la función, es muy similar al de `fplot` visto anteriormente. En este caso, la función debe tener dos variables, y opcionalmente se puede indicar el dominio en el cuál se quiere hacer la gráfica. Por defecto, se usa $[-5, 5] \times [-5, 5]$. Por ejemplo, `fplot(funcion2)` produce una gráfica similar a la representada en la Figura 1.6 (izquierda), solo que con un dominio distinto. Por otro lado, el comando `fsurf(fx,fy,fz)` también permite dibujar una superficie paramétrica, dada a partir de sus tres componentes `fx`, `fy` y `fz`, que se asumen funciones de dos parámetros u y v.

Hemos visto algunas de los comandos elementales de MATLAB® y Octave para hacer representaciones gráficas, especialmente de funciones, pero hay muchos otros, más especializados o para otro tipo de gráficos. Como siempre, insistimos en la consulta de la documentación de estos entornos, especialmente la de MATLAB® , para una información más completa.

6.9. Operadores lógicos

Gran parte de las tareas de programación, también en matemáticas, consisten en realizar ciertas operaciones u otras según se cumpla o no alguna condición. En este apartado veremos cómo utilizar variables lógicas, también llamadas booleanas o binarias (su valor es 1 (verdadero, *true*) ó 0 (falso, *false*)), y como operar con ellas para construir condiciones lógicas complejas a partir de otras elementales (lo que formalmente se conoce en matemáticas y en computación como álgebra de Boole).

En MATLAB® y Octave, podemos definir una variable lógica asignándole un valor `true/false`: `a = true` o bien `a = false`. Sin embargo, estas generalmente se obtendrán como resultado de una operación de comparación de igualdad, de orden, u otras más complejas. Por ejemplo, podemos comprobar si dos variables son idénticas con el operador de igualdad `==`, o si son distintas con el operador "distinto de" `~=`, que dará justamente el resultado opuesto al anterior. Ejemplos: `2 == 1` dará como resultado un 0 lógico (falso), mientras que `pi ~= 3` dará un 1 lógico (verdadero). También se puede comparar el valor de dos números mediante los operadores `<`, `<=`, `>` y `>=`, cuya interpretación es obvia. La condición lógica contraria a una dada se construye anteponiendo el operador `~` [3] (operador negación, NO lógico, o NOT) a la condición. Por ejemplo: `~(1<0)` dará un 1 lógico (verdadero), pues su condición contraria, `1<0`, es falsa.

Para comprobar si se cumplen simultáneamente dos condiciones podemos utilizar el operador conjunción, Y lógico o AND. Este operador en MATLAB® y Octave se utiliza con los símbolos `&` o `&&` (llamado en español *et* o simplemente *y*, y en inglés, *ampersand*), y se escribe con ⇧ (Mayúsculas o *Shift*) + 6. El resultado de esta operación es verdadero solamente si ambas condiciones son verdaderas (o lo que es lo mismo, el resultado será falso si cualquiera de las condiciones es falsa).

De la misma forma, podemos comprobar si se cumple alguna de entre dos condiciones con el operador disyunción, O lógico u OR. Este operador se emplea con los símbolos | ó || (llamado en español pleca o barra vertical, y en inglés, *vertical bar* o *pipe*), y se escribe con AltGr+1. El resultado será verdadero siempre que cualquiera de las condiciones sea verdadera (o dicho de otra forma, el resultado solo será falso si todas las condiciones son falsas).

Veamos algunos ejemplos:

```
x = 5.66;        % Asignamos un valor numérico en x
x == 5           % Comprobamos si x es igual a 5
x < 6            % Comprobamos si x menor que 6
x>2 && x < 6     % Comprobamos si x menor que 6 y mayor que 2
~(x<2)           % Comprobamos si x NO es menor que 2 (equivalente a si es >=2)
```

```
ans =
    logical
    0
ans =
    logical
    1
ans =
    logical
    1
ans =
    logical
    1
```

Lo que significa que, evidentemente, la primera comprobación es falsa (0 lógico), mientras que las demás sí se satisfacen, son verdaderas (1 lógico).

Se pueden construir condiciones lógicas más complejas, con más de 2 condiciones lógicas elementales, combinando varios de los operadores anteriores, e incluyendo paréntesis si fuera necesario. Ejemplo: `(cond1 || cond2) && cond3`. Los operadores lógicos, además, se pueden

[3] Recuerda que el símbolo ~ (virgulilla) se escribe en Windows con las combinaciones de teclas AltGr+4 y después Espacio, o bien Alt+1 2 6 en el teclado numérico.

utilizar sobre vectores o matrices, y se comprobarán las condiciones elemento a elemento, excepto los operadores AND y OR dobles, por el motivo que a continuación se explica. En estos casos, el resultado será un vector o matriz lógico del mismo tamaño que la variable que se haya usado en la comparación.

Ampliación: Operadores cortocircuito

Los operadores anteriores (AND y OR) simples `&` y |, y los dobles `&&` y ||, son en muchas ocasiones equivalentes, pero con un importante matiz que los diferencia. Los dobles son lo que se llama operadores con cortocircuito *short-circuit*. Esto significa que, cuando la primera condición determina el resultado, la segunda no se evalúa. Por ejemplo, en la expresión `condA && condB`, si `condA` fuera falso, el resultado de la operación será falso, independientemente del valor de `condB`. Al haber usado el operador doble, con cortocircuito, no se ha llegado a ejecutar `condB` y comprobar si se cumple o no. Algo análogo pasa con `condA || condB`, si `condA` fuera verdadera. Para usar estos operadores con cortocircuito, cada una de las condiciones elementales (`condA` y `condB`) debe dar como resultado un escalar (un sólo valor lógico), y por tanto no son aptas para ser usadas sobre elementos de vectores o matrices.

6.10. Estructuras condicionales y bucles

Tras el apartado anterior, una vez que hemos visto cómo comprobar si se cumple o no una determinada condición lógica, elemental o compuesta, vamos a ver qué estructuras condicionales (ejecutar una parte del código cuando se cumpla una condición) y repetitivas o bucles (ejecutar una parte del código repetidas veces) se pueden usar en MATLAB® y Octave.

Estructuras `if` e `if-else` Por estructura condicional entenderemos una parte del código que se ejecutará solamente cuando se cumpla cierta condición. Esto se puede hacer con los bloques `if-end`, cuya estructura es:

Estructura if-end

```
if condicion == true      % Se comprueba si la condición lógica es verdadera o no
    instruccionesV;       % Si es verdadera, se ejecutan estas instrucciones
end                       % Indica el final de la estructura condicional
```

También se puede dar el caso de que queramos hacer acciones distintas, según sea verdadera o no cierta condición. En este caso, se usará el bloque `if-else-end`:

Estructura if-else-end

```
if condicion == true     % Se comprueba si la condición lógica es verdadera o no
    instruccionesV;      % Si la condición es verdadera, se ejecuta esto
else
    instruccionesF;      % Si la condición es falsa, se ejecuta esto
end                      % Indica el final de la estructura condicional
```

Veamos un ejemplo:

Ejemplo de estructura if

```
a = 50;
if (a>=5 & a<15)      % ¿Se cumple la condición lógica?
    x = 3*a;          % Se ejecuta solo si la condición es verdadera
else
    x = a/10;         % Se ejecuta solo si la condición es falsa
end
x
```

En este ejemplo, observamos que `a = 50`. En la estructura `if` (línea 2), se comprueba si la condición lógica `a>=5 & a<15` es verdadera o no. En este caso, vemos que no: `a` es mayor o igual a 6, pero no es menor que 15. Como el conector lógico es `&` (operación lógica 'y' o 'AND'), la condición compuesta es falsa. Entonces, no se ejecutará el código dentro del bloque `if` (línea 3), sino el que hay en el bloque `else` (línea 5): se divide `a` por 10, y se guarda el resultado en `x`. Por tanto, al finalizar la estructura `if-else`, el valor de `x` es 5.

Ampliación: Otras estructuras condicionales

Si se quieren comprobar varias condiciones lógicas en cascada, en lugar de anidar varios bloques `if` (lo cual es posible), se puede utilizar uno o más bloques `elseif`. Estos se intercalan entre `if` y `else`, con lo que la estructura en realidad será del tipo `if-elseif-elseif-...-end`.

Si se quieren realizar distintas acciones (normalmente más de 2 o 3), según sea el valor de una variable (no necesariamente numérica), es posible usar `switch`, que generalmente equivale a utilizar varios `if-end` anidados o `if-elseif-end`.

Como siempre, remitimos al lector a la documentación de MATLAB® y Octave para más detalles sobre estas estructuras.

Bucles `for` y `while` Una estructura repetitiva o bucle es una parte del código que se ejecuta un cierto número de veces. En ocasiones, este número de repeticiones está fijado a priori, y otras veces se va a estar repitiendo ese código siempre mientras que cierta condición se satisfaga. En el primer caso, lo más habitual es utilizar un bucle `for`, y en el segundo, un bucle `while`. Cada una de las ejecuciones del código contenido en un bucle se llama iteración. Veamos la estructura de cada tipo de bucle.

Estructura del bucle for

```
for contador = rango  % Para cada valor posible del contador en el rango (vector)
    instrucciones;    % se ejecutan las instrucciones dentro del bucle
end
```

En el bucle `for`, se van a ejecutar las `instrucciones` dentro del bucle para todos los valores posibles de `contador` dentro del `rango` especificado (un vector de valores que tomará el `contador`). Típicamente, las `instrucciones` dependen del valor numérico del `contador`, de forma que no se realizará exactamente la misma operación siempre, sino operaciones similares. En muchos casos, `rango` se puede sustituir por `1:N`, para cierto valor de `N` natural, de forma que se ejecutarán las instrucciones internas del bucle, primero para `contador=1`, después para `contador=2`, etc., hasta llegar a `contador=N` (inclusive).

Estructura del bucle while

```
while condicion            % Mientras que la condición lógica sea verdadera
    instrucciones;         % se ejecutan las instrucciones dentro del bucle
    actualizarCondicion;   % Importante: actualizar la condición lógica
end
```

En un bucle `while`, lo primero que se hace es comprobar si la `condicion` lógica se cumple o no (si es necesario, hay que inicializarla previamente al bucle). En caso de que la `condicion` sea verdadera, se ejecutarán las instrucciones dentro del bucle. Es importante que, entre estas instrucciones, haya alguna que actualice el valor de la `condicion`, para evitar bucles infinitos. Una vez ejecutado el código interior, se vuelve a comprobar si la `condicion` es verdadera o no, y se sigue ejecutando el bucle en caso afirmativo. La ejecución terminará en el momento en que `condicion` sea falsa.

Dentro de ambos tipos de bucles, `for` y `while`, se pueden utilizar algunas órdenes específicas para modificar el comportamiento de los bucles, como `break`, que corta por completo la ejecución del bucle (y se seguirá ejecutando el código que haya después del bucle), y `continue`, que lo que hace es avanzar a la siguiente iteración del bucle, dejando sin ejecutar el código interno del bucle que haya a partir de dicha orden (la ejecución del código continuará todavía en el bucle, comprobando la condición de continuidad, y aplicando las instrucciones internas si es necesario).

Veamos un ejemplo con ambos tipos de bucles:

Ejemplos de bucles

```
s = 0;
n = 1;
while (n<=5)
    s = s + n;
    n = n+1;
end
s

s = 0;
for n=1:5
    s = s + n;
end
s
```

Con ambos bucles se hace la misma operación, que es obtener, en la variable `s`, la suma de los primeros 5 números naturales. Son equivalentes. En ambos casos, se obtiene `s = 15` tras cada bucle.

En las líneas 1-7, el cálculo se hace con un bucle `while`. Primero, se inicializa la variable `s`, que será la suma, a 0. Iremos sumando en `s` cada número natural, desde 1 hasta 5. Antes del bucle, también se inicializa la variable `n` (línea 2) que aparece en la condición de continuidad `(n<=5)` del bucle (línea 3). En el bucle, se comprueba dicha condición, que inicialmente es verdadera. Por tanto, se ejecuta el código interno del bucle (líneas 4 y 5). Al hacer esto, ahora `s = 1` y `n=2`. Se comprueba de nuevo la condición de continuidad (línea 3) y se repite el proceso, iteración a iteración. Al final de la 5ª iteración, se tiene `n = 6` y el bucle `while` finaliza. En es momento, se muestra `s` (línea 7) y vemos que `s = 15`.

En las líneas 9-13, hacemos lo mismo con un bucle `for`. Igual que antes, se inicializa `s` (la suma) a cero (línea 9). En el bucle `for`, se aplican las operaciones internas (línea 11) para

todos los valores de `n` (variable contador o índice) especificados (línea 10). En este caso, al poner `n=1:5`, se ejecutará para los valores `n=1`, `n=2`, ..., y `n=5`. Para cada uno de ellos, se sumará `n` a `s`, cuyo valor se actualiza. Tras aplicar las 5 iteraciones, el bucle `for` finaliza, se muestra el resultado (línea 13), y comprobamos que también ahora `s = 15`.

6.11. Otros comandos útiles

Tanto en MATLAB® como en Octave hay una serie de comandos que pueden ser útiles para facilitar la escritura y depuración de los programas en desarrollo. Se trata de comandos para realizar ciertas modificaciones en el entorno de programación: eliminar cálculos o resultados previos, cambiar el formato de los resultados, etc. Mencionamos algunos de los más importantes:

- `clear`: Este comando permite eliminar del *Workspace* variables o funciones específicas utilizadas previamente, mediante `clear nombre_variable`. También es posible borrar todo el contenido del *Workspace*, con `clear all` (se eliminarán todas las variables y funciones cargadas, y el *Workspace* quedará vacío).
  Aunque algunos desarrolladores, y el propio MATLAB®, recomiendan evitar este comando, en opinión del autor es útil ponerlo al principio de los scripts, sobre todo en el ámbito docente. Un error muy frecuente entre usuarios noveles se produce al ejecutar un script con ciertas variables, cambiar el nombre, quitar o poner alguna, y volver a ejecutarlo. O bien, se ejecuta un script, que utiliza unas variables, y luego otro distinto, con otras. En estas ocasiones, en la segunda ejecución puede que no se produzcan errores de ejecución, pero en realidad se esté utilizando alguna variable que no se ha definido en el propio script, sino en una versión anterior o en otro distinto, y que cuando fue utilizada se almacenó en en el *Workspace* y aún persiste. Si se incluye `clear all` al principio de un script, o al menos `clear variables` o `clearvars`, y la ejecución del script funciona, se puede estar seguro de que el script contiene todas las variables necesarias, y no se están utilizando otras variables residuales. Así se garantiza que cada script está autocontenido.

- `clc`: Esta orden limpia todos el contenido de la *Command window*, permitiendo dejarla vacía antes de una nueva ejecución de un script. Cuando se muestran muchos resultados por pantalla (práctica que se debería evitar, terminando con `;` las líneas cuyos resultados sea innecesario mostrar), esto puede facilitar la búsqueda de algún resultado específico.

- `close all`: Este comando sirve para cerrar todas las ventanas de gráficas que haya abiertas. Es usual que en un script se realicen varias gráficas distintas, usando la orden `figure`. Si un script genera múltiples gráficas, y además se ejecuta varias veces, el número de ventanas de gráficas puede ser enorme. Utilizando `close all` podemos cerrar todas, y al volver a ejecutar un script sólo veremos las nuevas. También puede servir para cerrar una o varias gráficas específicas, cambiando `all` por sus identificadores.

- `format`: Esta orden, con varias variantes, permite cambiar la forma en que se muestran los resultados numéricos. Por defecto, se muestran en `format short`, con 4 dígitos decimales. Con `format long` se pueden ver hasta 15 dígitos decimales. Otra opción, útil en ocasiones, es usar `format rat` o `format rational`, que mostrará los resultados como fracciones de números enteros pequeños. Hay otras opciones que se pueden consultar en la ayuda de esta orden. Poniendo solamente `format` se puede volver al formato "corto" por defecto.

- Borrar el historial de comandos o *Command History*: El historial de comandos guarda un listado de todos los comandos que se van ejecuando, incluso entre distintas sesiones (después de cerrar y reabrir el programa). A veces interesar borrar todo el historial, para

facilitar la búsqueda de nuevos comandos o por diversos motivos.
En MATLAB®, desde la versión 7.0 (R14), esto se puede hacer con el comando:
`com.mathworks.mlservices.MLCommandHistoryServices.removeAll`
En Octave es más sencillo, basta con escribir: `history -c`

Por los motivos expuestos en los puntos anteriores, para un usuario en fase inicial de su aprendizaje de MATLAB® u Octave, el autor recomienda que se incluyan los siguientes comandos al principio de cada script:

```
clear all
close all
clc
```

7. Recursos de aprendizaje adicionales

Como complemento a la introducción presentada en este documento se sugiere la realización de algún curso online autoguiado sobre MATLAB®. Además de los citados abajo, hay otros muchos cursos, gratuitos y de pago, disponibles en distintas plataformas de aprendizaje online (Coursera, EdX, etc.) impartidos por universidades de todo el mundo. Se recomiendan los siguientes cursos, que están disponibles de forma gratuita en "MATLAB Academy" (`https://matlabacademy.mathworks.com`), una plataforma web para aprendizaje de MathWorks®:

- Curso *MATLAB Onramp*. Se presenta una introducción elemental y las nociones básicas de MATLAB. Duración aproximada de 2 horas. Disponible en:
  `https://matlabacademy.mathworks.com/es/details/matlab-onramp/gettingstarted`
- Ruta de aprendizaje *Programación en MATLAB* (varios cursos). Duración aproximada de 4 horas. Disponible en:
  `https://matlabacademy.mathworks.com/es/details/programming-in-matlab/lpmlprm`
- Ruta de aprendizaje *Desarrollo de Dominio de MATLAB* (varios cursos). Duración aproximada de 12 horas. Disponible en:
  `https://matlabacademy.mathworks.com/es/details/build-matlab-proficiency/lpmlbmp`