Tema 4

Visualización Científica

La visualización científica es una herramienta fundamental para comunicar y comprender conjuntos de datos. Permite transformar información abstracta en imágenes, gráficos y animaciones que son más fáciles de interpretar. Una adecuada visualización científica permite mejorar la comprensión del problema estudiado ya que las imágenes y gráficos son más intuitivos que los datos numéricos, lo que facilita la comprensión de conceptos complejos. Además puede ayudar a identificar patrones y tendencias que no son evidentes en los datos.

Para visualizar datos provenientes de funciones o datos experimentales, MATLAB puede trabajar de dos maneras diferentes, que pueden combinarse entre si. La primera de ellas es recibiendo los valores numéricos contenidos en uno, dos o más vectores (o matrices) para posteriormente representarlos. La segunda manera consiste en definir, implícita o explicitamente, funciones y pedir a MATLAB que nos represente su valor en los intervalos que queramos. Así, en función del problema a tratar, se deberá usar un enfoque o el otro.

Tras el estudio de este tema el alumno debe familiarizarse con el uso de MATLAB para representaciones gráficas en dos y tres dimensiones, usando los comandos más habituales:

- Gráficas en 1D y 2D:
  1. Representación numérica en 1D: `plot(x,opciones), hist(x), etc.`
  2. Representación numérica en 2D: `plot(x,y,opciones), plot(x1, y1, x2, y2, x3, y3, opciones), scatter(x,y), stairs(x, y), etc.`
  3. Representación de funciones: `fplot(función,[límite inferior, límite superior], opciones), fimplicit(función,[límite inferior, límite superior], opciones), etc..`
- Gráficas en 3D:
  1. Representación numérica: `plot3(x,y,z,opciones), mesh(x,y,z), surf(x,y,z), contour(x,y,z), etc.`
  2. Representación de funciones: `fplot3(f, g, h, [a,b]), fmesh(f), fcontour(f), etc.`

Cada uno de estos comandos tiene multitud de opciones para representar gráficas de distintos tipos. Consulte la ayuda de MATLAB y los cursos virtuales disponibles para explorar las

posibilidades de estas funciones. En muchos casos querremos incluir las gráficas generadas en una sesión de cálculo en algún documento sin pérdida de calidad de la imagen, para ello es fundamental saber cómo guardar gráficas en archivos, para poder importarlas posteriormente con el procesador de textos que estemos usando.

El objetivo final de esta unidad didáctica es que el estudiante sea capaz de representar cualquier conjunto de datos, mediante la visualización más adecuada, en gráficas **autoexplicativas** para su posterior presentación en informes o presentaciones.

4.1. La ventana de visualización de MATLAB

Aunque MATLAB puede trabajar con los datos y guardar figuras como archivos sin representarlas previamente (con el modo ”invisible”, propiedades: ’’ `Visible`’’, ’’`off`’’), normalmente trabajaremos con el entorno de ventanas que proporciona para su visualización.
Para abrir estas ventanas, se debe ejecutar el comando ≫ `figure`, que abre una nueva ventana gráfica, `Figure n`, donde n es el primer número entero disponible. Se puede trabajar directamente sobre una ventana especificada, usando ≫ `figure(n)`, por ejemplo ≫ `figure(3)` para trabajar sobre la ventana número 3.
Además se puede asignar una variable a la ventana, como se muestra en la figura 4.1. Esto permitirá modificar después sus propiedades, añadir nuevas líneas a la gráfica, añadir leyendas o ajustar sus ejes, entre otras muchas posibilidades.

```
>> window=figure(2)

window =

  Figure (2) with properties:

      Number: 2
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [1000 918 560 420]
       Units: 'pixels'

  Show all properties
```

Figura 4.1: Asignación a una variable tipo ”Figure” de la ventana gráfica número 2.

Dado que puede haber varias ventanas abiertas de manera simultanea, la manera de designar cuál es la activa (en cuál se dibujará) es usar la orden ≫ `figure(n)`, usando el valor `n` correspondiente. De manera análoga, se puede usar ≫ `close(n)` para cerrar la figura `n`. ≫ `close all` cierra todas las figuras abiertas en la sesión. Además un comando muy interesante es ≫ `clf(n)`, que borra el contenido de la figura (n) sin cerrar la ventana para poder comenzar un nuevo gráfico. Si no usamos el entero `n`, podemos usar ≫ `clf`, que borra la ventana activa.

MATLAB permite representar varias gráficas en una misma ventana, para ello podemos usar el comando ≫ `subplot(n,m,k)`, que subdivide la ventana en una malla rectangular, donde `n` especifica el número de filas, `m` especifica el número de columnas y `k` especifica el elemento seleccionado. Es importante destacar que se pueden usar varias posiciones a la vez. Por ejemplo, el código

```
≫ subplot(3,3,1:2);
≫ subplot(3,3,4);
≫ subplot(3,3,[6,9]);
```

crea una ventana con 9 posibles gráficas en su interior (3 filas y 3 columnas). Inicialmente se asignan los espacios *1* y 2 a la primera gráfica, el 4 a la segunda gráfica y los elementos 6 y 9 de la malla a la tercera gráfica. La ventana resultante se puede observar en la figura 4.2

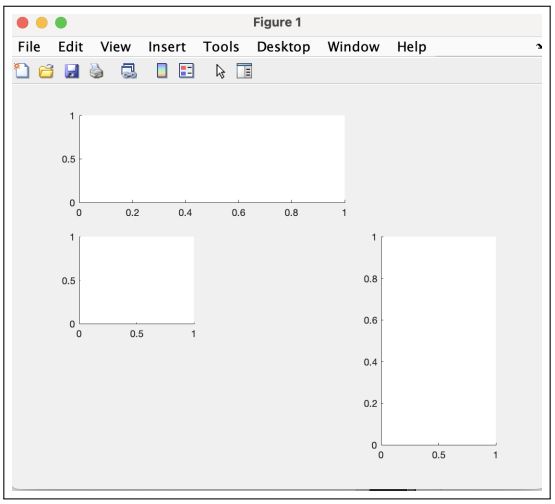

Figura 4.2: División de la ventana gráfica mediante `subplot`.

Hay que tener en cuenta que MATLAB numera los cuadros de izquierda a derecha y de arriba hacia abajo.

4.2. Realización de ráficas con MATLAB

Para crear gráficas de líneas en 2 dimensiones, a partir de datos numéricos, se debe usar la función ≫ `plot()`, que es una de las más importantes y potentes de MATLAB. Esta función dibuja en la figura activa (o abre una nueva) el vector numérico que deseemos. Por ejemplo, para representar el valor del coseno de x en el intervalo de 0 a 2π se deberá ejecutar el siguiente código:

```
≫ x=linspace(0, 2*pi, 10); %Genero un vector de 10 elementos en la variable 'x'.
≫ y=cos(x); %Calculo el coseno de cada elemento del vector 'x'.
≫ plot(x,y) %Dibujo x e y en una gráfica
```

Sin embargo, hay que tener en cuenta que en este caso hemos definido la variable `x` usando solamente 10 puntos, por lo que la curva resultante no será suave. Esto se debe a que el ordenador representa solamente los puntos que le asignamos, a menos que le especifiquemos la precisión con la que queremos trabajar. Así, cualquier gráfica de una curva en un ordenador es simplemente una línea poligonal que une un **conjunto discreto y ordenado de puntos**. Si queremos que nuestra representación se aproxime a la **representación continua**, con la que hemos trabajado más a menudo, deberemos aumentar el número de puntos en los que calcular la función que queramos representar. Obviamente, cuantos más puntos usemos, peor será el rendimiento del ordenador, así que **siempre se debe hacer un balance entre resolución y rendimiento**. En el siguiente código se puede ver como el número de puntos influye

de manera determinante en la visualización (Figura 4.3-A, en azul se define la variable `x`, para evaluar el coseno, con `10` puntos y en rojo con `100`):

```
≫ x1=linspace(0, 2*pi, 10);%Genero un vector de 10 elementos en la variable 'x1'.
≫ x2=linspace(0, 2*pi, 100);%Genero un vector de 100 elementos en la variable 'x2'.
≫ y1=cos(x1);%Calculo el coseno de cada elemento del vector 'x1'.
≫ y2=cos(x2);%Calculo el coseno de cada elemento del vector 'x2'.
≫ plot(x1,y1,x2,y2)%Dibujo x1 e y1, junto a x2 e y2 en una gráfica
```

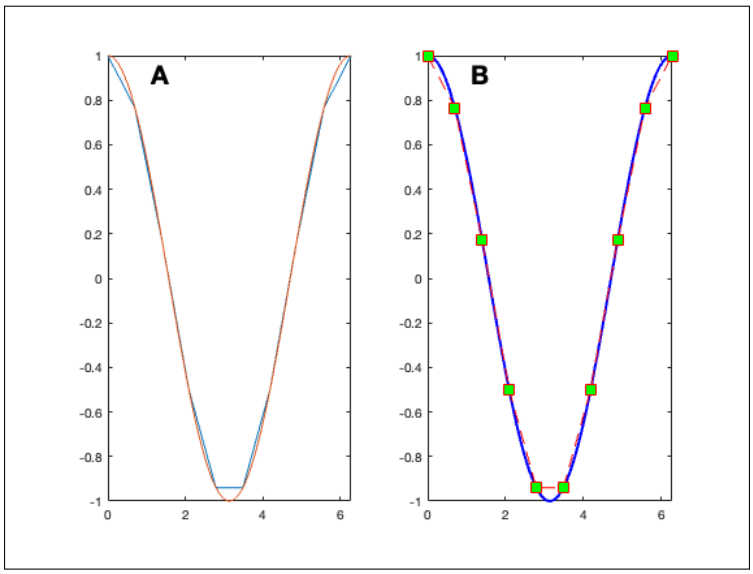

Figura 4.3: Efecto del número de puntos empleados en representar el coseno de `x`.

El comando `plot` admite un argumento opcional, en forma de cadena de caracteres, que se puede añadir a cada par de pareja de datos x, y. Este argumento permite seleccionar múltiples aspectos relacionados con la visualización, como el color o el tipo de la línea o los símbolos a representar. Cuando se dibujan varias curvas en una misma gráfica es indispensable representarlas de manera independiente para poder identificarlas rápidamente. Veamos como podemos usar estos argumentos para mejorar la gráfica anterior:

```
≫ x1=linspace(0, 2*pi, 10);%Genero un vector de 10 elementos en la variable 'x1'.
≫ x2=linspace(0, 2*pi, 100);%Genero un vector de 100 elementos en la variable 'x2'.
≫ y1=cos(x1);%Calculo el coseno de cada elemento del vector 'x1'.
≫ y2=cos(x2);%Calculo el coseno de cada elemento del vector 'x2'.
≫ plot(x2,y2,'-b','LineWidth',2);%Dibujo y2 frente a x2.
≫ hold on;%Congelo la gráfica.
≫ plot(x1,y1,'-rs','MarkerFaceColor','g','MarkerSize',10);%Dibujo y1 frente a x1
```

Como se puede observar en la Figura 4.3-B, se ha representado `y2` frente a `x2` usando una línea azul de tamaño `2`, mientras que se ha dibujado `y1` frente a `x1`, usando una curva roja discontinua y unos marcadores verdes tamaño `10`. Cabe destacar que, en el código anterior, hemos usado el comando `hold`, que nos permite congelar (`hold on`) o descongelar (`hold off`) la figura en la que estemos trabajando para añadir representaciones adicionales. Si no hubiéramos congelado la figura, el nuevo plot habría remplazado completamente al anterior.

Existe documentación extensiva sobre la función `plot` en el centro de ayuda de MATLAB `https://es.mathworks.com/help/matlab/ref/plot.html`.

Además del comando `plot`, MATLAB tiene muchas herramientas que permiten representar los datos de la manera más apropiada. Entre ellos, caben destacar:

- `scatter(x,y)`: Representa un diagrama de dispersión de los datos (x, y).
- `histogram(x)`: Representa el histograma de una variable (x).
- `stairs(x)`: Representa el gráfico de escaleras de la variable seleccionada (x).
- `bar(x)`: Representa la gráfica de barras de la variable seleccionada (x).
- `barh(x)`: Representa la gráfica de barras horizontal de la variable seleccionada (x).
- `stem(x)`: Representa la variable seleccionada (x) de manera discreta.

Todas estas funciones trabajan con datos 1D o 2D y son muy fáciles de personalizar. Por ejemplo, usemos la función coseno vista anteriormente, evaluada sobre 100 puntos en el intervalo de $[0, 2\pi]$, definido como: `x=linspace(0, 2*pi, 100);`. En este caso, vamos a añadir algo de ruido aleatorio al cálculo del coseno: `y= cos(x)+ rand(1,100);`. De esta manera sumamos al valor del coseno un numero aleatorio en el intervalo de $[0, 1]$ (ver generación de números aleatorios: $\gg$ `help rand`). Alguna de las diferentes representaciones posibles pueden verse en la Figura 4.4.

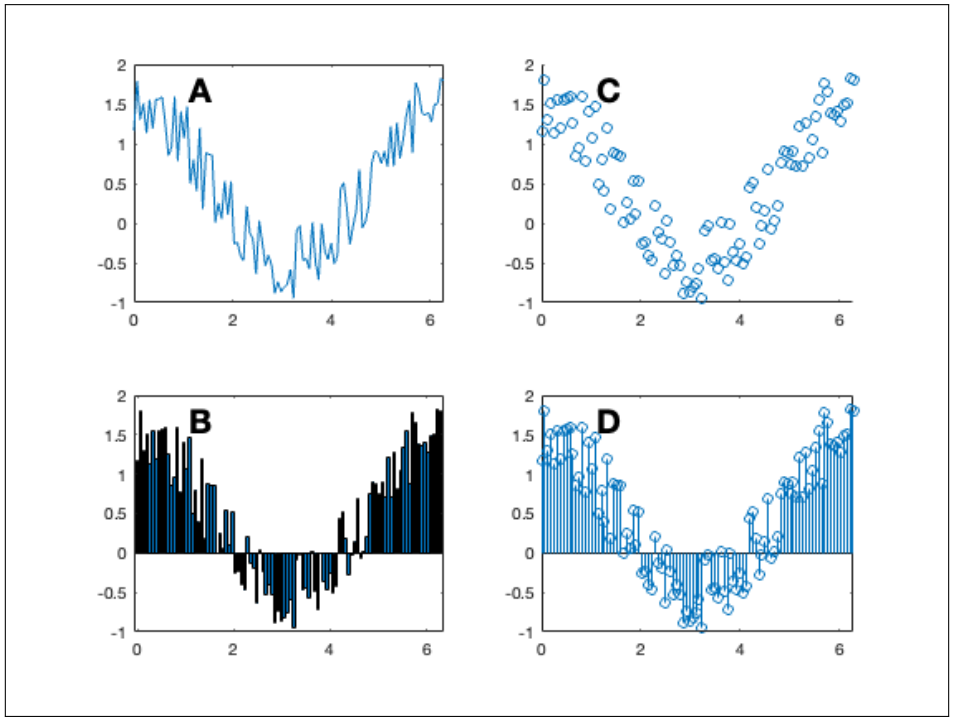

Figura 4.4: Diferentes representaciones del coseno en el intervalo de $[0, 2\pi]$ sumando ruido aleatorio. En el **panel A** se ha usado la función `plot(x,y)`, en el **panel C** la función `scatter(x,y)`, en el **panel B** la función `bar(x,y)` y, en el **panel D** la función `stem(x,y)`.

Además de usar coordenadas cartesianas para representar datos, MATLAB permite también su representación usando coordenadas polares. Para ello, se utilizan comandos equivalentes a los vistos anteriormente con la palabra ”polar” delante:

- `polarplot(theta,r)`: Representa una línea en coordenadas polares usando los datos de los vectores (θ, r).
- `polarscatter(theta,r)`: Representa un diagrama de dispersión de los datos de los vectores(θ, r) en coordenadas polares.
- `polarhistogram(theta)`: Representa el histograma de una variable (θ) en coordenadas polares.
- etc...

En la figura 4.5, se muestran diferentes representaciones en coordenadas polares usando como ángulo θ, `theta = linspace(0,10*pi,200);` y, de izquierda a derecha y de arriba a abajo:

1. `rho= ones(size(theta)); polarplot(theta,r);` % Circunferencia de radio, $\rho = 1$.
2. `rho= theta; polarplot(theta,r);` % Espiral de radio $\rho = \theta$.
3. `rho= theta.*sin(theta); polarplot(theta,r);` % Serie de circunferencias concéntricas.
4. `rho=sin(0.5*theta)./theta; polarplot(theta,r);` % Una flor con su tallo.

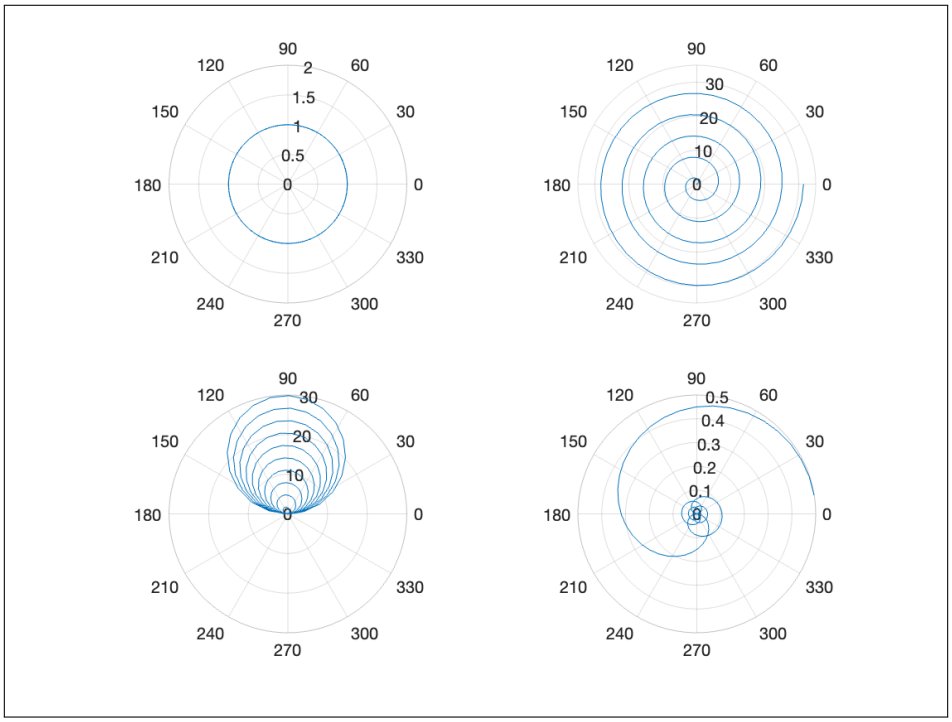

Figura 4.5: Diferentes representaciones usando coordenadas polares.

4.2.1. Gráficas de Funciones en 1D y en 2D con MATLAB

Para dibujar una curva definida por una función, $f(x)$ de una variable real dada por una expresión analítica, se puede usar como alternativa a `plot` la función `fplot`. Esta función representa la evaluación de la función indicada como primer argumento usando por defecto el intervalo $[-5, 5]$.

Para definir la función que queremos representar, podemos usar una sintaxis especial:

```
@(x) expresión
```

Por ejemplo, si quisieramos representar la función $sin(x)$, deberíamos escribir: `f= @(x) sin(x); fplot(f);` o directamente, `fplot(@(x) sin(x));`

Aunque por defecto, el comando `flpot` se aplica en el intervalo $[-5, 5]$, MATLAB permite, lógicamente, elegir el intervalo de representación.

Así, si escribimos: `f = @(x) sin(exp(x)); fplot(f)` representaremos la función $sin(e^x)$ en el intervalo $[-5, 5]$ (Figura 4.6-A), mientras que si escribimos: `f = fplot(f, [-2,2])`, obtendremos la misma función en el intervalo $[-2, 2]$ (Figura 4.6-B).
Una de las ventajas de `fplot` es que es suficientemente inteligente como para solamente representar aquellos valores en los que la función está definida. Por ejemplo, si queremos representar el logaritmo de x, podremos escribir directamente `fplot(@(x) log(x));` que mostrará la gráfica de la función únicamente donde está definida, esto es para $x > 0$ (Figura 4.6-C). Al igual que `plot`, cuando usamos `fplot` se pueden añadir "capas" al gráfico congelándolo con `hold on`; por ejemplo `fplot(@(x) sin(x)); hold on; fplot(@(x) cos(x))` (Figura 4.6-D).

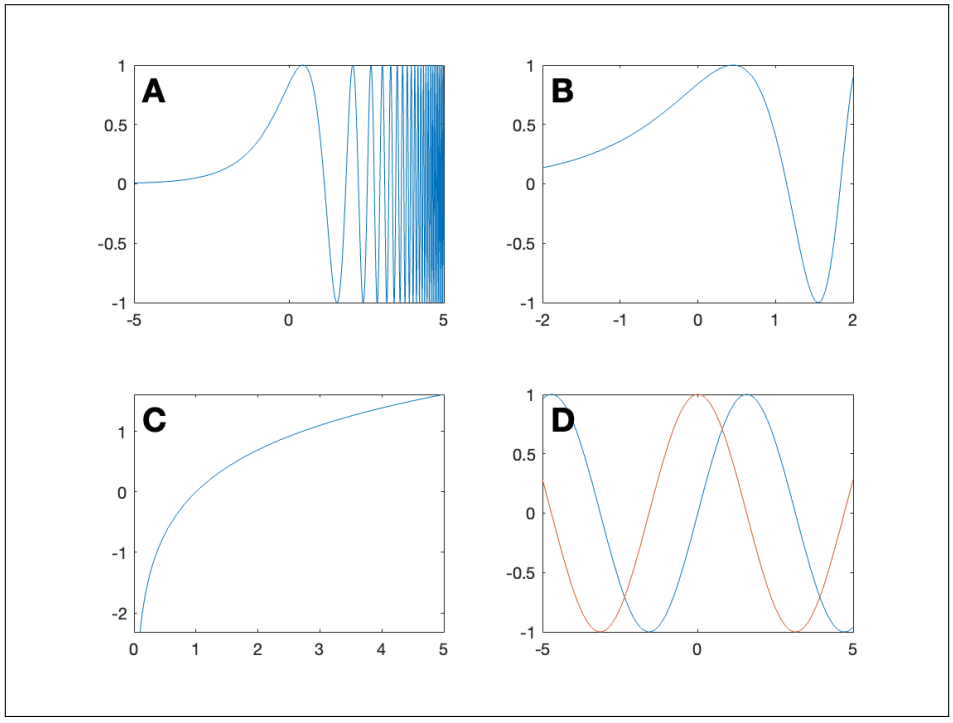

Figura 4.6: Diferentes usos del comando `fplot`.

`fplot` es muy potente, de manera que "entiende" incluso curvas paramétricas, Por ejemplo, podríamos representar una circunferencia usando $x = sin(t)$ e $y = cos(t)$ mediante el comando `fplot(@(x) sin(x),@(x) cos(x));` (Figura 4.7 Izquierda) o funciones más complejas como $x = 10 \times cos(t) \times sin(t)$ e $y = cos(t)$, dependientes ambas de la variable t, usando `f1 = @(x) 10*cos(x)*sin(x); f2 = @(x) cos(x); fplot(f1,f2)` (Figura 4.7 Derecha).

Observe que, en el caso del círculo, `fplot` es capaz de identificar los valores que faltan para cerrar la curva, incluso cuando estos no están definidos $[-5, 5]$. Como ejercicio, intente representar $x = sin(t)$ e $y = cos(t)$ usando `plot` y, primero la variable `t` entre $[-5, 5]$ y luego entre $[-2\pi, 2\pi]$.

Como curiosidad, MATLAB entiende que la definición de la función es independiente de la variable que usemos para definirla. Por lo tanto escribir `fplot(@(x) sin(x),@(x) cos(x));` es equivalente a escribir `fplot(@(t) sin(t),@(t) cos(t));`.

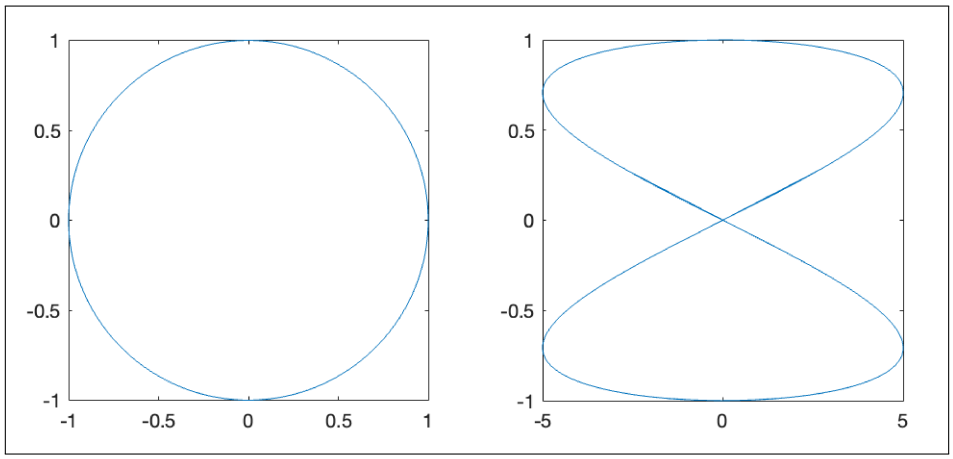

Figura 4.7: `fplot` usado en funciones paramétricas.

MATLAB tiene también la capacidad de representar funciones definidas de manera implícita, mediante el comando `fimplicit`. Como ejemplo podemos representar una circunferencia de radio 1, definida por la condición, $x^2 + y^2 = 1$, la cual se representa fácilmente como el lugar de los ceros de la función, $f(x, y) = x^2 + y^2 - 1$, pasándole esa expresión de la forma `fimplicit(@(x,y) x.^2 +y.^2 - 1)`.

4.3. Gráficas en 3D con MATLAB

Muchos de los problemas que encontramos al estudiar Física son tridimensionales, y debemos ser capaces de representar las soluciones y análisis apropiados para resolverlos mediante superficies, mallas, contornos, volúmenes, etc... Por este motivo, lo primero que deberemos entender es el concepto de "malla": Una malla es una estructura digital que representa una superficie o un objeto en el espacio tridimensional. Se compone de una serie de vértices y aristas que se conectan para formar las caras o teselas de la malla.

Para formar una malla en MATLAB podemos usar el comando `meshgrid`, que permite combinar 2 vectores para crear una malla cartesiana equiespaciada donde, para cada dirección

usada tendremos siempre la misma distancia entre 2 puntos. Tomemos por ejemplo los 2 vectores creados por `x0=linspace(-5,5,10);` e `y0=linspace(-5,5,10);`. Es obvio que si hacemos `plot(x0,y0)` tendremos la representación de una recta. Si queremos generar una malla usando estos vectores, deberemos escribir `[x,y]=meshgrid(x0,y0)`, que convierte nuestros 2 vectores de 1 fila y 10 columnas, en 2 matrices cuadradas de (10×10), creando una malla donde podemos evaluar una función dependiente de ambos parametros $z = f(x, y)$, como se representa en la Figura 4.8-A (`plot(x,y,'o')`. La función `ndgrid` es equivalente a `meshgrid` pero permite crear mallas de mayor dimensionalidad (3D, 4D, 5D, etc.).

Imaginemos ahora que queremos representar la función $z = 2x(sin(x))^2$, a lo largo de las coordenadas x e y. Para ello bastará con definirla `z = 2.*x.*(sin(x)).^ 2;` (recuerde el uso de "." en las operaciones elemento-a-elemento entre vectores) y dibujarla mediante la función `mesh(x,y,z)`.

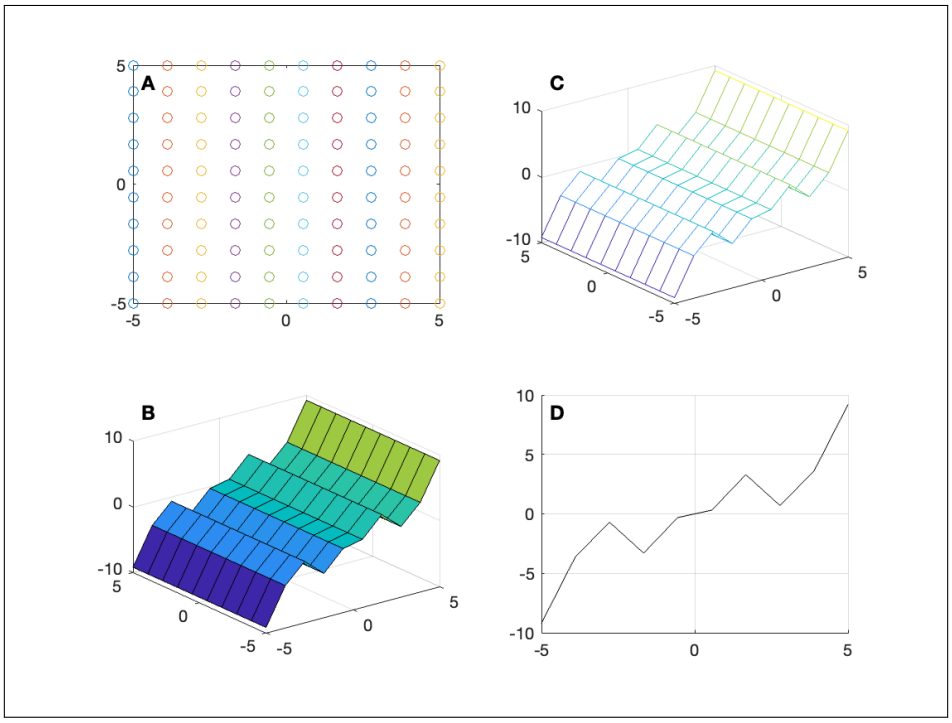

Figura 4.8: Representación tridimensional de la función $z = 2x(sin(x))^2$. **Panel A**, malla de valores. **Panel C**, malla de soluciones. **Panel B**, Superficie de la solución. **Panel D**, Superficie de la solución desde la vista perpendicular al eje y.

Como se puede ver en la Figura 4.8-C, el resultado de **mesh** es la malla definida por el conjunto de las 3 matrices cuadradas x, y, z de (10×10). Si quisiéramos representar la superficie definida por la variable z deberíamos hacer uso de la función **surf**, 4.8-B. Como puede observar, MATLAB representa las curvas en 3D usando una perspectiva isometrica, que permite ver las tres direcciones del espacio de una sola vez. Sin embargo, una de las funcionalidades gráficas es la posibilidad de mover el punto de vista del observador, bien mediante el uso del ratón (pinchando sobre el gráfico y arrastrando en la dirección deseada) o bien usando el comando `view`. En la Figura 4.8-D podemos observar la misma superficie representada en la Figura 4.8-C, tras ejecutar `view(0,90)`, donde 0 es el ángulo de acimut y 90 el de elevación de la línea de visión de la cámara para los ejes actuales. Como curiosidad, observe que MATLAB

colorea las superficies usando el valor de la variable z.

Entre las representaciones tridimensionales disponibles se encuentran, además, las extensiones de las que hemos visto en 2D, como `plot3(x,y,z)`, `stem3(x,y,z)` o `scatter3(x,y,z)`, que dejamos que el estudiante que pruebe por sí mismo.

Otras funciones integradas en MATLAB que son muy útiles para visualizar datos en 3D son `contour3`, `contour` o `contourf`, que representan tantas líneas de nivel (en "formato" línea o superficie) de $f(x, y)$ como queramos. Usemos el ejemplo típico MATLAB (que además coincide con su logo) para ilustrar estas visualizaciones: la función `peaks`, que representa unas montañas y valles (`mesh(peaks)`). Como se puede observar en la figura 4.9, el uso de estos comandos produce diferentes representaciones de las curvas de nivel usando diferentes perspectivas, que permiten representar la elevación del terreno.

Una aplicación de estos comandos extremadamente interesante, y que seguro va a estudiarse durante el grado en Física, es la distribución de temperatura en una placa plana, que viene dada por una ecuación en derivadas parciales, la cual puede resolverse numéricamente y ser visualizada usando MATLAB con las herramientas que acabamos de mostrar.

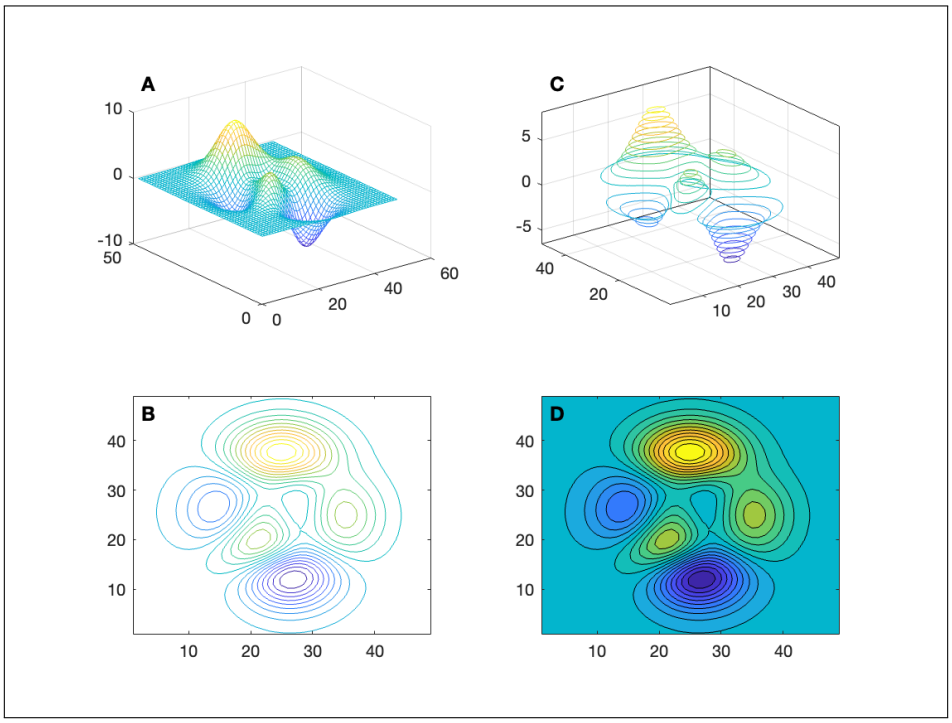

Figura 4.9: Representaciones tridimensionales de la función `peaks` usando líneas de nivel y superficies.**Panel A**: función `mesh`, **Panel B**: función `contour3`, **Panel C**: función `contour`, **Panel D**: función `contourf`.

4.3.1. Gráficas de Funciones en 3D con MATLAB

Todas las curvas en 3D pueden representarse usando funciones, exactamente igual a lo que hemos visto para 2D. Para ello es necesario solamente aplicar lo aprendido anteriormente a la siguientes funciones:

- `fplot3`, para representar puntos o líneas en 3D.
- `fmesh`, para representar mallas en 3D.
- `fsurf`, para representar superficies en 3D.
- `fcontour3`, para representar líneas de nivel 3D.

Observe, que todas esta funciones admiten las entradas `(f)`,`(f, [a,b])`, `(fx,fy,fx)` o del tipo `(fx,fy,fx,intervalo)`, en función de si queremos representar una función o tres funciones definidas mediante ecuaciones paramétricas en los intervalos deseados o el usado por defecto, que recordamos es $[-5, 5]$.

4.4. Representación de campos vectoriales con MATLAB

MATLAB ofrece herramientas para visualizar campos vectoriales, que son campos que tienen asignado un vector de alguna magnitud a cada punto del espacio. Dos de las funciones más utilizadas para este fin son `quiver` y `streamlines`.

Como ejemplo, vamos a representar el campo de velocidad en un fluido generado por un vórtice. Un vórtice (”vortex” en inglés) describe el flujo de un fluido (un gas también es un fluido) en rotación alrededor de un eje central. Ejemplos de vórtices en diversas escalas y contextos son los huracanes (creados por diferencias de presión y temperatura en la atmósfera), o los formados tras el desprendimientos de la capa límite del aire en las alas de un avión, que pueden alterar la sustentación de otros aviones en su estela.

El siguiente código crea una malla de puntos y define un campo vectorial que representa un vórtice. Usando la función `quiver` se dibujan los vectores que salen de cada punto de la malla, indicando la dirección y magnitud de la velocidad del flujo en ese punto:

```
% Crear una malla de puntos
clear; close all;
x0=-2:0.2:2; % vector de puntos en direccion x
y0=-2:0.2:2; % vector de puntos en direccion y
[x, y] = meshgrid(x0,y0); % creo la malla

% Se define el campo vectorial de velocidades inducido por un vortice
u = -y; % velocidad del flujo en direccion x
v = x; % velocidad del flujo en direccion y

% Visualizar el campo vectorial con quiver
figure(1); % creo figura 1
quiver(x, y, u, v, 1.5, 'LineWidth', 1.5);
```

En este caso, `quiver` ha recibido como argumentos, los puntos de la malla donde tenemos información de la velocidad `(x,y)` y los valores de la velocidad en cada dirección `(u,v)`. Además le hemos pedido que magnifique los vectores, multiplicando por 1.5 el valor de su módulo, y hemos asignado a los vectores un grosor de línea de 1.5, para mejorar la visualización.

Otra opción para representar campos vectoriales es representar sus líneas de corriente que son las aquellas paralelas a la velocidad (o alguna otra magnitud de interés) del campo en cada punto. Un código equivalente al anterior para representar las líneas de corriente usando MATLAB es:

```
% Crear una malla de puntos
clear; close all;
x0=-2:0.2:2; % vector de puntos en direccion x
y0=-2:0.2:2; % vector de puntos en direccion y
[x, y] = meshgrid(x0,y0); % creo la malla

% Defino el campo vectorial de velocidades inducido por un vortice
u = -y; % velocidad del flujo en direccion x
v = x; % velocidad del flujo en direccion y

% Visualizar el campo vectorial con líneas de corriente
% Definimos los puntos donde empieza cada línea:
x1=-2:0.3:2;

% y dibujamos:
figure(1);
plot(x1,x1,'o-k'); % Represento los puntos de donde salen las líneas de
    corriente, en este caso, elijo los puntos situados en la diagonal a mi
    dominio, y =x.
hold on; % Congelo el gráfico
streamline(x,y,u, v, x1, x1,0.5); % Pinto las líneas de corriente
```

Se puede observar que `streamline` ha recibido como argumentos, los puntos de la malla donde tenemos información de la velocidad `(x,y)` y los valores de la velocidad en cada dirección, `(u,v)`. Además le hemos tenido que indicar el punto de inicio de cada línea de corriente, para lo que hemos usado `21` puntos de la díagonal de nuestro dominio (definidos por `x1, x1`. Finalmente, hemos corregido la longitud de cada línea de corriente usando el factor `0.5`, para mejorar la visualización.

Los resultados de ambos códigos pueden verse en la figura 4.10.

Al igual que en 2D, los campos vectoriales se pueden representar en 3D usando la función `quiver3` o añadiendo variables adicionales a `streamline`, por ejemplo haciendo: `streamline(X,Y,Z,U,V,W,startX,startY,startZ)`.

En casos experimentales, donde las velocidades no siguen una distribución analítica (no están dados por una expresión como la que hemos usado nosotros), se pueden calcular las líneas de corriente usando las funciones `stream2` y `stream3` en 2 y 3 dimensiones.

4.5. Guardado de gráficas

Para guardar gráficas en MATLAB, existen varias opciones que permiten almacenarlas en diferentes formatos de archivos de gráficos, ya sea como imágenes basadas en píxeles o como archivos de gráficos vectoriales. Por otro lado, para minimizar el espacio en blanco al guardar o copiar gráficas,se puede utilizar la barra de herramientas de los ejes que permiten reducir el

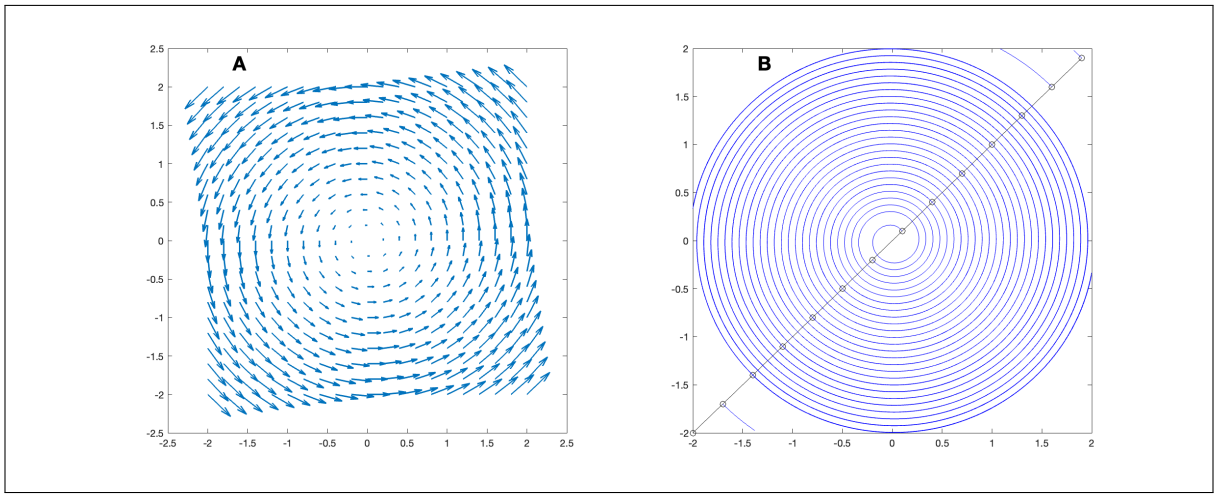

Figura 4.10: Representaciones del campo fluido generado por un vórtice usando vectores (**Panel A**) y líneas de corriente (**Panel B**).

espacio en blanco alrededor de la gráfica y ofrecen más flexibilidad en el proceso de guardado. Por ejemplo, se puede guardar una sola gráfica seleccionando el elemento adecuado en la lista desplegable debajo de **file**, tras pulsarlo con el ratón y posteriormente elegir el tipo de archivo en el que se quiere almacenar. Hay que tener en cuenta, que si deformarmos la pantalla, en función del tipo de archivo que elijamos, MATLAB guardará lo que estemos viendo en pantalla.

Entre las funciones disponibles para guardar gráficas en MATLAB podemos destacar:

- `print`, por ejemplo ≫ `print -djpeg -r200 'myimagen.jpeg'`, que guarda la ventana activa en el formato ".jpg" con una resolucón del 200 % y con el nombre "'myimagen.jpeg".

- `exportgraphics`, por ejemplo `exportgraphics(gcf,'nombredearchivo.pdf','ContentType','vec` que guarda la ventan activa (`gfc`: "get current figure") y su contenido como un archivo PDF que contiene gráficos vectoriales.

- `imwrite`, por ejemplo ≫ `imwrite(figure(1), 'archivo.tif', 'tif');` que guardará la figura `1` imagen en formato TIFF con el nombre "archivo.tif". Es importante recordar que al guardar una imagen con imwrite, el comportamiento predeterminado es reducir automáticamente la profundidad en bits a uint8. Sin embargo, para formatos como TIFF, puedes especificar el tipo de compresión que MATLAB utiliza para almacenar la imagen.

- `savefig`, por ejemplo ≫ `savefig('PeaksFile.fig')`, que guarda la pantalla activa en el formato de ventanas de MATLAB, con extensión ".fig".

4.6. Importante: Cómo generar gráficas de calidad

Para representar adecuadamente una gráfica en MATLAB, es fundamental seguir algunos consejos que ayudarán a mejorar la presentación y comprensión de los datos visualizados. Algunas recomendaciones incluyen enfocarse especialmente en la:

1. **Claridad:** El objetivo principal de una gráfica es comunicar información de forma clara y concisa.
2. **Precisión:** Los datos y las características de la gráfica deben ser precisos y representar fielmente la información.
3. **Eficiencia**: La gráfica debe ser fácil de entender y no sobrecargar al usuario con información innecesaria.

Los elementos clave que tiene que tener una buena gráfica son:

- Unos **ejes apropiados**. Los ejes deben estar nombrados usando etiquetas, que deben ser claras, concisas e incluir unidades de medida. Además tienen que estar en escalas apropiadas para el rango de datos y facilitar la comparación. Finalmente, tienen que tener unas líneas de división que permitan visualizar los valores intermedios en los ejes. Para etiquetar los ejes se pueden usar las funciones `xlabel` e `ylabel`.El comando `axis` controla los límites y el aspecto de la caja que ocupa la gráfica. Por defecto, los límites de los ejes vienen determinados por los valores de los datos representados.
- **Leyenda**. Si se representan múltiples conjuntos de datos en la misma gráfica se debe incluir una leyenda para distinguir entre ellos fácilmente. En este caso, MATLAB dispone del comando `legend`
- Hay que usar **tipos de línea diferentes** o colores para cada serie de datos si se representan más de una en la misma gráfica, lo que facilita la distinción entre ellas.
- Se debe **ajustar el zoom** de la gráfica para resaltar áreas específicas de interés, utilizando funciones como `xlim` y `ylim` o `zoom` para controlar los límites de los ejes. Además para ajustar el zoom adecuado en una gráfica se puedes utilizar la herramienta ”lupa” de la barra de herramientas de la ventana.
- Uso de **mallas**. Ayudan a visualizar el espacio representado. Deben ser lo suficientemente densas para mostrar los intervalos representados, pero no tan densas como para dificultar la visualización. Se pueden controlar con la función `grid`.
- Siempre es una buena idea incluir un **título** descriptivo que resuma el contenido de la gráfica y etiquete claramente cualquier anotación o punto relevante. No obstante, si la figura va a ser incluida en un documento esa información puede ir mejor en el pie de figura.

La Figura 4.11 muestra tres ejemplos de como **NO** se deben presentar gráficas.

Si se observa, la Figura 4.11-A representa una curva en un color que no se ve, y tampoco se aprecian las variaciones de las curvas azul y roja debido a una mala elección del zoom. La Figura 4.11-B muestra varias gráficas que se entrecruzan con el mismo color y tipo de línea sin que se pueda saber donde empieza una y termina otra. Finalmente, la Figura 4.11-C presenta dos gráficas representadas a la vez con escalas diferentes qué no permiten entender que representan (para ilustrar lo mal que está representada la información sepa que los puntos azules son exactamente los mismos que los representados en la figura 4.4-C.Además, ninguna de las gráficas tiene nombre en los ejes, leyendas o títulos, ni usa un tamaño de fuente

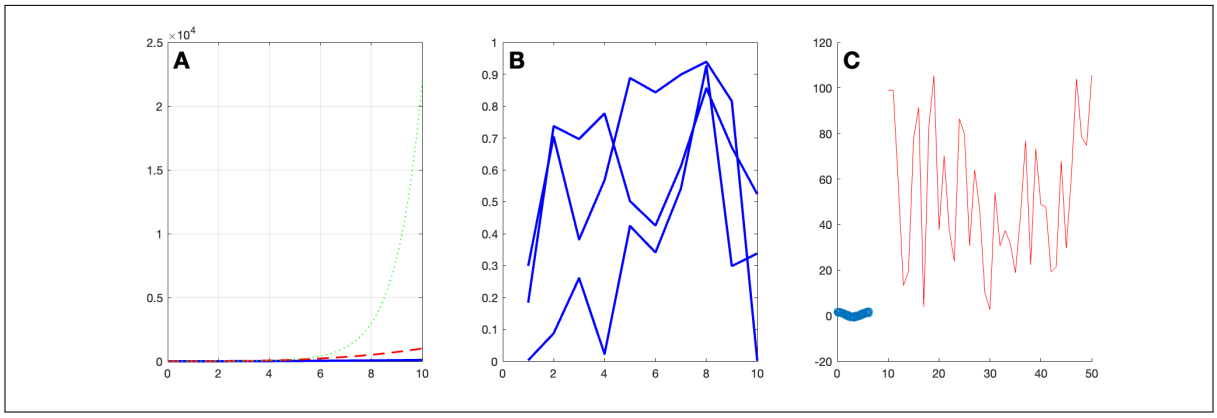

Figura 4.11: Tres gráficas muy mal hechas.

adecuado, lo que impide comprender la información que representan.

A continuación presentamos un código de ejemplo de cómo crear una buena gráfica, que se muestra en la figura 4.12:

```
% Crear datos de ejemplo.
% Imaginemos que quermos representar 2 movimientos oscilatorios
t = 0:0.1:10; % el tiempo en segundos
y1 = sin(t); % Un movimiento oscilatorio, por ejemplo un seno
y2 = cos(t); % Un movimiento oscilatorio, por ejemplo un coseno

% Crear el gráfico
figure(1);
plot(t, y1, 'b--', 'LineWidth', 2); % Graficar la función seno con línea
    discontinua azul
hold on; % Mantener el gráfico actual para agregar más elementos
plot(t, y2, 'r', 'LineWidth', 1.5); % Graficar la función coseno con línea
    continua roja

% Personalizar el gráfico
xlabel('tiempo␣(s)', 'FontSize', 14); % Etiqueta del eje X con letra de 14
    puntos
ylabel('Posicion␣(m)'); % Etiqueta del eje Y letra de 14 puntos
title('Movimiento␣Oscilatorio'); % Título del gráfico
legend('Posicion_1␣=␣sin(x)', 'Posicion_2␣=␣cos(t)'); % Leyenda de las líneas
grid on; % Mostrar la malla en el gráfico
set(gca, 'FontSize', 12); % Tamaño de fuente de los ejes

% Ajustar visualización de la leyenda y ejes para mejorar la legibilidad
legend('Location', 'southwest'); % Posicionar la leyenda en la esquina inferior
    izquierda
set(gca, 'LineWidth', 1.5); % Grosor de las líneas de los ejes
```

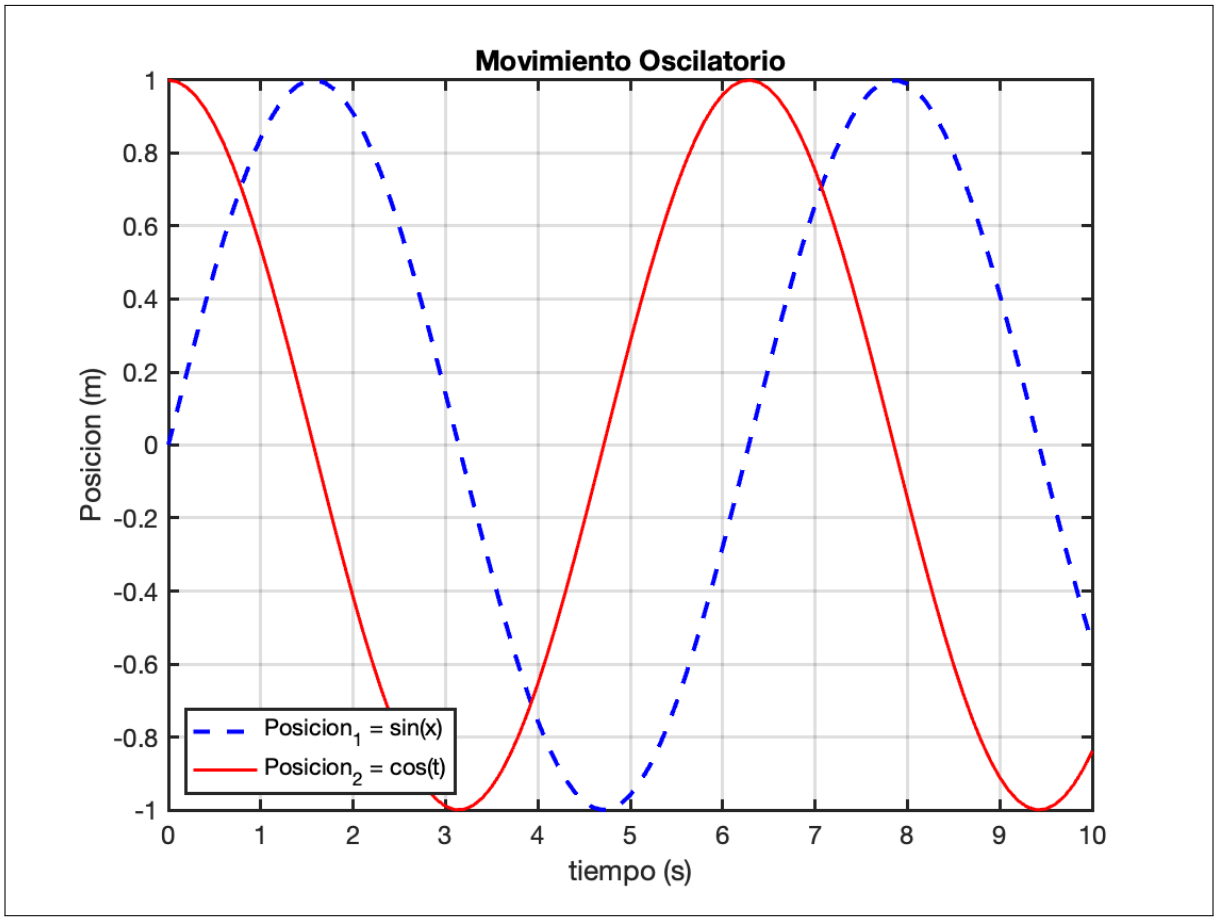

Figura 4.12: Ejemplo de como representar datos de manera apropiada.

4.7. Ejemplo de un problema resuelto

Ejercicio 1

Emplee lo aprendido en este tema para visualizar en una gráfica una función y sus correspondientes desarrollos en serie de Taylor en torno a un punto dado a orden cada vez más alto (por ejemplo puede considerar la función $\sin x + \cos x$ y sus desarrollos sucesivos para n entre 0 y 10 alrededor de $x = 0$).

Solución

Como ya asumimos que el estudiante tiene un conocimiento básico de MATLAB vamos a plantear la solución en forma de script que llevará anidadas las funciones de Taylor.

Para resolver este problema, lo más interesante es ver como se comportan estas series en un cierto entorno alrededor de $x = 0$ (pongamos el intervalo $[-2\pi, 2\pi]$). Por lo tanto nuestro script comienza, tras limpiar el entorno, asignando ese intervalo a una variable (`x`):

```
% Limpiar el entorno y la ventana de comandos
clear all; clc;
% Cerrar todas las Graficas
close all;
% Definir el dominio.
x = linspace(-2*pi, 2*pi, 100);
```

Para poder ver la precisión de los desarrollos de Taylor vamos a escribir tambien la función exacta que queremos desarrollar y a representarla, para luego ensamblar las diferentes aproximaciones en el mismo gráfico:

```
% Definir la función numérica exacta, para luego comparar los resultados
Fexacta= sin(x)+cos(x);

% Preparamos la figura
figure(1);
plot(x, Fexacta, 'k', 'Linewidth',2)
hold on; % Congelo la figura para incluir después las aproximaciones
grid on; % Incluyo una malla
```

Ahora, crearemos las funciones que calculen los desarrollos de Taylor para el seno y el coseno. En este caso, sabemos que la expansión del coseno viene dada por:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{2n!} \tag{4.1}$$

Por lo que podemos escribir el código equivalente para cualquier orden n:

```
% Desarrollo en series de Taylor del coseno a orden n
function y = cosTaylor(x,n)
y = 0;
for k = 0:n
       y = y + ((-1).^k) * x.^(2*k) / factorial(2*k);
end
end
```

De manera simila, para el caso del seno sabemos que:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... = \sum_{n=0}^{\infty} (-1)^{n+1} \frac{x^{2n+1}}{(2n+1)!} \tag{4.2}$$

Así, de la misma forma que con el coseno podemos definir una función para su expansión a cualquier orden n:

```
% Desarrollo en series de Taylor del seno a orden n
function y = sinTaylor(x,n)
y = 0;
for k = 0:n
       y = y + ((-1).^k) * x.^(2*k+1) / factorial(2*k+1);
end
end
```

Una vez definidas estas funciones podemos llamarlas para calcular nuestros desarrollos de Taylor. Por simplicidad vamos a usar un bucle en el que representar solamente las funciones orden 1,3,5 y 7:

```
% Bucle para representar los desarrollos de orden 1,3,5 y 7
for k=1:2:7
    % Calculo la función aproximada de orden k
    TaylorAprox= cosTaylor(x,k)+ sinTaylor(x,k);

    % Pinto la función aproximada con los atributos 'n'
    plot(x,TaylorAprox);
end
```

El problema de usar la sintaxis anterior es que no podemos distinguir fácilmente el orden que representa cada línea. Para corregir esta cuestión, vamos a asignar un atributo a cada una de ellas a lo largo del bucle. Como el índice `k` salta de 1 a 7 en intervalos de 2, vamos a utilizar un contador que nos permita dibujar una a una y de manera específica nuestras cuatro funciones. Así, el bucle anterior lo podemos modificar escribiendo:

```
% Atributos que vamos a representar
att={'-dc','-sg','-.r','-ob'}; % elijo un atributo diferente para cada línea
attL=[1]; % Tamaño de cada línea. En este caso todas iguales
attS=[4]; % Tamaño de cada marcador. En este caso todos iguales

% Creo un contador para ir asignando los atributos y uso n0 para no confundir
   con el orden de la aproximación.
n0=0;

% Bucle para representar los desarrollos de orden 1,3,5 y 7
for k=1:2:7

    % Sumo 1 al contador para representar la siguiente aproximación.
    n0=n0+1;

    % Calculo la función aproximada de orden k
    TaylorAprox= cosTaylor(x,k)+ sinTaylor(x,k);

    % Pinto la función aproximada con los atributos 'n0'
    plot(x,TaylorAprox, att{n0}, 'LineWidth', attL, 'MarkerSize', attS);

end
```

Los desarrollos en serie de Taylor son precisos en un cierto entorno alrededor del punto respecto al que desarrollamos ($x = 0$ en este caso), pero una vez nos salimos de ese entorno crecen de manera muy rápida, alejándose de la función que queremos aproximar, lo que puede dificultar una apropiada visualización. Para corregirlo, tenemos que decirle a MATLAB que queremos ver los resultados limitando el rango de ordenadas de la gráfica al conjunto de valores de la solución exacta. Definimos entonces el rango de valores de x e y en el que queremos que MATLAB nos muestre los resultados:

```
xlim([-7 7]); % aproximadamente alrededor de -2*pi y 2*pi
ylim([-2.5 2.5]); % aproximadamente los límites de la solución exacta
```

El siguiente problema que podemos encontrar al dibujar esta gráfica es que las etiquetas que indican qué función corresponde a cada línea ocupa demasiado espacio. Para evitar esto necesitamos indicar a MATLAB que no queremos emplear la opción por defecto para identificar cada curva de la gráfica, sino que queremos especificar qué nombres mostrar para cada una de estas curvas. Para ello hemos usado la función `legend`, cuya sintaxis puede consultarse en la ayuda de MATLAB. En nuestro caso necesitamos especificar una lista de "nombres" que nos permita identificar cada una de las curvas representadas:

```
% Añado Leyenda y ejes
legend('Solución␣Exacta', '␣Aproximación␣de␣orden␣1',...
'␣Aproximación␣de␣orden␣3','␣Aproximación␣de␣orden␣5',...
'␣Aproximación␣de␣orden␣7', 'FontSize', 11, 'Location', 'northeast');
```

Finalmente guardaremos nuestra gráfica en un archivo con formato `eps`. Por ejemplo, se puede hacer lo siguiente:

```
% Guardo la figura en EPS
print -depsc -r200 Taylor.eps
```

Hemos escogido el formato `.eps` (*encapsulated postscript*) porque es el más habitual cuando se trabaja en LaTeX. La principal ventaja del `.eps` es que es un formato vectorial, de modo que la calidad es muy alta y ocupa muy poco, aparte de ser estándar y gratuito. Con todo lo anterior, ya podemos escribir el script entero:

```
% Programa para calcular los desarrollos de Taylor del seno y coseno
% alrededor de x = 0.

% Limpiar el entorno y la ventana de comandos
clear all; clc
% Cerrar todas las ventanas
close all;
% Definir el dominio.
x = linspace(-2*pi, 2*pi, 100);

% Definir la función exacta, para luego comparar los resultados
Fexacta= sin(x)+cos(x);

% Vamos a representar la solución exacta,
% junto a las aproximaciones de orden 1, 3, 5 y 7

% Preparamos la figura
figure(1);
```

```
plot(x, Fexacta, '-k', 'Linewidth',2)

hold on;
grid on;

% Atributos que vamos a representar
att={'-dc','-sg','-.r','-ob'}; % elijo un atributo diferente para cada línea
attL=[1]; % Tamaño de cada línea. En este caso todas iguales
attS=[4]; % Tamaño de cada marcador. En este caso todos iguales

% Creo un contador para ir asignando los atributos
n0=0;

% Bucle para representar los desarrollos de orden 1,3,5 y 7
for k=1:2:7

    % Sumo 1 al contador para la representar la siguiente aproximación.
    n0=n0+1;

     % Calculo la función aproximada de orden k
    TaylorAprox= cosTaylor(x,k)+ sinTaylor(x,k);

    % Pinto la función aproximada con los atributos 'n'
    plot(x,TaylorAprox, att{n0}, 'LineWidth', attL, 'MarkerSize', attS);

end

% Corrijo los límites del gráfico
xlim([-7 7]);
ylim([-2.5 2.5]);

% Añado leyenda y ejes
legend('Solución␣Exacta', '␣Aproximación␣de␣orden␣1',...
'␣Aproximación␣de␣orden␣3','␣Aproximación␣de␣orden␣5',...
'␣Aproximación␣de␣orden␣7', 'FontSize', 11, 'Location', 'northeast');

title('Desarrollos␣de␣Taylor␣de␣sin(x)+␣cos(x)', 'FontSize', 14);


set(gca, 'FontSize', 14);
xlabel('X', 'FontSize', 14);
ylabel('Y', 'FontSize', 14);

% Guardo la figura en EPS
print -depsc -r200 Taylor.eps

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% DEFINICION DE FUNCIONES %%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Desarrollo en series de Taylor del coseno
function y = cosTaylor(x,n)
y = 0;
for k = 0:n
    y = y + ((-1).^k) * x.^(2*k) / factorial(2*k);
end
end

% Desarrollo en series de Taylor del seno
function y = sinTaylor(x,n)
y = 0;
for k = 0:n
    y = y + ((-1).^k) * x.^(2*k+1) / factorial(2*k+1);
end
end
```

El resultado se puede ver en la Figura 4.13. Como se puede ver, la soluciones a órdenes bajos se separan de la solución exacta en las inmediaciones de $x_0 = 0$, mientras que al aumentar el orden de la aproximación cada vez encontramos mejor precisión al alejarnos de x_0.

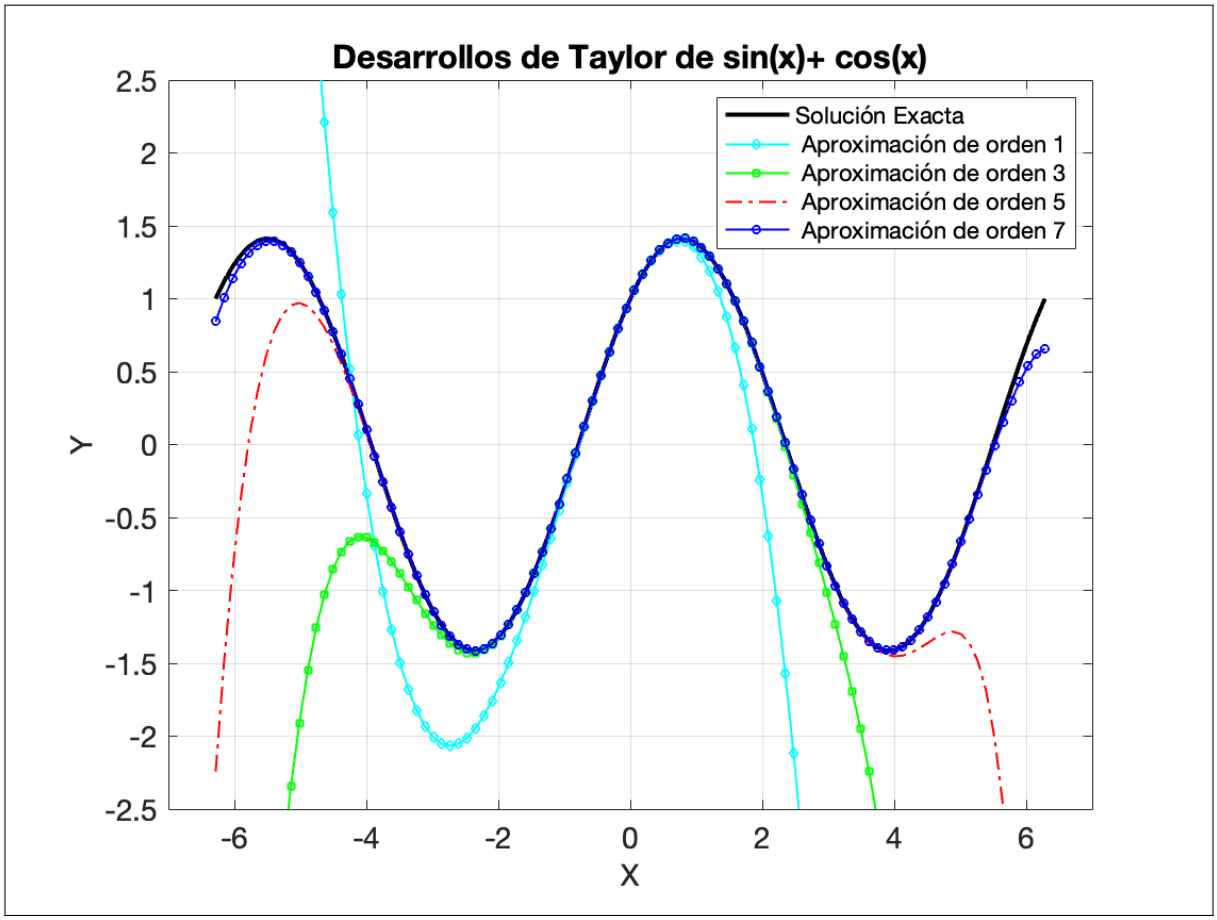

Figura 4.13: Resultado del problema propuesto.