Trabajo en Grupo Microeconomía

Participantes: Ariel Cadena, Lenin Loyola .

Correos Flacso: cpuruncajas@flacso.edu.ec , lloyolafl@flacso.edu.ec

Perfiles Github: Ariel98-lab, leninloyolavinces.

**1. Elaboración de Matriz 2 x2, encontrar vectores propios y mostrar explicación gráfica:**

Para el ejemplo se tomó una matriz de 2 x 2, como se muestra: (se refiere en este caso a solución en complejos).

Sistema de ecuaciones diferenciales:

dx0/dt = -1.0\*x0 - 3.0\*x1

dx1/dt = 3.0\*x0 - 1.0\*x1

La ecuación característica es:

lambda\*\*2 + 2.0\*lambda + 10.0

Los valores propios son:

Valores propios: [-1.+3.j -1.-3.j]

Los valores propios son complejos.

Gráfico para x:

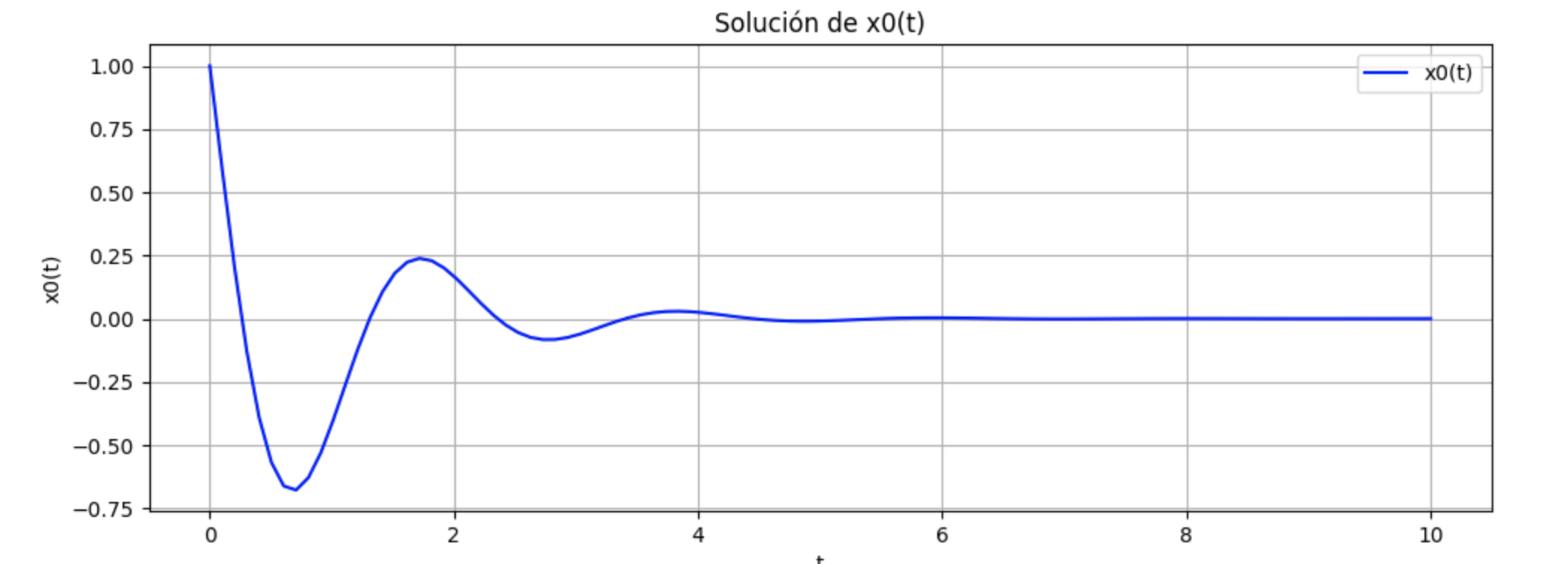


Gráfico para y:

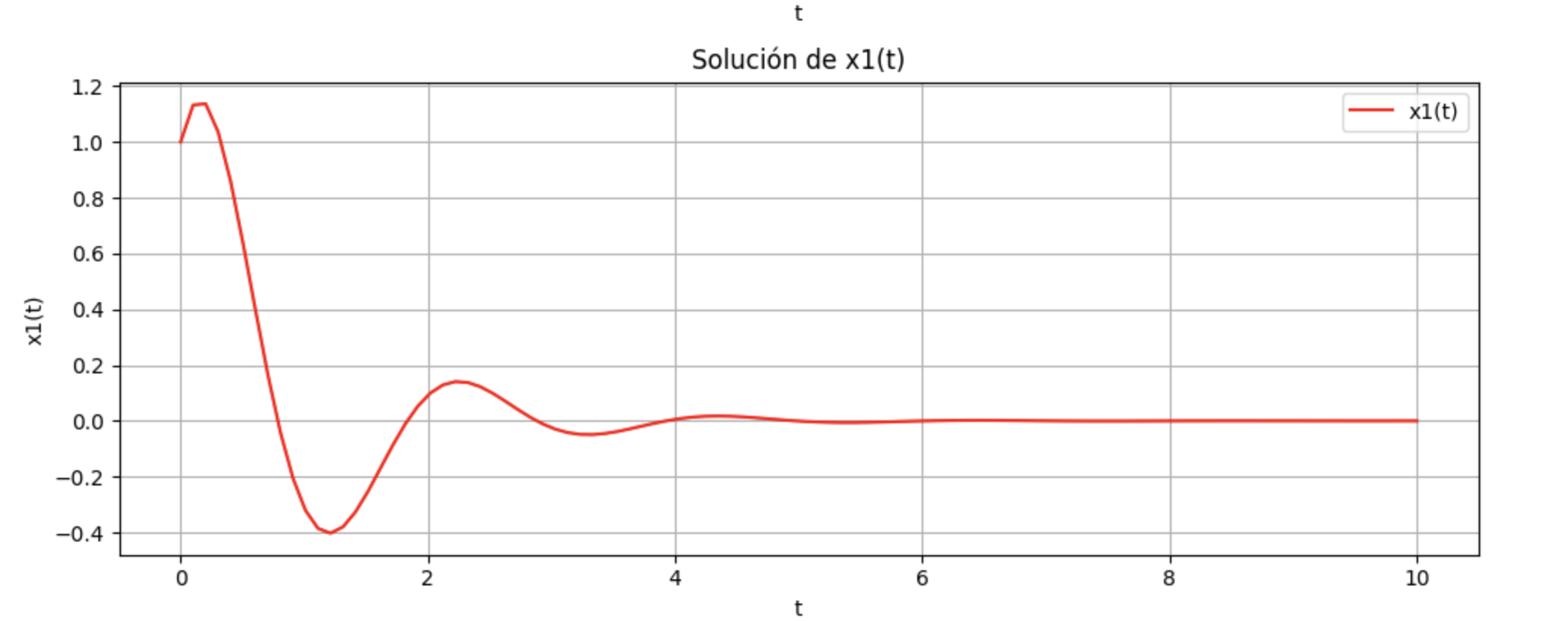
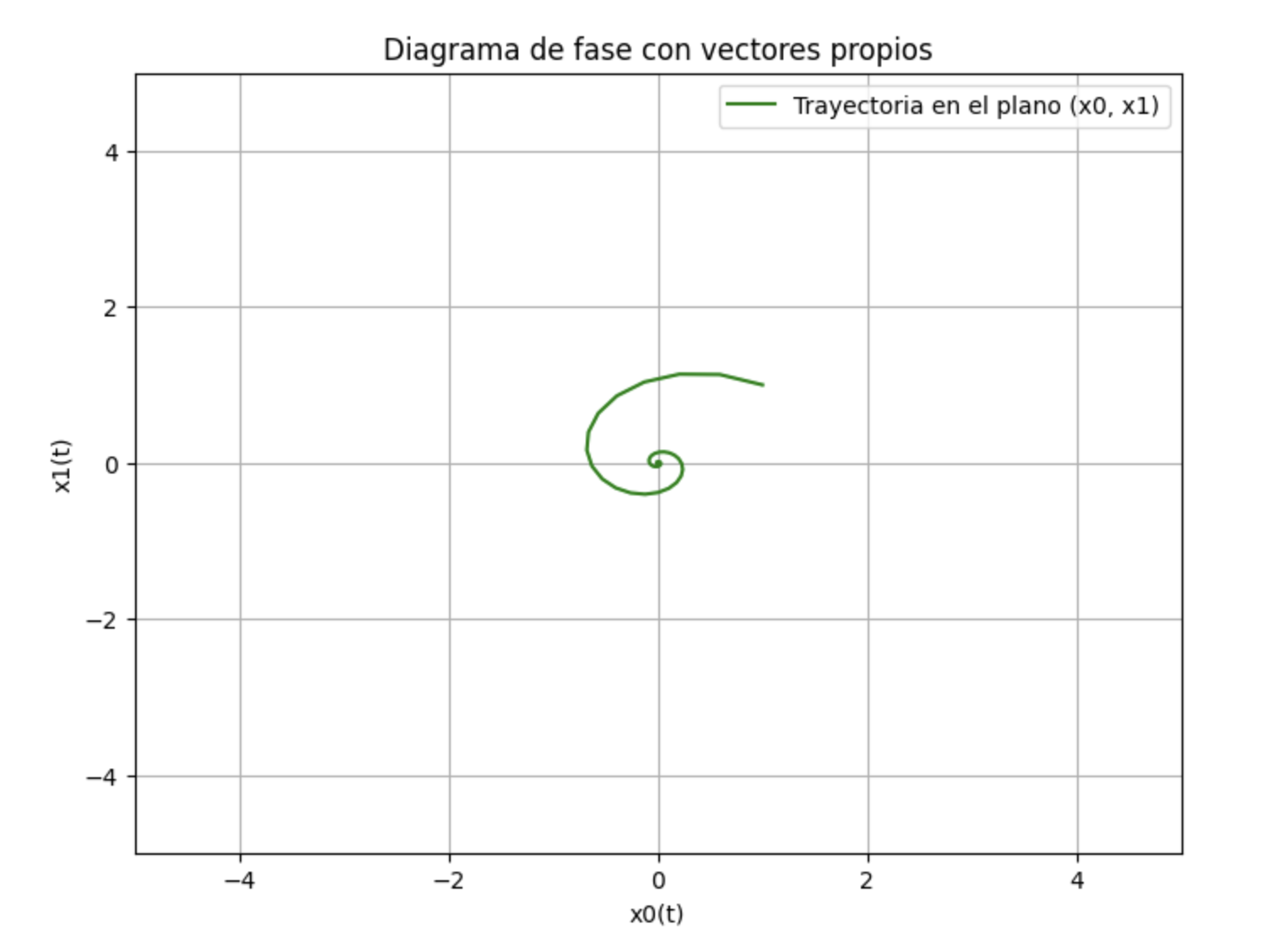


Diagrama para vectores propios:



**2. Explicación procedimental general del código de Phyton utilizado:**

2.1 Importación de bibliotecas:

* numpy: Para trabajar con matrices y álgebra lineal.
* sympy: Para cálculos simbólicos (como resolver ecuaciones diferenciales simbólicamente).
* matplotlib: Para crear gráficos.
* scipy.integrate: Para resolver ecuaciones diferenciales numéricamente con solve\_ivp.

2.2 Dibujo del robot: (no se incluye el detalle, no es parte del ejercicio).

2.3 Mostrar el gráfico:

* Se muestra el gráfico con el título "¡Hola! soy Panchito" usando plt.show().

**3. Explicación procedimental específica del código de Phyton utilizado:**

3.1 Función sistema\_ecuaciones\_diferenciales:

* Esta función interactúa con el usuario para resolver un sistema de ecuaciones diferenciales lineales.
* El usuario ingresa el tamaño de la matriz, que debe ser una matriz 2x2.
* Se pide al usuario que ingrese los valores de la matriz A, que es el sistema de coeficientes.

3.2 Ecuaciones diferenciales:

* Se crea una matriz simbólica A\_sym usando sympy para representar el sistema de ecuaciones diferenciales.
* El sistema de ecuaciones diferenciales es de la forma: dxidt=A⋅xi\frac{dx\_i}{dt} = A \cdot x\_idtdxi​​=A⋅xi​ Donde AAA es la matriz de coeficientes y xix\_ixi​ son las variables dependientes del tiempo.

3.3 Cálculo de valores propios y vectores propios:

* Se calcula la ecuación característica del sistema usando det(A - λI), donde λλλ es el valor propio y III es la matriz identidad.
* Luego, se calculan los valores propios y los vectores propios de la matriz A con numpy.linalg.eig().

3.4 Clasificación y solución:

* Si los valores propios son reales:

Se calcula la solución general usando exponenciales y los vectores propios.

* Si los valores propios tienen multiplicidad:

Se utiliza la forma de Jordan para encontrar la solución.

* Si los valores propios son complejos:

Se resuelven utilizando funciones trigonométricas (como seno y coseno) para representar la oscilación de las soluciones.

 **Función solucion\_general\_reales**:

* Calcula y muestra la solución general cuando los valores propios son reales y distintos.

 **Función solucion\_jordan**:

* Si hay multiplicidad en los valores propios, calcula la **forma de Jordan** y la matriz de cambio de base.

 **Función solucion\_compleja**:

* Si los valores propios son complejos, utiliza las funciones exponenciales y trigonométricas para calcular la solución.

**4. Resolución numérica del Sistema Diferencial:**

4.1 Sistema diferencial:

* Se define la función sistema\_diferencial, que representa la ecuación diferencial del sistema dxdt=A⋅x\frac{dx}{dt} = A \cdot xdtdx​=A⋅x.

4.2 Graficar la solución:

* Utiliza la función solve\_ivp de scipy para resolver el sistema de ecuaciones diferenciales numéricamente.
* Se grafica la solución de las variables dependientes (en este caso, x0(t)x\_0(t)x0​(t) y x1(t)x\_1(t)x1​(t)).

4.3 Graficar el diagrama de fase:

* También se crea un diagrama de fase, que muestra cómo las soluciones x0(t)x\_0(t)x0​(t) y x1(t)x\_1(t)x1​(t) evolucionan a lo largo del tiempo en el espacio de fases.
* Se agregan los vectores propios al gráfico para ilustrar la dirección y naturaleza de las soluciones.

**5. Explicación Teórica:**

Las ecuaciones diferenciales describen el cambio de las variables en el tiempo o a otra variable independiente. Un sistema de ecuaciones diferenciales lineales, es un conjunto donde cada ecuación depende linealmente de las variables y sus derivadas.

En este caso, el sistema tiene la forma:

dx1dt=a11x1+a12x2\frac{dx\_1}{dt} = a\_{11}x\_1 + a\_{12}x\_2dtdx1​​=a11​x1​+a12​x2​ dx2dt=a21x1+a22x2\frac{dx\_2}{dt} = a\_{21}x\_1 + a\_{22}x\_2dtdx2​​=a21​x1​+a22​x2​

Donde x1(t)x\_1(t)x1​(t) y x2(t)x\_2(t)x2​(t) son las variables dependientes del tiempo (en el código, el vector XXX las representa), y AAA es la matriz de coeficientes (los valores de aija\_{ij}aij​) que definen cómo las variables interactúan entre sí.

**¿Qué hace el código con estas ecuaciones?**

1. **Entrada de la matriz de coeficientes (A):**

El programa pide al usuario que ingrese una matriz AAA de tamaño n×nn \times nn×n. Esta matriz define el sistema de ecuaciones diferenciales. En el caso del código, se permite ingresar matrices de 2x2. Cada fila de la matriz contiene los coeficientes que definen cómo se relacionan las variables dependientes del sistema.

1. **Definición simbólica del sistema:**

Se utiliza sympy, una librería para cálculos simbólicos, para crear las ecuaciones diferenciales de manera simbólica. Esto significa que el sistema de ecuaciones es tratado de forma algebraica antes de resolverlo numéricamente.

1. **Cálculo de los valores propios y vectores propios:**

La matriz de coeficientes AAA tiene asociados unos valores propios y vectores propios. Los valores propios indican la "tasa de crecimiento" o "decadencia" de las soluciones del sistema, y los vectores propios nos dan las direcciones en las que las soluciones crecen o decrecen.

El código calcula estos valores propios y vectores propios para entender mejor cómo se comporta el sistema a largo plazo.

1. **Solución del sistema de ecuaciones:**

Dependiendo de los valores propios calculados, el código considera tres escenarios:

* + - Valores propios reales distintos: El sistema tiene soluciones de la forma exponencial, y el código genera la solución general utilizando exponentes y vectores propios.
    - Valores propios con multiplicidad: Si los valores propios se repiten, se usa la forma de Jordan para encontrar una solución más general, que implica técnicas de álgebra matricial más complejas.
    - Valores propios complejos: Si los valores propios son complejos, las soluciones tienen un comportamiento oscilatorio (consenos y senos). El código calcula la solución en términos de estas funciones trigonométricas.

1. **Representación gráfica de las soluciones:**

El código resuelve el sistema de ecuaciones tanto simbólicamente (algebraicamente) como numéricamente utilizando la función solve\_ivp de scipy, que resuelve ecuaciones diferenciales iniciales.

Se grafican las soluciones para mostrar cómo evolucionan las variables x1(t)x\_1(t)x1​(t) y x2(t)x\_2(t)x2​(t) a lo largo del tiempo.

1. **Diagrama de fase:**

Se crea un diagrama de fase, que es un gráfico que muestra las trayectorias de las soluciones del sistema en el espacio de las variables dependientes (por ejemplo, x1x\_1x1​ vs. x2x\_2x2​). Este diagrama es útil para analizar la estabilidad y el comportamiento del sistema.

**Conclusión Teórica:**

En términos teóricos, el código resuelve y visualiza un sistema de ecuaciones diferenciales lineales de la forma:

dxdt=A⋅x\frac{dx}{dt} = A \cdot xdtdx​=A⋅x

El código realiza los siguientes pasos:

1. **Entrada de la matriz de coeficientes AAA** y la configuración de las ecuaciones diferenciales.
2. **Cálculo de los valores y vectores propios** de la matriz AAA, que describe cómo se comportan las soluciones.
3. Dependiendo de la naturaleza de los valores propios, el código resuelve el sistema usando **soluciones exponenciales**, **forma de Jordan** o **funciones trigonométricas**.
4. **Resolución numérica** del sistema usando solve\_ivp y la visualización de las soluciones en **gráficos**.
5. Creación de un **diagrama de fase** para observar la evolución de las soluciones en el espacio de fases.

Este código es una herramienta útil para resolver y visualizar sistemas de ecuaciones diferenciales lineales y explorar cómo sus soluciones se comportan dependiendo de las propiedades de la matriz AAA. Además, la representación gráfica del robot y las interacciones con el usuario lo hacen más atractivo y comprensible.

**6. Código en Visual para resultados y explicación gráfica:**

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

import matplotlib.pyplot as plt

import matplotlib.patches as patches

# Crear la figura

fig, ax = plt.subplots()

# Dibujar el cuerpo del robot (rectángulo)

body = patches.Rectangle((1, 1), 4, 6, edgecolor="black", facecolor="lightblue", linewidth=2)

ax.add\_patch(body)

# Dibujar la cabeza (círculo)

head = patches.Circle((3, 8), 1.5, edgecolor="black", facecolor="lightgray", linewidth=2)

ax.add\_patch(head)

# Dibujar los ojos (círculos)

eye1 = patches.Circle((2.6, 8.8), 0.2, edgecolor="black", facecolor="white")

eye2 = patches.Circle((3.4, 8.8), 0.2, edgecolor="black", facecolor="white")

ax.add\_patch(eye1)

ax.add\_patch(eye2)

# Dibujar la sonrisa (arco curvado hacia arriba)

smile = patches.Arc((3, 8.1), 1.2, 0.8, angle=0, theta1=180, theta2=360, edgecolor="black", linewidth=2)

ax.add\_patch(smile)

# Dibujar los brazos (uno saludando levantado y otro abajo)

arm1 = patches.Rectangle((0, 5.5), 1, 0.5, edgecolor="black", facecolor="gray", linewidth=2) # Brazo abajo

arm2 = patches.Rectangle((4.5, 6.5), 0.5, 1.5, edgecolor="black", facecolor="gray", linewidth=2) # Brazo levantado

ax.add\_patch(arm1)

ax.add\_patch(arm2)

# Dibujar las piernas (rectángulos)

leg1 = patches.Rectangle((1.2, 0), 0.8, 1, edgecolor="black", facecolor="gray", linewidth=2)

leg2 = patches.Rectangle((3.0, 0), 0.8, 1, edgecolor="black", facecolor="gray", linewidth=2)

ax.add\_patch(leg1)

ax.add\_patch(leg2)

# Ajustar los límites y la proporción

ax.set\_xlim(-1, 7)

ax.set\_ylim(-1, 10)

ax.set\_aspect("equal")

ax.axis("off")

# Mostrar la gráfica

plt.title("¡Hola! soy Panchito")

plt.show()

def sistema\_ecuaciones\_diferenciales():

# Saludo inicial

print("Soy un software creado para ayudarte a resolver ecuaciones diferenciales.")

print("Sigue las instrucciones para que pueda ayudarte.")

# Ingreso del tamaño de la matriz

print("Ingresa el tamaño de la matriz (Ejemplo: 2x2).")

print("UNICAMENTE SE ACEPTAN MATRICES 2x2.")

tamaño = input()

# Extraemos el número de filas y columnas

n, m = map(int, tamaño.split('x'))

if n != m:

print("La matriz debe ser cuadrada (n x n). Por favor, ingresa un tamaño válido.")

return

# Crear la matriz A a partir de la entrada del usuario

A = []

print(f"Ingrese la matriz {n}x{m} fila por fila, separando los números con espacios:")

print("Ejemplo: 1 espacio 2")

for i in range(n):

fila = list(map(float, input().split()))

A.append(fila)

A = np.array(A)

# Definir vector de variables simbólicas

X = sp.Matrix(sp.symbols(f'x0:{n}'))

A\_sym = sp.Matrix(A)

sistema = A\_sym \* X

print("\nSistema de ecuaciones diferenciales:")

for i in range(n):

print(f"dx{i}/dt = {sistema[i]}")

# Cálculo de la ecuación característica

lambda\_ = sp.symbols('lambda')

I = sp.eye(n) # Matriz identidad de tamaño n

caracteristica = sp.det(A\_sym - lambda\_ \* I) # Ecuación característica

print("\nEcuación característica:")

print(caracteristica)

# Cálculo de los valores propios y los vectores propios

valores\_propios, vectores\_propios = np.linalg.eig(A)

# Mostrar los valores propios

print("\nValores propios:", valores\_propios)

# Clasificación de los valores propios

if np.all(np.isreal(valores\_propios)):

multiplicidades = [list(valores\_propios).count(l) for l in valores\_propios]

if all(m == 1 for m in multiplicidades):

print("Los valores propios son reales y distintos. La solución general es:")

print(solucion\_general\_reales(A, valores\_propios, vectores\_propios))

graficar\_solucion(A)

graficar\_diagrama\_fase(A, valores\_propios, vectores\_propios)

else:

print("Los valores propios tienen multiplicidad. Forma de Jordan:")

print(solucion\_jordan(A, valores\_propios, vectores\_propios))

graficar\_solucion(A)

graficar\_diagrama\_fase(A, valores\_propios, vectores\_propios)

else:

print("Los valores propios son complejos. La solución general es:")

print(solucion\_compleja(A, valores\_propios, vectores\_propios))

graficar\_solucion(A)

graficar\_diagrama\_fase(A, valores\_propios, vectores\_propios)

def solucion\_general\_reales(A, valores\_propios, vectores\_propios):

t = sp.Symbol('t')

solucion = []

for i in range(len(valores\_propios)):

vi = vectores\_propios[:, i]

solucion.append(sp.exp(valores\_propios[i] \* t) \* vi)

# Mostrar la solución en la consola

for i, sol in enumerate(solucion):

print(f"Solución {i+1}: {sol}")

# Mostrar en la consola el enlace como texto

print("\nPara más información sobre solución general de los reales, consulta:")

print("https://es.wikipedia.org/wiki/Ecuaci%C3%B3n\_diferencial")

return solucion

def solucion\_jordan(A, valores\_propios, vectores\_propios):

A\_sym = sp.Matrix(A)

J, P = A\_sym.jordan\_form() # Forma de Jordan

return f"Forma de Jordan: \n{J}\n\nMatriz de cambio de base P: \n{P}"

# Mostrar en la consola el enlace como texto

print("\nPara más información sobre solución por el método de Jordan, consulta:")

print("https://www.mate.unlp.edu.ar/practicas/51\_11\_13052019191305.pdf")

def solucion\_compleja(A, valores\_propios, vectores\_propios):

t = sp.Symbol('t')

solucion = []

for i, valor in enumerate(valores\_propios):

if np.iscomplex(valor):

alpha = np.real(valor)

beta = np.imag(valor)

solucion.append(

sp.exp(alpha \* t) \* (

sp.cos(beta \* t) \* vectores\_propios[:, i] + sp.sin(beta \* t) \* vectores\_propios[:, i]

)

)

# Mostrar la solución en la consola

print("\nLa solución con valores propios complejos está pendiente de implementación.")

print("Puedes consultar más detalles en el siguiente artículo:")

print("https://fernandorevilla.es/2014/03/23/ecuacion-diferencial-compleja/")

return solucion

def sistema\_diferencial(t, X, A):

return A @ X

def graficar\_solucion(A):

t\_span = (0, 10)

t\_eval = np.linspace(\*t\_span, 100)

X0 = np.ones(A.shape[0]) # Condición inicial (vector de unos)

solucion = solve\_ivp(sistema\_diferencial, t\_span, X0, t\_eval=t\_eval, args=(A,))

# Crear una figura con dos subgráficas

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

ax1.plot(solucion.t, solucion.y[0], label=f'x0(t)', color='b')

ax1.set\_title("Solución de x0(t)")

ax1.set\_xlabel("t")

ax1.set\_ylabel("x0(t)")

ax1.grid(True)

ax1.legend()

if A.shape[0] > 1:

ax2.plot(solucion.t, solucion.y[1], label=f'x1(t)', color='r')

ax2.set\_title("Solución de x1(t)")

ax2.set\_xlabel("t")

ax2.set\_ylabel("x1(t)")

ax2.grid(True)

ax2.legend()

plt.tight\_layout()

plt.show()

def graficar\_diagrama\_fase(A, valores\_propios, vectores\_propios):

t\_span = (0, 10)

t\_eval = np.linspace(\*t\_span, 100)

X0 = np.ones(A.shape[0]) # Condición inicial (vector de unos)

solucion = solve\_ivp(sistema\_diferencial, t\_span, X0, t\_eval=t\_eval, args=(A,))

# Graficar el diagrama de fase

plt.figure(figsize=(8, 6))

plt.plot(solucion.y[0], solucion.y[1], label="Trayectoria en el plano (x0, x1)", color='g')

# Graficar los vectores propios

for i in range(len(valores\_propios)):

valor = valores\_propios[i]

vector = vectores\_propios[:, i]

# Si el valor propio es real

if np.isreal(valor):

# Si hay multiplicidad, graficamos solo el vector propio

if list(valores\_propios).count(valor) > 1:

plt.quiver(0, 0, vector[0], vector[1], angles='xy', scale\_units='xy', scale=1, color='r', alpha=0.6)

else:

plt.quiver(0, 0, vector[0], vector[1], angles='xy', scale\_units='xy', scale=1, color='b', alpha=0.6)

# No graficamos si el valor propio es complejo

elif np.iscomplex(valor):

continue

# Añadir detalles al gráfico

plt.title("Diagrama de fase con vectores propios")

plt.xlabel("x0(t)")

plt.ylabel("x1(t)")

plt.xlim(-5, 5)

plt.ylim(-5, 5)

plt.grid(True)

plt.legend()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

sistema\_ecuaciones\_diferenciales()