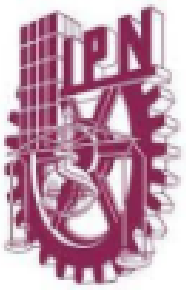


Instituto Politécnico Nacional.

Escuela Superior de Cómputo (ESCOM).

Ingeniería en Inteligencia Artificial.

Álgebra lineal.

Proyecto 4: Simulación de Órbitas Planetarias en 2D.

Profesor: David Correa Coyac.

Alumnos:
Cerón Samperio Lizeth Montserrat.
Higuera Pineda Angel Abraham.
Lorenzo Silva Abad Rey.
Moya Rivera Mia Paulina.

Grupo: 2BM2.

Fecha de entrega: 06 de enero de 2025.

Índice.

Introducción.

Conforme a la NASA, todos los objetos espaciales del Sistema Solar realizan trayectorias alrededor del Sol en un plano imaginario conocido como plano de la eclíptica. A pesar de la quietud desde el punto de vista terrestre, en realidad, estos objetos presentan órbitas constantes y recurrentes alrededor de otros objetos de mayor tamaño.

Se denomina satélites a estos objetos que orbitan alrededor de un cuerpo central, dado que su desplazamiento se basa directamente en la atracción gravitatoria que ejerce el objeto central. Nuestra luna es un ejemplo conocido de un satélite natural, manteniéndose en órbita alrededor de la Tierra.

La trayectoria elíptica de las orbitas que siguen los planetas y otros objetos alrededor del Sol está relacionada con la Ley de Gravitación Universal de Newton. Conforme a la ley de Newton, cualquier pareja de objetos de masa se atraen entre sí con una fuerza que es proporcional al resultado de sus masas e inversamente proporcional al cuadrado de la separación entre ellos. En este escenario, este atractivo gravitacional produce trayectorias orbitales constantes y recurrentes. La simulación puede facilitar la visualización de la manera en que la interacción gravitatoria entre el Sol y los planetas establece el movimiento orbital. Por ejemplo, los usuarios tienen la posibilidad de modificar parámetros como el radio orbital para observar cómo se modifica el período orbital conforme a las leyes físicas, evidenciando conceptos fundamentales como la correlación entre la velocidad, la distancia y la fuerza gravitatoria.

Además de calcular posiciones en órbitas, las transformaciones lineales poseen usos más sofisticados en la simulación de fenómenos astronómicos. Por ejemplo, el fenómeno de la precesión orbital, que se refiere al cambio paulatino y progresivo en la dirección de una órbita, puede ser modelado a través de rotaciones sucesivas realizadas a través del tiempo. Este fenómeno es esencial para comprender sucesos como las variaciones en la dirección del eje terrestre o las alteraciones en el orbital de objetos de menor tamaño. Asimismo, las transformaciones lineales son esenciales para simplificar el paso entre distintos sistemas de coordenadas.

Durante la historia, el entendimiento de las órbitas y el desplazamiento de los astros ha sido un elemento esencial en la astronomía. Las Leyes de Kepler, establecidas en el siglo XVII, transformaron nuestra percepción del cosmos al detallar matemáticamente las rutas elípticas de los planetas y establecer vínculos fundamentales, como la proporción entre el período orbital y la distancia media al Sol. Estas ideas establecieron los fundamentos para la Ley de Gravitación Universal de Newton y, más adelante, para los modelos de dinámica contemporáneos. Hoy

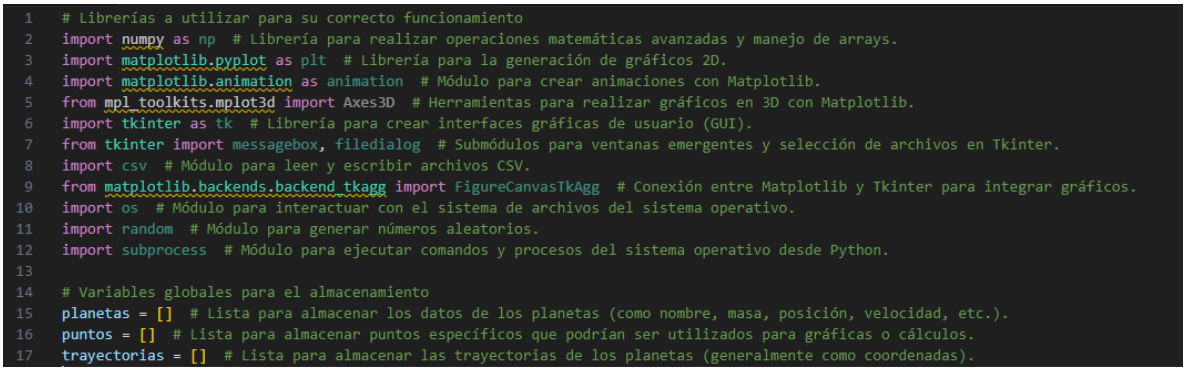
en día, entidades como la NASA utilizan simulaciones sofisticadas para organizar misiones espaciales, analizar sistemas planetarios y anticipar interacciones entre astros. Este tipo de instrumentos no solo poseen usos en la ciencia y la tecnología, sino que también motivan al público al estimular el interés por la exploración espacial y la ciencia.

La simulación de este programa respecto al movimiento orbital en un sistema solar tanto en un espacio bidimensional o tridimensional, en el que se incluyen parámetros como el radio orbital (Tamaño de la órbita) y la velocidad angular (Rapidez con la que los planetas completan sus órbitas), es para demostrar cómo las matrices de transformación lineal, específicamente matrices de rotación calculen la posición exacta de cada objeto en los diferentes puntos de su órbita, modelando su movimiento angular alrededor de la estrella; dado que las órbitas suelen ser elípticas, el uso de rotaciones es necesario para representar el movimiento angular alrededor del sol. Mientras que el uso de matrices de escalado ayuda para ajustar sus dimensiones, de una trayectoria circular a una elíptica en tiempo real, lo que es muy importante para modelar órbitas realistas, ya que en la mayoría de los casos no son perfectamente circulares.

Desarrollo.

Objetivo: Desarrollar un programa que simule el movimiento orbital de planetas en un sistema solar simplificado utilizando matrices de transformación lineal, específicamente matrices de rotación y escalado, para calcular y visualizar trayectorias elípticas en tiempo real. El proyecto permitirá a los usuarios ajustar parámetros como el radio orbital y la velocidad angular para personalizar la simulación.

- **Documentación del código.**



```python
# Librerías a utilizar para su correcto funcionamiento
import numpy as np  # Librería para realizar operaciones matemáticas avanzadas y manejo de arrays.
import matplotlib.pyplot as plt  # Librería para la generación de gráficos 2D.
import matplotlib.animation as animation  # Módulo para crear animaciones con Matplotlib.
from mpl_toolkits.mplot3d import Axes3D  # Herramientas para realizar gráficos en 3D con Matplotlib.
import tkinter as tk  # Librería para crear interfaces gráficas de usuario (GUI).
from tkinter import messagebox, filedialog  # Submódulos para ventanas emergentes y selección de archivos en Tkinter.
import csv  # Módulo para leer y escribir archivos CSV.
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg  # Conexión entre Matplotlib y Tkinter para integrar gráficos.
import os  # Módulo para interactuar con el sistema de archivos del sistema operativo.
import random  # Módulo para generar números aleatorios.
import subprocess  # Módulo para ejecutar comandos y procesos del sistema operativo desde Python.

# Variables globales para el almacenamiento
planetas = []  # Lista para almacenar los datos de los planetas (como nombre, masa, posición, velocidad, etc.).
puntos = []  # Lista para almacenar puntos específicos que podrían ser utilizados para gráficas o cálculos.
trayectorias = []  # Lista para almacenar las trayectorias de los planetas (generalmente como coordenadas).
```

- **Importación de librerías:** Se importan varias librerías que permiten realizar diferentes tareas como cálculos matemáticos, graficación 2D y 3D, manejo de interfaces gráficas, archivos, y procesos del sistema operativo.
- **Declaración de variables globales:**

1. planetas: Lista destinada a guardar información sobre los planetas que serán procesados en el programa.
2. puntos: Almacenará puntos que pueden ser relevantes para cálculos o visualización.
3. trayectorias: Guardará las trayectorias de los planetas, probablemente en forma de coordenadas (x, y, z) para visualización o simulaciones.

```python
# Función para cargar planetas desde un archivo CSV
def cargar_planetas_desde_csv():
    global planetas  # Se indica que la función utiliza la variable global "planetas".

    # Abrir un cuadro de diálogo para que el usuario seleccione el archivo CSV.
    file_path = filedialog.askopenfilename(filetypes=[("CSV files", "*.csv")])

    if file_path:  # Verifica si el usuario seleccionó un archivo.
        try:
            # Intenta abrir el archivo seleccionado en modo lectura.
            with open(file_path, newline='') as csvfile:
                reader = csv.DictReader(csvfile)  # Crea un lector para leer los datos como diccionarios.

                planetas.clear()  # Limpia la lista "planetas" para evitar duplicados si se cargan nuevos datos.

                # Itera sobre cada fila del archivo CSV.
                for row in reader:
                    try:
                        # Extrae y convierte los datos de cada fila.
                        nombre = row['Planeta']  # Nombre del planeta.
                        radio = float(row['Radio'])  # Radio del planeta.
                        velocidad_angular = float(row['Velocidad Angular'])  # Velocidad angular del planeta.
                        escala = float(row['Escala'])  # Escala de representación gráfica del planeta.

                        # Agrega los datos procesados a la lista "planetas" como un diccionario.
                        planetas.append({
                            "Planeta": nombre,
                            "Radio": radio,
                            "Velocidad Angular": velocidad_angular,
                            "Escala": escala
                        })
                    except ValueError:
                        # Si ocurre un error al convertir datos, muestra un mensaje de error.
                        messagebox.showerror(
                            "Error",
                            f"Datos inválidos en el archivo CSV: {row}"
                        )
                        continue  # Continúa con la siguiente fila sin interrumpir el proceso.

                # Muestra un mensaje indicando que los planetas se cargaron correctamente.
                messagebox.showinfo("Éxito", "Planetas cargados exitosamente desde el archivo CSV.")

                # Llama a la función regenerar_planetas para actualizar la visualización con los nuevos datos.
                regenerar_planetas()

        except Exception as e:
            # Si ocurre algún error al abrir o leer el archivo, muestra un mensaje con detalles del error.
            messagebox.showerror("Error", f"No se pudo leer el archivo CSV: {e}")
```

Explicación detallada del código

Variables:

1. **planetas (global):**

- Lista que almacena los datos de los planetas en forma de diccionarios.
- Cada planeta tiene las claves: Planeta (Nombre), Radio (Tamaño), Velocidad Angular (Movimiento relativo), y Escala (Para visualización).

2. file_path: Contiene la ruta absoluta del archivo CSV seleccionado por el usuario.
3. csvfile: Representa el archivo CSV abierto para lectura.
4. reader: Un objeto csv.DictReader que itera sobre las filas del archivo CSV y devuelve cada fila como un diccionario.

5. row: Representa cada fila del archivo CSV, donde las claves corresponden a los nombres de las columnas del archivo.
6. nombre, radio, velocidad_angular, escala: Variables que almacenan los valores extraídos de cada fila del archivo CSV.
7. e: Contiene la excepción capturada si ocurre un error durante la lectura del archivo.

Funciones:

1. filedialog.askopenfilename:

- Muestra un cuadro de diálogo para que el usuario seleccione un archivo.
- Devuelve la ruta del archivo seleccionado.

2. csv.DictReader: Lee un archivo CSV y convierte cada fila en un diccionario donde las claves son los nombres de las columnas.
3. messagebox.showerror: Muestra una ventana emergente con un mensaje de error.
4. messagebox.showinfo: Muestra una ventana emergente con un mensaje informativo.
5. regenerar_planetas: Función (Definida en otro lugar del programa) que actualiza la visualización gráfica o lógica de los planetas con los nuevos datos.

Módulos utilizados:

1. tkinter.filedialog: Para mostrar el cuadro de diálogo de selección de archivos.
2. csv: Para leer y procesar archivos CSV.
3. tkinter.messagebox: Para mostrar mensajes de error o información al usuario.

Flujo del programa:

1. Abrir archivo CSV: Se solicita al usuario que seleccione un archivo CSV mediante un cuadro de diálogo.
2. Leer y procesar el archivo:

- Si se selecciona un archivo válido, se abre y procesa línea por línea.
- Los datos se convierten y se almacenan en la lista planetas.

3. Manejo de errores:

- Se controlan errores en la conversión de datos (Por ejemplo, si una celda no contiene un número válido).
- También se maneja cualquier excepción durante la apertura o lectura del archivo.

4. Actualizar visualización: Una vez cargados los datos, se llama a regenerar_planetas para reflejar los cambios.
5. Notificar al usuario: Se informa al usuario si la operación fue exitosa o si ocurrió un error.

```python
# Función para guardar los datos de planetas en un archivo CSV
def guardar_planetas_en_csv():
    # Abre un cuadro de diálogo para seleccionar la ubicación y el nombre del archivo.
    file_path = filedialog.asksaveasfilename(defaultextension=".csv", filetypes=[("CSV files", "*.csv")])

    if file_path:  # Verifica si el usuario seleccionó un archivo o canceló la operación.
        try:
            # Intenta abrir el archivo seleccionado en modo escritura.
            with open(file_path, mode='w', newline='') as csvfile:
                # Define los nombres de las columnas para el archivo CSV.
                fieldnames = ['Planeta', 'Radio', 'Velocidad Angular', 'Escala']

                # Crea un objeto DictWriter para escribir datos como diccionarios.
                writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

                # Escribe la fila de encabezados (nombres de las columnas) en el archivo CSV.
                writer.writeheader()

                # Itera sobre la lista "planetas" y escribe cada diccionario como una fila en el archivo CSV.
                for planeta in planetas:
                    writer.writerow(planeta)

                # Muestra un mensaje de éxito al usuario indicando que los datos se guardaron correctamente.
                messagebox.showinfo("Éxito", "Planetas guardados exitosamente en el archivo CSV.")
        except Exception as e:
            # Si ocurre un error al guardar el archivo, muestra un mensaje con detalles del error.
            messagebox.showerror("Error", f"No se pudo guardar el archivo CSV: {e}")
```

Variables:

1. file_path: Almacena la ruta absoluta del archivo que el usuario selecciona o crea mediante el cuadro de diálogo.
2. csvfile: Representa el archivo abierto para escritura en modo texto.
3. fieldnames: Lista de cadenas que define los nombres de las columnas en el archivo CSV. Corresponden a las claves de los diccionarios almacenados en planetas.
4. writer: Objeto csv.DictWriter utilizado para escribir datos como diccionarios en un archivo CSV.

5. planeta: Representa cada diccionario dentro de la lista planetas, el cual contiene los datos de un planeta específico.
6. e: Almacena la excepción capturada si ocurre un error al guardar el archivo.

Funciones:

1. filedialog.asksaveasfilename:

- Muestra un cuadro de diálogo para que el usuario seleccione o introduzca el nombre y la ubicación del archivo a guardar.
- El argumento defaultextension=".csv" asegura que el archivo tenga la extensión .csv por defecto.

2. csv.DictWriter: Crea un objeto que permite escribir filas en un archivo CSV basándose en diccionarios.
3. writer.writeheader: Escribe la fila de encabezados (Nombres de las columnas) en el archivo CSV.
4. writer.writerow: Escribe una fila en el archivo CSV basada en un diccionario.
5. messagebox.showinfo: Muestra un mensaje informativo indicando que los datos se guardaron con éxito.
6. messagebox.showerror: Muestra un mensaje de error si ocurre un problema al guardar el archivo.

Módulos utilizados:

1. tkinter.filedialog: Proporciona herramientas para mostrar cuadros de diálogo de selección de archivos o directorios.
2. csv: Maneja operaciones de lectura y escritura de archivos CSV.
3. tkinter.messagebox: Permite mostrar mensajes informativos o de error al usuario.

Flujo del programa:

1. Abrir cuadro de diálogo: El usuario selecciona o ingresa el nombre y la ubicación del archivo CSV.
2. Preparar el archivo CSV: El archivo se abre en modo escritura, y se define la estructura de las columnas.
3. Escribir datos en el archivo: Se escribe una fila de encabezados seguida de una fila por cada planeta en la lista planetas.
4. Notificar al usuario:

- Si los datos se guardaron correctamente, se muestra un mensaje de éxito.
- Si ocurre un error (Como permisos insuficientes o ruta no válida), se muestra un mensaje de error.

```python
# Configurar el gráfico principal
def configurar_grafico():
    global ax, info_frame  # Variables globales para configurar los elementos visuales del gráfico.

    # Limpiar la figura actual para eliminar elementos residuales.
    fig.clear()

    # Establecer el color de fondo de la figura a negro.
    fig.patch.set_facecolor("#2f2f2f")

    # Configuración del gráfico para modo 3D.
    if modo_3d.get():
        ax = fig.add_subplot(111, projection="3d")  # Crear un subplot con proyección 3D.

        # Establecer límites de los ejes en 3D.
        ax.set_xlim([-200, 200])
        ax.set_ylim([-200, 200])
        ax.set_zlim([-200, 200])

        # Título del gráfico en 3D.
        ax.set_title("Simulación 3D de Órbitas Planetarias", color="white")

        # Ocultar el cuadro de información en modo 3D si está definido.
        if 'info_frame' in globals():
            info_frame.place_forget()
    else:
        # Configuración del gráfico para modo 2D.
        ax = fig.add_subplot(111)  # Crear un subplot en 2D.

        # Establecer límites de los ejes en 2D.
        ax.set_xlim([-200, 200])
        ax.set_ylim([-200, 200])

        # Título del gráfico en 2D.
        ax.set_title("Simulación 2D de Órbitas Planetarias", color="white")

    # Color de fondo del área del gráfico.
    ax.set_facecolor("#2f2f2f")

    # Configuración del color de las etiquetas de los ejes (en blanco).
    if not modo_3d.get():
        for spine in ax.spines.values():
            spine.set_color("white")

    ax.tick_params(colors="white")  # Color de las marcas de los ejes.
    ax.yaxis.label.set_color("white")  # Color de la etiqueta del eje Y.
    ax.xaxis.label.set_color("white")  # Color de la etiqueta del eje X.

    # Configuración adicional para el eje Z en modo 3D.
```

```
# Configuración adicional para el eje Z en modo 3D.
if modo_3d.get():
    ax.zaxis.label.set_color("white")

# Configuración de la cuadrícula (color gris y línea punteada).
ax.grid(color="gray", linestyle="--")

# Representar el Sol como un punto amarillo en el centro.
ax.scatter(0, 0, color="yellow", s=200, label="Sol")

# Dibujar los cambios en el gráfico.
canvas.draw()
```

Variables:

1. ax (global):

- Representa el área del gráfico donde se dibujan los elementos.
- Puede configurarse en modo 2D o 3D según la opción seleccionada.

2. info_frame (global):

- Un marco que contiene información adicional (Probablemente elementos de la interfaz).
- Se oculta en modo 3D para evitar superposiciones no deseadas.

3. fig: Representa la figura principal de Matplotlib donde se dibujan los gráficos.
4. modo_3d: Variable vinculada a un control (Como un tk.BooleanVar) que indica si el modo 3D está habilitado.
5. canvas: Objeto que permite integrar gráficos de Matplotlib en la interfaz de Tkinter.

Funciones:

1. fig.clear(): Limpia la figura, eliminando gráficos anteriores y evitando residuos visuales.
2. fig.patch.set_facecolor(color): Cambia el color de fondo de la figura.
3. fig.add_subplot(): Añade un subplot (Una sección del gráfico) con soporte para 2D o 3D.
4. ax.set_xlim, ax.set_ylim, ax.set_zlim: Establecen los límites visibles en los ejes X, Y y Z, respectivamente.

5. ax.set_title(title, color): Establece el título del gráfico con un texto y color específicos.
6. ax.set_facecolor(color): Cambia el color del fondo del área del gráfico.
7. ax.spines.values(): Permite configurar las líneas que delimitan el área del gráfico (aplicable en 2D).
8. ax.tick_params(colors): Cambia el color de las marcas de los ejes.
9. ax.grid(color, linestyle): Activa la cuadrícula con un color y estilo de línea específicos.
10. ax.scatter(x, y, color, s, label): Dibuja un punto en el gráfico. Aquí se utiliza para representar al Sol.
11. canvas.draw(): Redibuja el gráfico para reflejar los cambios realizados.

Módulos utilizados:

1. matplotlib.pyplot (como plt): Se utiliza para configurar y crear gráficos en 2D o 3D.
2. mpl_toolkits.mplot3d: Habilita las herramientas necesarias para crear gráficos en 3D.
3. tkinter: Proporciona herramientas para la interfaz gráfica.
4. matplotlib.backends.backend_tkagg: Permite integrar gráficos de Matplotlib en aplicaciones basadas en Tkinter.

Flujo del programa:

1. Limpieza inicial: Elimina gráficos anteriores para preparar el área para nuevos elementos.
2. Configuración del modo gráfico: Define si el gráfico será en 2D o 3D, estableciendo límites y configuraciones específicas.
3. Personalización visual: Ajusta colores de fondo, títulos, etiquetas y cuadrículas para mejorar la visualización.
4. Añadir el Sol: Representa el Sol como un punto amarillo en el centro del gráfico.
5. Redibujar: Actualiza la interfaz gráfica para reflejar los cambios realizados.

```python
def rotacion_3d(punto, angulo, eje='z'):
    """
    Aplica una rotación en 3D a un punto alrededor de un eje especificado.

    Parámetros:
    - punto (tuple): Coordenadas originales del punto (x, y, z).
    - angulo (float): Ángulo de rotación en grados.
    - eje (str): Eje de rotación ('x', 'y' o 'z'). Por defecto es 'z'.

    Retorna:
    - nuevo_punto (numpy.ndarray): Coordenadas del punto rotado.
    """
    # Desempaquetar las coordenadas del punto.
    x, y, z = punto

    # Convertir el ángulo de grados a radianes para los cálculos trigonométricos.
    angulo_rad = np.radians(angulo)

    # Definir la matriz de rotación según el eje especificado.
    if eje == 'z':
        # Rotación alrededor del eje Z.
        matriz_rotacion = np.array([
            [np.cos(angulo_rad), -np.sin(angulo_rad), 0],
            [np.sin(angulo_rad), np.cos(angulo_rad), 0],
            [0, 0, 1]
        ])
    # Nota: Se pueden añadir matrices de rotación para los ejes X e Y.
    # Por ejemplo:
    # elif eje == 'x':
    #     matriz_rotacion = ...
    # elif eje == 'y':
    #     matriz_rotacion = ...

    # Aplicar la matriz de rotación al punto.
    nuevo_punto = matriz_rotacion @ np.array([x, y, z])

    # Retornar el punto rotado como un arreglo de Numpy.
    return nuevo_punto
```

Variables:

1. punto: Una tupla que contiene las coordenadas originales del punto en 3D: (x, y, z).
2. angulo: Ángulo de rotación en grados. Es convertido a radianes porque las funciones trigonométricas de Numpy operan en radianes.
3. eje: Indica el eje de rotación. En este caso, solo está implementado para el eje 'z'.

4. angulo_rad: Representa el ángulo de rotación en radianes.
5. matriz_rotacion: Matriz que define cómo rotar un punto en 3D alrededor del eje especificado.
6. nuevo_punto: Resultado de aplicar la matriz de rotación al punto original. Es un vector tridimensional rotado.

Módulos utilizados:

1. numpy:

- Se utiliza para manejar cálculos matriciales y trigonométricos:
    - np.radians(angulo): Convierte el ángulo de grados a radianes.
    - np.cos y np.sin: Calculan los valores del coseno y el seno para el ángulo.
    - np.array: Representa vectores y matrices de forma eficiente.
    - @ (Operador de producto matricial): Aplica la multiplicación de matrices entre la matriz de rotación y el vector del punto.

- Cómo funciona:

- Conversión del ángulo: Convierte el ángulo de grados a radianes usando np.radians para ser compatible con las funciones trigonométricas.

```python
def rotacion_2d(punto, angulo):
    """
    Aplica una rotación en 2D a un punto alrededor del origen.

    Parámetros:
    - punto (tuple): Coordenadas originales del punto (x, y).
    - angulo (float): Ángulo de rotación en grados.

    Retorna:
    - nuevo_punto (numpy.ndarray): Coordenadas del punto rotado.
    """
    # Desempaquetar las coordenadas del punto.
    x, y = punto

    # Convertir el ángulo de grados a radianes para los cálculos trigonométricos.
    angulo_rad = np.radians(angulo)

    # Definir la matriz de rotación para el plano 2D.
    matriz_rotacion = np.array([
        [np.cos(angulo_rad), -np.sin(angulo_rad)],
        [np.sin(angulo_rad), np.cos(angulo_rad)]
    ])

    # Aplicar la matriz de rotación al punto.
    nuevo_punto = matriz_rotacion @ np.array([x, y])

    # Retornar el punto rotado como un arreglo de Numpy.
    return nuevo_punto
```

Variables:

1. punto: Una tupla que contiene las coordenadas originales del punto en 2D: (x, y).
2. angulo: Ángulo de rotación en grados. Es convertido a radianes porque las funciones trigonométricas de Numpy operan en radianes.
3. angulo_rad: Representa el ángulo de rotación en radianes, calculado con np.radians(angulo).
4. matriz_rotacion: Matriz que define cómo rotar un punto en el plano 2D
5. nuevo_punto: Resultado de aplicar la matriz de rotación al punto original. Es un vector bidimensional rotado.

Módulos utilizados:

1. numpy:

- Se utiliza para manejar cálculos matriciales y trigonométricos:
    - np.radians(angulo): Convierte el ángulo de grados a radianes.
    - np.cos y np.sin: Calculan los valores del coseno y el seno para el ángulo.
    - np.array: Representa vectores y matrices de forma eficiente.
    - @ (Operador de producto matricial): Aplica la multiplicación de matrices entre la matriz de rotación y el vector del punto.

Cómo funciona:

1. Conversión del ángulo: El ángulo dado en grados se convierte a radianes utilizando np.radians.
2. Definición de la matriz de rotación: La matriz R es una matriz 2x2 que rota un punto alrededor del origen en el plano 2D.
3. Multiplicación matricial: Se multiplica la matriz de rotación R por el vector del punto [x,y], obteniendo las nuevas coordenadas del punto rotado.
4. Retorno del nuevo punto: Se devuelve el punto rotado como un arreglo de Numpy.

```python
def escalado_2d(punto, escala_x, escala_y):
    """
    Aplica un escalado en 2D a un punto.

    Parámetros:
    - punto (tuple): Coordenadas originales del punto (x, y).
    - escala_x (float): Factor de escala en el eje X.
    - escala_y (float): Factor de escala en el eje Y.

    Retorna:
    - nuevo_punto (numpy.ndarray): Coordenadas del punto escalado.
    """
    # Desempaquetar las coordenadas del punto.
    x, y = punto

    # Definir la matriz de escalado para el plano 2D.
    matriz_escalado = np.array([
        [escala_x, 0],
        [0, escala_y]
    ])

    # Aplicar la matriz de escalado al punto.
    nuevo_punto = matriz_escalado @ np.array([x, y])

    # Retornar el punto escalado como un arreglo de Numpy.
    return nuevo_punto
```

Variables:

1. punto: Una tupla que contiene las coordenadas originales del punto en 2D: (x, y).
2. escala_x: Factor de escala en el eje X. Si es mayor que 1, el punto se aleja del origen en la dirección del eje X. Si está entre 0 y 1, se acerca al origen.
3. escala_y: Factor de escala en el eje Y. Funciona de manera similar a escala_x, pero para el eje Y.
4. matriz_escalado: Matriz que define el escalado en 2D.
5. nuevo_punto:Resultado de aplicar la matriz de escalado al punto original. Es un vector bidimensional escalado.

Módulos utilizados:

1. numpy:

- Se utiliza para manejar cálculos matriciales de manera eficiente:
    - np.array: Representa vectores y matrices.
    - @ (Operador de producto matricial): Aplica la multiplicación de matrices entre la matriz de escalado y el vector del punto.

Cómo funciona:

1. Definición de la matriz de escalado: La matriz S es una matriz diagonal que contiene los factores de escala para los ejes X e Y.
2. Multiplicación matricial: Multiplica la matriz de escalado S por el vector del punto [x,y], obteniendo las nuevas coordenadas del punto escalado.
3. Retorno del nuevo punto: Devuelve el punto escalado como un arreglo de Numpy.

```python
def escalado_3d(punto, escala_x, escala_y, escala_z):
    """
    Aplica un escalado en 3D a un punto.

    Parámetros:
    - punto (tuple): Coordenadas originales del punto (x, y, z).
    - escala_x (float): Factor de escala en el eje X.
    - escala_y (float): Factor de escala en el eje Y.
    - escala_z (float): Factor de escala en el eje Z.

    Retorna:
    - nuevo_punto (numpy.ndarray): Coordenadas del punto escalado.
    """
    # Desempaquetar las coordenadas del punto.
    x, y, z = punto

    # Definir la matriz de escalado para 3D.
    matriz_escalado = np.array([
        [escala_x, 0, 0],
        [0, escala_y, 0],
        [0, 0, escala_z]
    ])

    # Aplicar la matriz de escalado al punto.
    nuevo_punto = matriz_escalado @ np.array([x, y, z])

    # Retornar el punto escalado como un arreglo de Numpy.
    return nuevo_punto
```

Variables:

1. punto: Una tupla que contiene las coordenadas originales del punto en 3D: (x, y, z).
2. escala_x: Factor de escala en el eje X. Si es mayor que 1, el punto se aleja del origen en la dirección del eje X. Si está entre 0 y 1, se acerca al origen.
3. escala_y: Factor de escala en el eje Y, funcionando de manera similar a escala_x.
4. escala_z: Factor de escala en el eje Z, funcionando de manera similar a los factores anteriores.
5. matriz_escalado: Matriz diagonal que define el escalado en 3D
6. nuevo_punto: Resultado de aplicar la matriz de escalado al punto original. Es un vector tridimensional escalado.

Módulos utilizados:

1. numpy:

- Utilizado para realizar cálculos matriciales de forma eficiente:
    - np.array: Representa vectores y matrices.
    - @ (Operador de producto matricial): Aplica la multiplicación de matrices entre la matriz de escalado y el vector del punto.

Cómo funciona:

1. Definición de la matriz de escalado: La matriz S es una matriz diagonal que contiene los factores de escala para los ejes X, Y, y Z.
2. Multiplicación matricial: Multiplica la matriz de escalado S por el vector del punto [x,y,z], obteniendo las nuevas coordenadas del punto escalado.
3. Retorno del nuevo punto: Devuelve el punto escalado como un arreglo de Numpy.

```python
def regenerar_planetas():
    """
    Regenera la visualización de los planetas y sus órbitas.
    Limpia los datos actuales y redibuja los planetas en el gráfico.

    Utiliza la configuración almacenada en la lista global de planetas.
    """
    global puntos, trayectorias, planetas, ax

    # Limpia el eje actual
    ax.cla()
    configurar_grafico()  # Reconfigura el gráfico (ejes, fondo, etc.)

    # Limpia las listas de puntos y trayectorias
    puntos.clear()
    trayectorias.clear()

    # Itera sobre los planetas definidos
    for planeta in planetas:
        a = planeta["Radio"]  # Semieje mayor de la órbita
        b = planeta["Radio"] * 0.8  # Semieje menor ajustado (elíptico)
        escala = planeta["Escala"]  # Factor de escala del planeta

        # Genera los ángulos para describir una órbita completa (360°)
        angulos = np.linspace(0, 2 * np.pi, 360)

        # Coordenadas de la órbita en el plano XY
        x_orbita = a * np.cos(angulos) * escala
        y_orbita = b * np.sin(angulos) * escala

        if modo_3d.get():
            # En 3D, las órbitas están en el plano XY, con Z constante
            z_orbita = np.zeros_like(x_orbita)  # Plano Z

            # Añade la trayectoria (línea) y el punto (planeta) al gráfico 3D
            trayectorias.append(ax.plot(x_orbita, y_orbita, z_orbita, linestyle="--", color="white")[0])
            puntos.append(ax.scatter([], [], [], label=planeta["Planeta"]))
        else:
            # En 2D, dibuja la órbita y el planeta en el plano XY
            trayectorias.append(ax.plot(x_orbita, y_orbita, linestyle="--", color="white")[0])
            puntos.append(ax.plot([], [], "o", label=planeta["Planeta"])[0])

    # Añade una leyenda con los nombres de los planetas
    ax.legend()

    # Redibuja el lienzo del gráfico actualizado
    canvas.draw()
```

Variables:

1. puntos: Lista global que contiene los objetos visuales que representan los planetas en el gráfico.
2. trayectorias: Lista global que almacena las trayectorias (Órbitas) de los planetas en forma de líneas.
3. planetas: Lista global con los datos de los planetas. Cada elemento es un diccionario con las siguientes claves:

- "Radio": Radio base de la órbita.

- "Escala": Factor de escala que ajusta el tamaño relativo de la órbita.
- "Planeta": Nombre del planeta.

4. ax: Objeto del eje actual en el que se dibuja el gráfico.
5. modo_3d: Variable global que indica si el modo 3D está activado (True) o no (False).

Módulos utilizados:

1. numpy (alias np):

- Generación de ángulos para describir la órbita en coordenadas polares: np.linspace.
- Cálculos matemáticos como coseno y seno: np.cos, np.sin.
- Manejo de arreglos de datos para definir las coordenadas X, Y (y Z en 3D).

2. matplotlib (ax):

- Gestión de gráficos 2D y 3D:

- ax.plot: Dibuja trayectorias (Órbitas).
- ax.scatter: Dibuja puntos (Planetas).
- ax.legend: Añade una leyenda descriptiva al gráfico.

Cómo funciona:

1. Limpieza del gráfico: Se elimina todo el contenido del eje actual con ax.cla() y se reconfigura el gráfico mediante configurar_grafico().
2. Limpieza de datos: Se vacían las listas globales puntos y trayectorias para evitar duplicados o basura visual.
3. Cálculo de órbitas:

- Genera puntos de una elipse con semiejes definidos por el radio del planeta y un ajuste (80% del radio en el eje menor).
- Los puntos se escalan según el valor de "Escala" del planeta.

4. Visualización 2D y 3D:

- Si el modo 3D está activo:

- Las órbitas se dibujan en el plano XY con z=0.

- Los planetas se añaden como puntos en el espacio 3D.

- En el modo 2D:

- Las órbitas y los planetas se representan en el plano XY.

5. Leyenda y actualización:

- Se añade una leyenda que muestra el nombre de cada planeta.
- Se redibuja el gráfico con canvas.draw().

```python
def actualizar(frame):
    """
    Actualiza la posición de los planetas en sus órbitas durante la animación.

    Parámetros:
    - frame (int): El número de fotograma actual para la animación (usado para calcular la posición de los planetas).
    """
    global puntos, planetas

    # Si se activa el evento de actualización, regenerar los planetas y sus órbitas
    if actualizar_event.get():
        regenerar_planetas()  # Regenera las órbitas y planetas si se activa la actualización
        actualizar_event.set(False)  # Desactiva la actualización

    # Actualiza la posición de cada planeta
    for i, planeta in enumerate(planetas):
        angulo_actual = (planeta["Velocidad Angular"] * frame) % 360  # Cálculo del ángulo en función de la velocidad angular y el fotograma
        a = planeta["Radio"]  # Semieje mayor de la órbita
        b = planeta["Radio"] * 0.8  # Semieje menor ajustado (órbita elíptica)
        escala = planeta["Escala"]  # Factor de escala de la órbita

        # Calcula la posición actual en la órbita elíptica en 2D
        x = a * np.cos(np.radians(angulo_actual)) * escala  # Coordenada X
        y = b * np.sin(np.radians(angulo_actual)) * escala  # Coordenada Y
        z = 0  # Z siempre igual a 0 en el plano 2D (se puede modificar si se añade inclinación)

        # Si el modo es 3D, actualiza las posiciones en 3D
        if modo_3d.get():
            puntos[i]._offsets3d = ([x], [y], [z])  # Actualiza las posiciones 3D
        else:
            puntos[i].set_data([x], [y])  # Actualiza las posiciones 2D

    # Redibuja el gráfico actualizado
    canvas.draw()

    # Retorna la lista de puntos actualizada
    return puntos
```

Variables:

1. frame: Número de fotograma actual utilizado para calcular la posición de los planetas a lo largo de sus órbitas. A medida que avanza la animación, este valor aumenta y las posiciones de los planetas cambian.
2. puntos: Lista global que contiene los objetos gráficos (Puntos) que representan a cada planeta. Se actualizan con las nuevas coordenadas de cada planeta en cada fotograma.
3. planetas: Lista global de diccionarios que contiene los parámetros de cada planeta (Como el nombre, radio, velocidad angular y escala). Esta lista se utiliza para calcular la nueva posición de cada planeta.

4. actualizar_event: Evento de control que, cuando está activado, indica que los planetas deben ser regenerados (Por ejemplo, si se cargan nuevos datos o se actualiza la configuración).

Módulos utilizados:

1. numpy (alias np):

- Utilizado para realizar cálculos matemáticos de forma eficiente:

- np.radians: Convierte el ángulo de grados a radianes.
- np.cos, np.sin: Calcula el coseno y seno de un ángulo, respectivamente, para obtener las coordenadas x y y de la órbita.

2. matplotlib (en particular, ax y canvas):

- puntos[i]: Cada elemento de la lista puntos es un objeto gráfico de matplotlib, ya sea un scatter (3D) o plot (2D).
- puntos[i].set_data([x], [y]): En el caso 2D, se actualizan las coordenadas de los puntos (planetas) en el gráfico.
- puntos[i]._offsets3d = ([x], [y], [z]): En el caso 3D, se actualizan las coordenadas de los puntos en el espacio 3D.
- canvas.draw(): Redibuja el gráfico con las nuevas posiciones de los planetas.

Cómo funciona:

1. Comprobación de actualización: Si el evento actualizar_event está activado (lo que podría ocurrir cuando se cambia la configuración o se cargan nuevos planetas), se llama a regenerar_planetas() para recrear las órbitas y planetas, y luego se desactiva el evento.
2. Cálculo de nuevas posiciones:

- Para cada planeta, el ángulo actual se calcula utilizando la velocidad angular del planeta y el fotograma actual.
- Se calculan las coordenadas x y y usando las ecuaciones de la órbita elíptica, y si el modo es 2D, se mantiene z=0

3. Actualización de las posiciones gráficas:

- En modo 3D, las posiciones se actualizan en 3D usando puntos[i]._offsets3d.

- En modo 2D, las posiciones se actualizan en 2D con puntos[i].set_data([x], [y]).

4. Redibujar el gráfico: Se llama a canvas.draw() para actualizar el gráfico con las nuevas posiciones de los planetas.
5. Retorno de la lista de puntos: La función retorna la lista de puntos, que ahora contiene las posiciones actualizadas de los planetas.

```
def ajustar_limites(valor=None):
    """
    Ajusta los límites del gráfico según el valor del zoom (controlado por un slider).

    Parámetros:
    - valor (float, opcional): El valor de zoom para ajustar los límites, si se proporciona.
                               Si no se proporciona, se usa el valor actual del slider.
    """
    global ax, slider_zoom

    # Obtener el valor actual del zoom desde el slider
    zoom = slider_zoom.get()  # El slider controla el nivel de zoom

    # Establecer los límites del gráfico 2D (X, Y) según el valor del zoom
    ax.set_xlim([-10 * zoom, 10 * zoom])  # Ajusta el límite del eje X
    ax.set_ylim([-10 * zoom, 10 * zoom])  # Ajusta el límite del eje Y

    # Si el gráfico está en modo 3D, ajustar también el límite del eje Z
    if modo_3d.get():
        ax.set_zlim([-10 * zoom, 10 * zoom])  # Ajusta el límite del eje Z

    # Redibuja el gráfico con los nuevos límites
    fig.canvas.draw_idle()
```

Variables:

1. zoom (int o float):

- Representa el nivel de zoom actual. Su valor proviene de un control deslizante (Slider) que permite al usuario ajustar el zoom del gráfico.
- El valor de zoom se usa para modificar los límites del gráfico.

2. slider_zoom: Es un control deslizante (Slider en Tkinter o similar) que se utiliza para ajustar el nivel de zoom. slider_zoom.get() devuelve el valor actual del control deslizante, que se usa para ajustar los límites del gráfico.
3. ax: Es el objeto de los ejes de matplotlib, sobre el que se dibujan los gráficos (Órbitas planetarias, en este caso). Se usa para modificar los límites de los ejes (X, Y, Z).

4. modo_3d: Es una variable que indica si el gráfico debe mostrarse en 3D o 2D. Si está activado (True), el gráfico se renderiza en 3D, y se ajustan los límites del eje Z.
5. fig: Es el objeto de la figura de matplotlib que contiene los ejes (ax) y los gráficos. Permite redibujar la figura al aplicar cambios en ella.

Módulos utilizados:

1. matplotlib (en particular, ax y fig):

- ax.set_xlim() y ax.set_ylim(): Ajustan los límites de los ejes X y Y del gráfico. Multiplican el valor de zoom por 10 para ampliar o reducir el área visible.
- ax.set_zlim(): Ajusta los límites del eje Z en gráficos 3D (si modo_3d es verdadero).
- fig.canvas.draw_idle(): Redibuja la figura con los nuevos límites. El método draw_idle() asegura que la actualización sea eficiente, sin necesidad de redibujar toda la figura innecesariamente.

Cómo funciona:

1. Obtener el valor del zoom: El valor de zoom se obtiene de un slider (Slider_zoom.get()). Este valor controla cuánto se expanden o se contraen los límites del gráfico. El zoom se utiliza para ajustar dinámicamente los límites de la visualización.
2. Ajustar límites 2D: ax.set_xlim() y ax.set_ylim() se utilizan para modificar los límites de los ejes X y Y. El valor del zoom se multiplica por 10 para determinar el rango visible en el gráfico. Si el zoom es 1, el gráfico mostrará de -10 a 10 en ambos ejes. Si el zoom es 2, el rango será de -20 a 20, y así sucesivamente.
3. Ajustar límites 3D (si es necesario): Si el gráfico está en 3D (Determinado por modo_3d.get()), también se ajustan los límites del eje Z utilizando ax.set_zlim(). Esto asegura que el gráfico mantenga una vista proporcional en los tres ejes.
4. Redibujar el gráfico: Después de ajustar los límites, fig.canvas.draw_idle() redibuja la figura para que los nuevos límites sean visibles de inmediato.

```python
def agregar_planetas():
    """
    Agrega nuevos planetas a la lista global de planetas y actualiza la visualización.

    Esta función permite agregar planetas de manera dinámica, incrementando un contador
    cada vez que se agrega un planeta nuevo. Los nuevos planetas se almacenan en la lista
    `planetas` y se actualiza la visualización.
    """
    global agregar_mas_planetas, contador_planetas, planetas
```

Variables:

1. agregar_mas_planetas (BooleanVar): Es una variable vinculada a un widget de control (Por ejemplo, un checkbox o un botón) en la interfaz gráfica. Indica si se debe agregar un nuevo planeta.

```python
def solicitar_cantidad():
    """
    Esta función se invoca cuando el usuario desea ingresar la cantidad de planetas a agregar. Si la entrada es válida,
    se pasa la cantidad y la opción de generación aleatoria a la siguiente ventana. Si la entrada es inválida, muestra un mensaje de error.
    """
    nonlocal ventana_solicitar

    try:
        # Obtener la cantidad de planetas ingresada por el usuario
        cantidad = int(entry_cantidad.get())
        aleatorios = var_aleatorios.get()  # Obtener el estado de la casilla de aleatorios
        if cantidad <= 0:
            raise ValueError  # Si la cantidad no es válida (menor o igual a 0), se lanza un error
        ventana_solicitar.destroy()  # Cierra la ventana de entrada de cantidad
        mostrar_ventana_agregar(cantidad, aleatorios)  # Llama a la función para agregar planetas
    except ValueError:
        # Si la cantidad no es un número válido o es menor o igual a 0, muestra un mensaje de error
        messagebox.showerror("Error", "Por favor, introduce un número válido.")

# Crear ventana para solicitar la cantidad de planetas
ventana_solicitar = tk.Toplevel(root)
ventana_solicitar.title("Cantidad de Planetas")  # Título de la ventana
ventana_solicitar.geometry("300x200")  # Resolución de la ventana

# Etiqueta informativa para el usuario
tk.Label(ventana_solicitar, text="¿Cuántos planetas deseas agregar?", font=("Unbounded ExtraBold", 9)).pack(pady=10)

# Campo de texto para que el usuario ingrese la cantidad de planetas
entry_cantidad = tk.Entry(ventana_solicitar, font=("Arial", 12))
entry_cantidad.pack(pady=5)

# Casilla para elegir si los datos serán generados aleatoriamente
var_aleatorios = tk.BooleanVar(value=False)  # Inicializa el valor de la casilla como False (no aleatorio)
tk.Checkbutton(
    ventana_solicitar,
    text="¿Generar datos aleatorios?",  # Texto que se muestra junto a la casilla
    variable=var_aleatorios,  # Variable asociada a la casilla (True si está marcada)
    font=("Unbounded ExtraBold", 9)
).pack(pady=10)

# Botón de aceptar para confirmar la cantidad de planetas
tk.Button(ventana_solicitar, text="Aceptar", command=solicitar_cantidad, font=("Martian Mono Condensed ExtraBold", 10), bg="#4f4f4f", fg="white").pack(pady=10)

ventana_solicitar.mainloop()  # Inicia el bucle principal de la ventana
```

Variables:

1. ventana_solicitar (Toplevel): Es una ventana secundaria creada con Tkinter, que solicita al usuario que ingrese la cantidad de planetas que desea agregar. Esta ventana también tiene una casilla para indicar si los planetas deben generarse con datos aleatorios.

2. entry_cantidad (Entry): Es un campo de entrada donde el usuario ingresa el número de planetas a agregar. Este valor será leído como un número entero en la función solicitar_cantidad().
3. var_aleatorios (BooleanVar):

- Es una variable booleana que se vincula a una casilla de verificación (Checkbutton). Permite al usuario seleccionar si los datos de los planetas deben ser generados aleatoriamente o no.
- Si el valor es True, los planetas serán generados con datos aleatorios; si es False, se utilizarán valores predefinidos o personalizados.

4. cantidad (int): Es el valor numérico que el usuario ingresa en el campo de texto entry_cantidad, que indica cuántos planetas desea agregar.
5. aleatorios (bool): Es el valor de la casilla de verificación, que indica si los planetas deben generarse con datos aleatorios. Se obtiene a partir de var_aleatorios.get().

Módulos utilizados:

1. tkinter (tk):

- Este módulo se utiliza para crear la interfaz gráfica de la ventana emergente. La función Toplevel crea una nueva ventana, Entry se utiliza para ingresar datos, Checkbutton para la casilla de verificación, y Button para el botón de aceptación.
- También se usan las funciones pack() para organizar los elementos en la ventana y mainloop() para iniciar el bucle de la interfaz.

2. messagebox (de Tkinter): Se utiliza para mostrar un cuadro de mensaje de error si el usuario ingresa una cantidad no válida (por ejemplo, un número negativo o un valor no numérico). El cuadro de mensaje se muestra con la función messagebox.showerror().
3. mostrar_ventana_agregar(cantidad, aleatorios): Esta es una función que no está definida en el fragmento, pero su propósito es manejar la lógica de agregar los planetas después de que el usuario ha ingresado una cantidad y una opción de datos aleatorios. Recibe como parámetros la cantidad de planetas y si los datos serán aleatorios o no.

Función solicitar_cantidad():

1. Obtener los valores del usuario:

- Se lee la cantidad de planetas ingresada por el usuario en entry_cantidad.get().

- Se obtiene el valor de la casilla de verificación var_aleatorios.get(), que indica si los planetas deben generarse con datos aleatorios.

2. Validación de la cantidad: Si el valor ingresado es menor o igual a cero o no es un número válido (Por ejemplo, una cadena de texto), se muestra un mensaje de error con messagebox.showerror().
3. Cierre de la ventana y llamada a la siguiente función: Si la cantidad es válida, se cierra la ventana de solicitud (Ventana_solicitar.destroy()) y se llama a la función mostrar_ventana_agregar() con la cantidad de planetas y el estado de aleatoriedad como parámetros.
4. Bucle de la ventana: El bucle ventana_solicitar.mainloop() asegura que la ventana permanezca activa hasta que el usuario interactúe con ella.

```python
def mostrar_ventana_agregar(cantidad, aleatorios):
    """
    Muestra la ventana para agregar planetas, generando planetas de forma aleatoria
    o usando datos específicos proporcionados por el usuario.

    Esta función se invoca después de que el usuario ha especificado la cantidad de planetas
    a agregar y si deben ser generados con datos aleatorios.

    Parámetros:
        cantidad (int): La cantidad de planetas que el usuario desea agregar.
        aleatorios (bool): Un valor booleano que indica si los planetas deben tener datos aleatorios
                           (True) o si deben ser ingresados de forma manual (False).
    """
    global contador_planetas
    contador_planetas = 0  # Inicializa el contador de planetas agregados

    if aleatorios:
        # Generar todos los planetas con datos aleatorios
        for _ in range(cantidad):
            nombre = f"Planeta_{contador_planetas + 1}"  # Nombre único para cada planeta
            # Generación de datos aleatorios para cada planeta
            radio = round(random.uniform(10, 500), 2)  # Radio aleatorio entre 10 y 500
            velocidad = round(random.uniform(0.1, 30), 2)  # Velocidad angular aleatoria entre 0.1 y 30
            escala = round(random.uniform(1, 3), 2)  # Escala aleatoria entre 1 y 3
            # Crear un diccionario con los datos del planeta y agregarlo a la lista de planetas
            planetas.append({"Planeta": nombre, "Radio": radio, "Velocidad Angular": velocidad, "Escala": escala})
            contador_planetas += 1  # Incrementar el contador de planetas agregados

        regenerar_planetas()  # Regenerar la visualización de los planetas en el gráfico
        messagebox.showinfo("Éxito", f"Se han agregado {cantidad} planetas con datos aleatorios exitosamente.")
        return  # Finaliza la función después de agregar planetas aleatorios
```

Variables:

1. contador_planetas (global): Es una variable global utilizada para llevar un conteo de cuántos planetas se han agregado hasta el momento. Se inicializa en 0 al inicio de cada ejecución de la función y se incrementa por cada planeta agregado.

2. cantidad (int): Es el parámetro que se pasa a la función y especifica cuántos planetas el usuario desea agregar. Se obtiene desde la ventana anterior donde el usuario introdujo este valor.
3. aleatorios (bool): Este parámetro indica si los planetas deben ser generados con datos aleatorios (True) o si deben ser proporcionados por el usuario manualmente (False).

Flujo de la función:

1. Inicialización de contador_planetas: Al comienzo de la función, contador_planetas se restablece a 0. Esto asegura que el contador de planetas se inicie desde cero cada vez que el usuario desea agregar nuevos planetas.
2. Generación de planetas aleatorios:

- Si el parámetro aleatorio es True, la función procede a generar planetas con características aleatorias.

Para cada uno de los planetas generados:

- Nombre: Se genera un nombre único para cada planeta usando el contador, por ejemplo, "Planeta_1", "Planeta_2", etc.
- Radio: Se genera un valor aleatorio entre 10 y 500, que representa el tamaño del planeta.
- Velocidad Angular: Se genera un valor aleatorio entre 0.1 y 30, que representa la velocidad a la que el planeta orbita.
- Escala: Se genera un valor aleatorio entre 1 y 3, que ajusta la escala del planeta en la visualización.

Cada planeta generado se agrega a la lista planetas como un diccionario que contiene todos estos datos.

3. Llamada a regenerar_planetas(): Después de agregar los planetas, se llama a la función regenerar_planetas() para actualizar la visualización de los planetas en el gráfico.
4. Mensaje de éxito: Al finalizar la adición de planetas, se muestra un cuadro de mensaje con messagebox.showinfo() informando al usuario que los planetas fueron agregados exitosamente.

Módulos utilizados:

1. random:

- El módulo random se usa para generar números aleatorios que se asignan a las propiedades de cada planeta (Radio, velocidad angular, escala).
- La función random.uniform(a, b) genera un número decimal aleatorio entre a y b, lo que permite que los valores para las características del planeta estén distribuidos en un rango.

2. messagebox (de Tkinter): Se utiliza para mostrar un cuadro de mensaje de éxito al usuario después de que los planetas han sido agregados.
3. regenerar_planetas(): Esta función es llamada para actualizar la visualización de los planetas en el gráfico una vez que los planetas han sido agregados o modificados.

```python
def guardar_datos():
    """
    Recupera los datos ingresados por el usuario, los valida, y los agrega a la lista de planetas.
    Si los datos son válidos, se actualiza la visualización de los planetas en el gráfico.

    Esta función es invocada cuando el usuario desea agregar un nuevo planeta a la simulación.
    La información se extrae de campos de texto en la interfaz de usuario (nombre, radio, velocidad angular, escala).

    Si los datos son válidos, se agrega un nuevo planeta a la lista de planetas.
    Si los datos no son válidos, se muestra un mensaje de error.
    """
    try:
        # Recuperar datos ingresados por el usuario desde los campos de entrada
        nombre = entry_nombre.get()  # Obtiene el nombre del planeta
        radio = float(entry_radio.get())  # Obtiene el radio y lo convierte a float
        velocidad = float(entry_velocidad.get())  # Obtiene la velocidad angular y lo convierte a float
        escala = float(entry_escala.get())  # Obtiene la escala y lo convierte a float

        # Agregar el nuevo planeta a la lista de planetas
        planetas.append({"Planeta": nombre, "Radio": radio, "Velocidad Angular": velocidad, "Escala": escala})

        # Llamar a la función para regenerar los planetas en el gráfico (actualizar visualización)
        regenerar_planetas()

    except ValueError:
        # Si ocurre un error de tipo (por ejemplo, datos no numéricos en campos de texto),
        # mostrar un cuadro de mensaje con el error
        messagebox.showerror("Error", "Datos inválidos. Por favor, revisa tus entradas.")
```

Variables involucradas:

- entry_nombre, entry_radio, entry_velocidad, entry_escala:
- Son los campos de texto en la interfaz gráfica donde el usuario ingresa los datos del nuevo planeta (Nombre, radio, velocidad angular, y escala).
- La función obtiene estos valores y los convierte en los tipos de datos correspondientes (Por ejemplo, float para radio, velocidad, y escala).

Flujo de la función:

1. Recuperación de datos ingresados: La función utiliza los métodos .get() de los campos de texto para obtener los valores que el usuario ha ingresado. Cada valor ingresado es almacenado en una variable: nombre, radio, velocidad, y escala.
2. Conversión a tipo adecuado: Los valores de radio, velocidad, y escala se convierten de cadenas de texto a números de punto flotante (Float) usando la función float(). Esto es necesario para que los valores puedan ser utilizados en cálculos y visualizaciones.
3. Validación de los datos: Si el usuario ha ingresado un valor que no puede convertirse en un número (Por ejemplo, texto en lugar de un número), se lanza una excepción ValueError, y en ese caso, se muestra un cuadro de mensaje con el error usando messagebox.showerror(). Esto le indica al usuario que debe revisar sus entradas.
4. Adición de un nuevo planeta:

- Si todos los datos son válidos, se agrega un nuevo planeta a la lista planetas. Cada planeta es representado como un diccionario que contiene:

- Planeta: El nombre del planeta.
- Radio: El radio del planeta (Número).
- Velocidad Angular: La velocidad de rotación del planeta (Número).
- Escala: El factor de escala para el gráfico (Número).

5. Actualización de la visualización: Después de agregar el nuevo planeta a la lista, se llama a la función regenerar_planetas(), que se encarga de actualizar el gráfico y reflejar el nuevo estado de los planetas en la interfaz.
6. Manejo de errores: Si ocurre un error en la conversión de los valores de texto a números (ValueError), la función maneja este error mostrando un cuadro de diálogo con el mensaje de error apropiado, informando al usuario que debe corregir los valores ingresados.

Módulos utilizados:

1. messagebox (de Tkinter): Se utiliza para mostrar un cuadro de mensaje en caso de que los datos ingresados sean inválidos (Por ejemplo, si el usuario ingresa texto en lugar de números).
2. regenerar_planetas() (función externa): Esta función se invoca después de agregar el nuevo planeta a la lista planetas. Su tarea es actualizar la visualización gráfica para reflejar la adición del nuevo planeta

```python
def siguiente_planeta():
    """
    Esta función es responsable de guardar los datos del planeta actual, cerrar la ventana de ingreso de datos
    y mostrar la ventana para agregar el siguiente planeta (en caso de que haya más planetas por agregar).

    Se invoca después de que el usuario ha ingresado los datos del planeta y presionado el botón para proceder
    al siguiente planeta. Primero, guarda los datos, luego cierra la ventana actual y presenta la ventana para
    ingresar los datos del siguiente planeta.

    Si es el último planeta, el proceso de agregar planetas termina.
    """
    nonlocal ventana_agregar  # Permite acceder a la variable ventana_agregar desde el ámbito externo

    # Guardar los datos del planeta actual
    guardar_datos()

    # Cerrar la ventana de agregar planetas actual
    ventana_agregar.destroy()

    # Mostrar la ventana para agregar el siguiente planeta
    mostrar_ventana_agregar(cantidad - 1, aleatorios)  # Llama a la función para mostrar la siguiente ventana
```

Variables involucradas:

- ventana_agregar: Es la ventana en la que el usuario ingresa los datos del planeta. Al llamar a ventana_agregar.destroy(), se cierra esta ventana.
- cantidad: Representa la cantidad total de planetas que el usuario desea agregar. Se decrementa en 1 al pasar al siguiente planeta, ya que la función mostrar_ventana_agregar() maneja el ciclo de agregar planetas.
- aleatorios: Es una variable booleana que indica si los datos de los planetas deben generarse aleatoriamente o no. Esta información se pasa a la función mostrar_ventana_agregar() para controlar la lógica de agregar planetas.

Flujo de la función:

1. Guardar los datos del planeta actual: La función comienza invocando guardar_datos(), que se encarga de almacenar el planeta ingresado por el usuario y actualizar la visualización de los planetas en el gráfico.
2. Cerrar la ventana actual: Luego, se llama a ventana_agregar.destroy() para cerrar la ventana en la que el usuario está ingresando los datos del planeta. Esta acción elimina la ventana de la interfaz de usuario.
3. Mostrar la ventana para el siguiente planeta: Finalmente, se llama a mostrar_ventana_agregar(cantidad - 1, aleatorios), pasando la cantidad restante de planetas por agregar (cantidad - 1) y la variable aleatorios para continuar con el proceso de agregar planetas. Si aún hay planetas por agregar, esta función volverá a mostrar la ventana de ingreso de datos para el siguiente planeta.

Módulos utilizados:

1. guardar_datos() (función externa): Guarda los datos ingresados para el planeta actual y actualiza la visualización de los planetas en el gráfico.
2. ventana_agregar.destroy() (Tkinter): Cierra la ventana actual de entrada de datos del planeta.
3. mostrar_ventana_agregar(cantidad - 1, aleatorios) (función externa): Muestra la ventana para agregar el siguiente planeta, pasando los parámetros necesarios para determinar cuántos planetas quedan por agregar.

```
def terminar_agregar():
    """
    Esta función guarda los datos del último planeta ingresado, cierra la ventana de agregar planetas
    y muestra un mensaje informando que todos los planetas han sido agregados exitosamente.

    Se invoca cuando el usuario ha terminado de ingresar todos los planetas que desea agregar,
    ya sea presionando el botón de "Terminar" o completando la última entrada.

    El proceso incluye:
    1. Guardar los datos del último planeta.
    2. Cerrar la ventana de agregar planetas.
    3. Mostrar un mensaje de éxito.
```

Variables involucradas:

1. ventana_agregar:: Es la ventana donde el usuario está ingresando los datos del planeta actual. Al llamar a ventana_agregar.destroy(), se cierra esta ventana.

```
def eliminar_planeta():
    """
    Esta función abre una ventana que permite al usuario ingresar el nombre de un planeta para eliminarlo
    de la lista de planetas. Si el planeta existe, se elimina, se actualiza la visualización y se muestra
    un mensaje de éxito. Si no se encuentra el planeta, se muestra un mensaje de error.

    La eliminación se realiza tras la confirmación del usuario. Los datos de los planetas y la visualización
    del gráfico se actualizan tras eliminar un planeta.

    El proceso incluye:
    1. Mostrar ventana de eliminación con un campo de entrada para el nombre del planeta.
    2. Confirmar si el planeta existe y eliminarlo de la lista.
    3. Actualizar la visualización de los planetas y el gráfico.
    4. Mostrar mensaje de éxito o error según el resultado de la búsqueda.
    """

    def confirmar_eliminacion():
        """
        Función interna que busca el planeta por su nombre en la lista de planetas.
        Si encuentra el planeta, lo elimina y actualiza la visualización.
        Si no lo encuentra, muestra un mensaje de error.
        """
        nombre = entry_nombre.get()  # Recuperar el nombre ingresado por el usuario
        for planeta in planetas:
            if planeta["Planeta"].lower() == nombre.lower():  # Buscar planeta por nombre (insensible a mayúsculas)
                planetas.remove(planeta)  # Eliminar planeta de la lista
                messagebox.showinfo("Éxito", f"Planeta '{nombre}' eliminado exitosamente.")  # Mensaje de éxito
                regenerar_planetas()  # Actualizar la visualización de los planetas
                canvas.draw()  # Redibujar el gráfico
                ventana_eliminar.destroy()  # Cerrar la ventana de eliminación
                return
        messagebox.showerror("Error", f"No se encontró un planeta con el nombre '{nombre}'.")  # Mensaje de error si no se encuentra el planeta

    # Crear una nueva ventana para que el usuario ingrese el nombre del planeta a eliminar
    ventana_eliminar = tk.Toplevel(root)  # Crear una ventana secundaria
    ventana_eliminar.title("Eliminar Planeta")  # Título de la ventana
    ventana_eliminar.geometry("300x150")  # Tamaño de la ventana

    # Etiqueta para indicar qué debe ingresar el usuario
    tk.Label(ventana_eliminar, text="Nombre del Planeta a Eliminar:", font=("Unbounded ExtraBold", 10)).pack(pady=10)

    # Campo de entrada para que el usuario escriba el nombre del planeta a eliminar
    entry_nombre = tk.Entry(ventana_eliminar, font=("Arial", 12))  # Entrada de texto
    entry_nombre.pack(pady=10)

    # Botón para confirmar la eliminación del planeta
    tk.Button(ventana_eliminar, text="Eliminar", command=confirmar_eliminacion, font=("Martian Mono Condensed ExtraBold", 9), bg="#4f4f4f", fg="white").pack(pady=10)
```

Variables involucradas:

- planetas: Es la lista que contiene todos los planetas. Se busca un planeta por su nombre para eliminarlo de esta lista.
- entry_nombre: Es el campo de entrada donde el usuario debe escribir el nombre del planeta que desea eliminar.
- ventana_eliminar: Es la ventana de Tkinter donde el usuario ingresa el nombre del planeta a eliminar.

Flujo de la función:

1. Mostrar ventana de eliminación: Se crea una ventana secundaria (ventana_eliminar) que solicita al usuario el nombre del planeta que desea eliminar.
2. Buscar y eliminar el planeta: La función confirmar_eliminacion() busca el planeta en la lista planetas utilizando el nombre proporcionado por el usuario. Si el planeta existe, se elimina de la lista y se actualiza la visualización del gráfico.
3. Actualización y mensajes:

- Si el planeta es encontrado y eliminado, se muestra un mensaje de éxito y se actualiza la visualización de los planetas con regenerar_planetas().
- Si no se encuentra el planeta, se muestra un mensaje de error indicando que no se ha encontrado un planeta con ese nombre.

4. Cierre de la ventana: Después de completar la eliminación, la ventana de eliminación se cierra automáticamente con ventana_eliminar.destroy().

Módulos utilizados:

- tkinter.messagebox: Para mostrar mensajes de éxito o error al usuario, según el resultado de la búsqueda y eliminación del planeta.
- regenerar_planetas() (función externa): Actualiza la visualización de los planetas en el gráfico después de eliminar uno.

```python
def on_hover(event):
    """
    Esta función se ejecuta cada vez que el cursor del mouse pasa por encima del gráfico.
    Detecta si el cursor está cerca de la órbita de algún planeta (en el caso de una visualización 2D)
    y muestra información sobre el planeta más cercano a la órbita donde el cursor está ubicado.
    La información del planeta se muestra en un panel derecho de la interfaz gráfica. Si el cursor está
    fuera de una órbita o no está cerca de ningún planeta, el panel con la información desaparece.
    Parámetros:
        event (Tkinter Event): Evento del mouse que contiene información sobre la posición del cursor en el gráfico.
    """
    global panel_derecho, info_frame, info_label, planetas  # Variables globales

    if not modo_3d.get() and event.inaxes == ax:  # Solo funciona en modo 2D y dentro de los límites del gráfico
        x_event, y_event = event.xdata, event.ydata  # Obtener la posición del mouse en el gráfico
        min_distance = float('inf')  # Iniciar la variable de distancia mínima como infinita
        closest_planet = None  # Inicializar el planeta más cercano como None

        # Verificar la cercanía del cursor a las órbitas elípticas
        for planeta in planetas:
            a = planeta["Radio"]  # Semieje mayor de la órbita
            b = a * 0.8  # Semieje menor (ajustable, en este caso es el 80% del semieje mayor)
            escala = planeta["Escala"]  # Factor de escala de la órbita
            # Escalar las coordenadas del cursor según la órbita
            x_scaled = x_event / (a * escala)
            y_scaled = y_event / (b * escala)
            # Evaluar la ecuación de la elipse (x^2/a^2 + y^2/b^2 = 1)
            distancia_a_orbita = abs(x_scaled**2 + y_scaled**2 - 1)

            if distancia_a_orbita < 0.05:  # Umbral de cercanía (ajustable) a la órbita
                if distancia_a_orbita < min_distance:  # Si es el planeta más cercano
                    min_distance = distancia_a_orbita
                    closest_planet = planeta
        # Mostrar información del planeta si está cerca de la órbita
        if closest_planet:
            # Si no existe el frame de información, lo crea
            if 'info_frame' not in globals():
                info_frame = tk.Frame(panel_derecho, bg="#ffffff", relief="solid", bd=1)
                info_label = tk.Label(info_frame, text="", font=("Arial", 12), bg="#ffffff", justify="left")
                info_label.pack(padx=5, pady=5)
            # Coloca el panel de información a la derecha de la ventana
            info_frame.place(relx=0.75, rely=0.85, anchor="center")
            # Actualiza el texto de la información con los datos del planeta más cercano
            info_label.config(text=f"Nombre: {closest_planet['Planeta']}\n"
                                   f"Velocidad: {closest_planet['Velocidad Angular']} g/s\n"
                                   f"Radio Orbital: {closest_planet['Radio']} ")
        else:
            # Si no se encuentra un planeta cerca, oculta el panel de información
            if 'info_frame' in globals():
                info_frame.place_forget()
```

Flujo de la función:

1. Detección de movimiento del cursor:

- La función es llamada cada vez que el cursor se mueve sobre el gráfico (evento on_hover).
- Se verifica si el modo 3D está desactivado y si el evento ocurrió dentro de los límites del gráfico (event.inaxes == ax).

2. Determinación de la cercanía a las órbitas:

- Se recorre la lista planetas para verificar si el cursor está cerca de la órbita de algún planeta.

- Para cada órbita de planeta, se calcula una medida de "distancia" entre el cursor y la órbita utilizando la ecuación de una elipse, donde a y b son los semiejes mayor y menor de la órbita, y escala es un factor de escala.
- Si la distancia calculada es pequeña (Menor a un umbral, ajustable con 0.05), se considera que el cursor está cerca de la órbita.

3. Visualización de la información:

- Si el cursor está cerca de una órbita, se muestra un panel con la información del planeta más cercano a esa órbita.
- Si no se encuentra un planeta cercano, el panel con la información desaparece.

4. Manejo del panel de información:

- El panel de información (info_frame) se crea solo cuando es necesario y se coloca en la parte derecha de la ventana (relx=0.75, rely=0.85).
- La información del planeta (nombre, velocidad angular y radio orbital) se actualiza en el panel.
- Si no hay planetas cerca del cursor, el panel se elimina de la vista con place_forget().

Variables involucradas:

- panel_derecho: Es el panel en el que se muestra la información del planeta en la interfaz gráfica.
- info_frame: Es el contenedor (frame) donde se coloca la información sobre el planeta.
- info_label: Es la etiqueta (Label) dentro de info_frame que muestra la información textual del planeta.
- planetas: Lista global que contiene los datos de los planetas, usada para verificar si el cursor está cerca de la órbita de algún planeta.

Eventos:

- event.xdata, event.ydata: Son las coordenadas del cursor dentro del gráfico (en el sistema de coordenadas del gráfico).

Componentes gráficos:

- info_frame.place(...): Coloca el panel de información en el gráfico, ajustando su posición según las coordenadas relativas (relx y rely).
- info_label.config(...): Actualiza el contenido textual de la etiqueta que muestra la información sobre el planeta.

```python
def salir_con_video():
    """
    Esta función muestra un video antes de cerrar el programa.

    El flujo de la función sería el siguiente:
    - Reproducir un video que se pueda mostrar en una ventana.
    - Cerrar la ventana principal del programa después de que el video termine.

    No se muestra ningún video en esta implementación, pero puedes integrarlo con alguna librería que permita mostrar videos, como `tkinter` o `pygame`.
    """
    # Aquí deberías agregar el código para reproducir un video
    # Ejemplo de código usando Tkinter para mostrar un video antes de salir (opcional)
    # Si no se desea video, simplemente se cerrará el programa después de un retraso.

    import time


    # Ahora que el video "terminó", cerramos la ventana principal.
    root.destroy()  # Cierra la ventana principal
```

- Reproducción del video: La función salir_con_video() puede incluir código que permita la reproducción de un video (Por ejemplo, usando bibliotecas como opencv o pygame para cargar y mostrar un archivo de video).
- Cierre de la ventana: Después de que termine el video o el tiempo de espera (Simulando el video), se llama a root.destroy() para cerrar la ventana principal de la aplicación, cerrando efectivamente el programa.

```python
def iniciar_interfaz():
    global planetas, puntos, trayectorias, ax, fig, canvas, actualizar_event, modo_3d, slider_zoom, root, panel_derecho, info_frame, info_label

    # Crear la ventana principal
    root = tk.Tk()
    root.title("Simulación de Órbitas Planetarias")  # Título de la ventana
    root.geometry("1500x900")  # Resolución de la ventana
    root.configure(bg="#e0f7fa")  # Color de fondo de la ventana principal

    # Variables globales
    actualizar_event = tk.BooleanVar(value=False)
    modo_3d = tk.BooleanVar(value=False)

    # Panel izquierdo con botones y texto
    panel_izquierdo = tk.Frame(root, bg="#81d4fa", width=200)
    panel_izquierdo.pack(side="left", fill="y")

    # Etiquetas de texto en el panel izquierdo
    tk.Label(panel_izquierdo, text="ORBITAS PLANETARIAS", font=("Unbounded ExtraBold", 20, "bold"), bg="#81d4fa", fg="white").pack(pady=20)
    tk.Label(panel_izquierdo, text=" ", font=("Arial", 16, "bold"), bg="#81d4fa", fg="white").pack(pady=5)
    tk.Label(panel_izquierdo, text="---------OPCIONES----------", font=("Unbounded ExtraBold", 16, "bold"), bg="#81d4fa", fg="white").pack(pady=5)

    # Botones en el panel izquierdo
    tk.Button(panel_izquierdo, text="      Cargar CSV      ", command=cargar_planetas_desde_csv, bg="#0288d1", fg="white", font=("EXCRATCH", 9, "bold")).pack(pady=9)
    tk.Button(panel_izquierdo, text="      Guardar CSV     ", command=guardar_planetas_en_csv, bg="#0288d1", fg="white", font=("EXCRATCH", 9)).pack(pady=9)
    tk.Button(panel_izquierdo, text=" Agregar Planetas", command=agregar_planetas, bg="#0288d1", fg="white", font=("EXCRATCH", 9)).pack(pady=9)
    tk.Button(panel_izquierdo, text=" Eliminar Planeta", command=eliminar_planeta, bg="#0288d1", fg="white", font=("EXCRATCH", 9)).pack(pady=9)

    # Panel derecho para mostrar información
    panel_derecho = tk.Frame(root, bg="#ffffff", width=400)
    panel_derecho.pack(side="right", fill="y")

    # Aquí puedes agregar más elementos, como el gráfico de las órbitas o más botones
    # Por ejemplo, el gráfico de las órbitas planetarias se puede integrar en un canvas, que se colocaría aquí.

    # Ejecutar la interfaz gráfica
    root.mainloop()
```

1. Ventana principal (root): Se crea la ventana principal con un tamaño de 1500x900 píxeles y un color de fondo #e0f7fa (Un tono claro de azul).

- Panel izquierdo: Se crea un panel en el lado izquierdo con un fondo #81d4fa (Un azul claro) y un ancho de 200 píxeles. En este panel se añaden varias etiquetas y botones:

- Título de la aplicación ("ORBITAS PLANETARIAS").
- Un subtítulo con la etiqueta "---------OPCIONES----------".
- Botones que permiten cargar, guardar planetas, agregar planetas, o eliminar planetas, cada uno con su comando respectivo.

- Panel derecho: El panel derecho es donde se puede agregar información adicional (Por ejemplo, detalles sobre el planeta seleccionado o mostrar gráficos de las órbitas). Este panel está vacío por ahora y puedes llenarlo con más contenido según sea necesario.
- Ciclo de la interfaz gráfica: La llamada a root.mainloop() inicia el ciclo principal de la interfaz gráfica de Tkinter, permitiendo que el usuario interactúe con la ventana.

```
def cambiar_modo():
    modo_3d.set(not modo_3d.get())
    configurar_grafico()
    regenerar_planetas()
    cambiar_modo_boton.config(
        text="Cambiar a 3D" if not modo_3d.get() else "Cambiar a 2D"
    )

cambiar_modo_boton = tk.Button(panel_izquierdo, text="     Cambiar a 3D     ", command=cambiar_modo, bg="#0288d1", fg="white", font=("EXCRATCH", 9))
cambiar_modo_boton.pack(pady=10)
#Texto
tk.Label(panel_izquierdo, text="------------------------------", font=("Unbounded ExtraBold", 16, "bold"), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="Proyecto elaborado por:", font=("Martian Mono Condensed ExtraBold", 10), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="Lizeth Montserrat Cerón Samperio", font=("Martian Mono Condensed ExtraBold", 10), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="Angel Abraham Higuera Pineda", font=("Martian Mono Condensed ExtraBold", 10), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="Abad Rey Lorenzo Silva", font=("Martian Mono Condensed ExtraBold", 10), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="Mia Paulina Moya Rivera", font=("Martian Mono Condensed ExtraBold", 10), bg="#81d4fa", fg="white").pack(pady=5)
tk.Label(panel_izquierdo, text="------------------------------", font=("Unbounded ExtraBold", 16, "bold"), bg="#81d4fa", fg="white").pack(pady=5)
#Boton de salir
tk.Button(
    panel_izquierdo,
    text="Salir",
    command=salir_con_video,
    bg="#0288d1",
    fg="white",
    font=("EXCRATCH", 9)
).pack(pady=10)


# Panel derecho para el zoom
panel_derecho = tk.Frame(root, bg="#e1f5fe")
panel_derecho.pack(side="right", fill="both", expand=True)

fig = plt.figure(figsize=(6, 6))
canvas = FigureCanvasTkAgg(fig, master=panel_derecho)
canvas.get_tk_widget().pack(side="left", fill="both", expand=True)

ax = fig.add_subplot(111)
configurar_grafico()
regenerar_planetas()

# Zoom
zoom_frame = tk.Frame(panel_derecho, bg="#e1f5fe")
zoom_frame.pack(side="right", fill="y", padx=10)

tk.Label(zoom_frame, text="Acercar", font=("EXCRATCH", 9), bg="#e1f5fe").pack()
```

```
    # LargO
    slider_zoom = tk.Scale(
        zoom_frame,
        from_=1,
        to=1000.0,
        resolution=0.1,
        orient="vertical",
        command=ajustar_limites,
        bg="#e1f5fe",
        fg="black",
        highlightbackground="#0288d1",
        length=800,
        font=("EXCRATCH", 10)
    )
    slider_zoom.set(1.0)
    slider_zoom.pack()

    tk.Label(zoom_frame, text="Alejar", font=("EXCRATCH", 9), bg="#e1f5fe").pack()

    info_frame = tk.Frame(panel_derecho, bg="#ffffff", relief="solid", bd=1)
    info_label = tk.Label(info_frame, text="", font=("Arial", 12), bg="#ffffff", justify="left")
    info_label.pack(padx=5, pady=5)
    info_frame.place_forget()


    anim = animation.FuncAnimation(fig, actualizar, frames=360, interval=50, blit=False)

    canvas.mpl_connect("motion_notify_event", on_hover)
    root.mainloop()
# inicia la interfaz
def main():
    iniciar_interfaz()

if __name__ == "__main__":
    main()
```

Módulos Importados:

En este código se usan varias bibliotecas de Python:

- tkinter: Utilizada para crear interfaces gráficas de usuario (GUIs). Proporciona widgets como botones, etiquetas, cuadros de texto, etc.
- matplotlib.pyplot: Utilizada para generar gráficos estáticos, animados e interactivos.
- FigureCanvasTkAgg: Parte de matplotlib, se usa para integrar gráficos generados con matplotlib en una ventana de tkinter.
- matplotlib.animation: Se usa para animaciones en gráficos, permitiendo que las visualizaciones sean dinámicas.

Variables:

Algunas variables clave en este código son:

- panel_izquierdo y panel_derecho: Son marcos (frames) dentro de la ventana principal (root). El panel izquierdo contiene controles como botones y etiquetas, mientras que el panel derecho contiene el gráfico y el control de zoom.
- modo_3d: Una variable utilizada para controlar si el gráfico se muestra en 2D o 3D.
- cambiar_modo_boton: Es un botón que cambia entre las vistas en 2D y 3D, llamando a la función cambiar_modo cuando se presiona.
- slider_zoom: Un control deslizante (Slider) vertical que permite acercar o alejar la vista del gráfico.
- info_frame y info_label: Un marco y una etiqueta que probablemente se usan para mostrar información adicional cuando el usuario interactúa con el gráfico.

Funciones:

cambiar_modo(): Esta función cambia entre los modos 2D y 3D. Utiliza el valor de modo_3d para determinar el modo actual y luego alterna entre ambos. También llama a configurar_grafico() para actualizar el gráfico y a regenerar_planetas() para actualizar los objetos dentro del gráfico. Luego, cambia el texto del botón para reflejar el modo actual (Ya sea "Cambiar a 3D" o "Cambiar a 2D").

iniciar_interfaz(): Esta es la función principal que configura la interfaz gráfica. Dentro de ella:

- Se define el panel izquierdo y el panel derecho.
- Se crean botones, etiquetas y sliders.
- Se configura el gráfico de matplotlib en el panel derecho.
- Se inicia la animación de la figura con animation.FuncAnimation(), lo que permite actualizar el gráfico dinámicamente.

on_hover(): Aunque no está completamente definido en el código, parece ser una función de controlador de eventos. Se utiliza para detectar eventos de movimiento del ratón sobre el gráfico, probablemente para mostrar información adicional en la etiqueta info_label.

ajustar_limites(): Tampoco está completamente definida, pero esta función probablemente se usa para ajustar los límites de zoom del gráfico en función del

valor del control deslizante (Slider_zoom). Esto ajustaría el nivel de acercamiento/alejamiento del gráfico.

configurar_grafico() y regenerar_planetas(): Aunque no se definen completamente, se espera que estas funciones configuren y actualicen los gráficos o planetas representados en el gráfico.

actualizar(): Esta es la función que se pasa a FuncAnimation y es llamada en cada frame de la animación para actualizar el gráfico en tiempo real.

salir_con_video(): Es una función que se ejecuta cuando el botón "Salir" es presionado, permitiendo cerrar el programa o detener un video.

4. Widgets de la Interfaz Gráfica: El código crea diferentes widgets de tkinter, que son elementos visuales dentro de la interfaz:

- Botón "Cambiar a 3D/2D": Cambia entre las vistas 2D y 3D cuando se presiona.
- Etiquetas (Label): Se usan para mostrar texto. Varias etiquetas se agrupan en el panel izquierdo para mostrar información sobre los autores del proyecto.
- Botón "Salir": Permite cerrar la aplicación.
- Slider de Zoom: Permite al usuario acercar o alejar el gráfico.

Animación y Gráficos:

En la parte derecha de la interfaz:

- Se crea un gráfico de matplotlib dentro de un marco (panel_derecho), y el gráfico se actualiza de manera continua con la animación creada por FuncAnimation.

Interacción con el Usuario:

La interfaz es interactiva. Algunas interacciones posibles son:

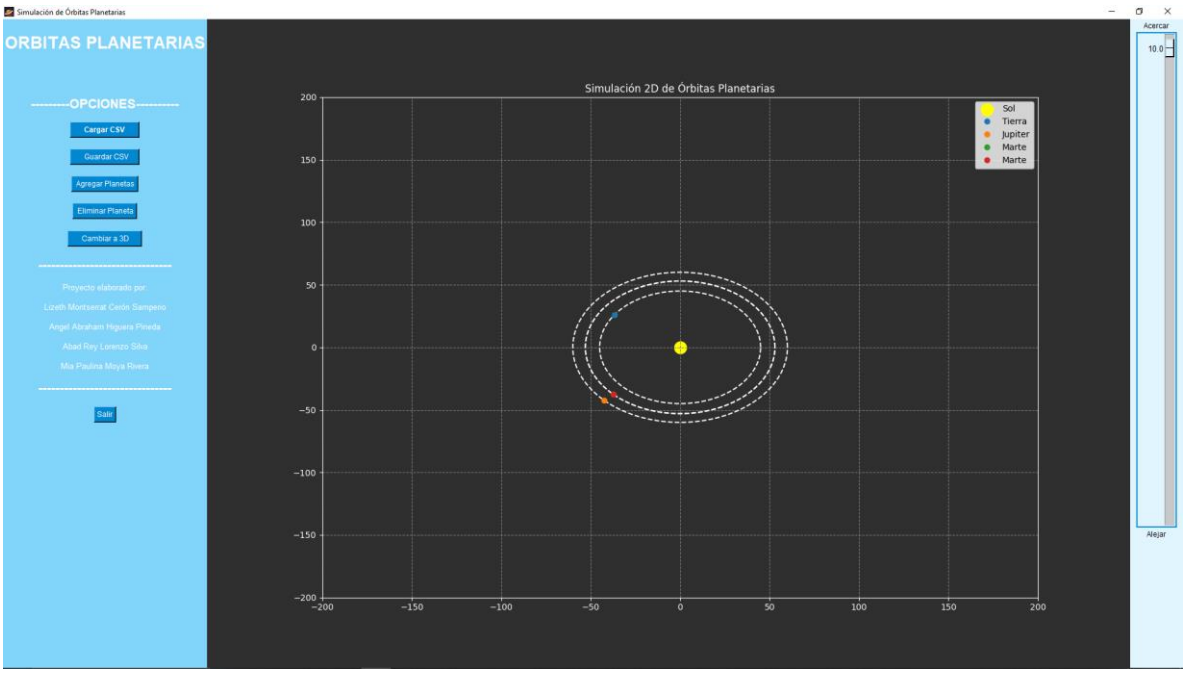
- Cambio de vista (2D a 3D): El botón permite alternar entre los modos 2D y 3D del gráfico.
- Zoom: El control deslizante permite acercar o alejar el gráfico.

Estructura de la Aplicación:

El flujo principal de la aplicación está controlado por la función main() que invoca iniciar_interfaz(). Esta última configura toda la interfaz y comienza el bucle principal de tkinter con root.mainloop().

- **Pruebas y resultados.**

Bidimensional:



1. **Visualización inicial**:

En la gráfica central, se observan las órbitas de los planetas ya cargados en la simulación. El Sol aparece en el centro como un punto amarillo, mientras que los planetas Tierra (Azul), Júpiter (Naranja) y Marte (Rojo) siguen trayectorias elípticas bien definidas alrededor de él.

Las trayectorias están representadas por líneas punteadas blancas para mayor claridad.

2. **Interacción con las opciones**:

Cargamos un archivo de datos preexistente con información de órbitas. Al hacerlo, la simulación se actualiza automáticamente para mostrar los planetas definidos en el archivo.

"Agregar Planetas": Añadimos un nuevo planeta ficticio, por ejemplo, "Saturno". Aparece una nueva órbita en la gráfica con un color distinto asignado automáticamente, y la leyenda se actualiza para incluirlo.

"Eliminar Planetas": Seleccionamos Marte para eliminarlo de la simulación. Desaparece su trayectoria y su nombre de la leyenda.

"Cambiar a 3D": Al activar esta opción, notamos que la simulación cambia de perspectiva, mostrándonos las órbitas en un espacio tridimensional. Esto ayuda a visualizar mejor la relación espacial entre los planetas.

3. **Uso de la barra de zoom**:

Ajustamos la barra deslizante en el panel derecho para acercar la vista y analizar con mayor detalle las órbitas más cercanas al Sol.

Posteriormente alejamos la vista para abarcar todo el sistema simulado.

4. **Resultados observados**:

Las órbitas son estables y se mueven de acuerdo con las leyes de Kepler, donde los planetas más cercanos al Sol tienen trayectorias más rápidas.

Los colores asignados facilitan distinguir a cada planeta, y la posición en tiempo real permite analizar su movimiento.

Usando estas fórmulas:

a) **Matriz de escalado** (S):

Esta ajusta los radios en x y y (en caso de órbitas elípticas):

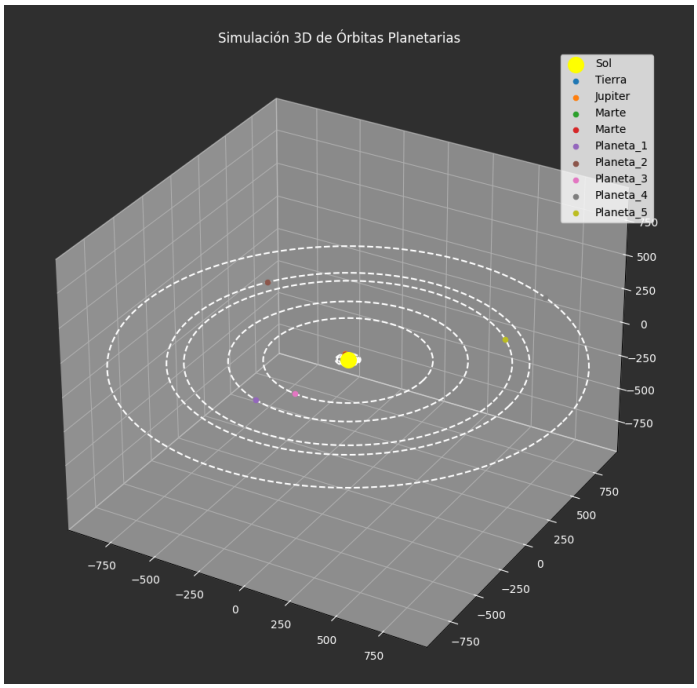
$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

b) **Matriz de rotación** ($R(\theta)$):

Esta rota los puntos según el ángulo θ:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Tridimensional:



La imagen muestra una simulación en 3D de órbitas planetarias. En esta representación, se visualizan varias trayectorias elípticas correspondientes a diferentes cuerpos celestes orbitando alrededor del Sol, que está ubicado en el centro del sistema.

Detalles del resultado:

1. **Elementos visualizados:**

El **Sol** está representado como un punto amarillo brillante en el centro del sistema.

Las órbitas son líneas elípticas representadas con trazos discontinuos en color blanco.

Hay varios planetas etiquetados, incluyendo:

- Planetas conocidos como **Tierra**, **Júpiter**, y **Marte**.
- Otros objetos denominados genéricamente como **Planeta_1**, **Planeta_2**, etc.

2. **Características técnicas:**

El espacio tridimensional está claramente delimitado por un sistema de coordenadas cartesianas (x, y, z), con un rango que abarca desde aproximadamente -750 a 750 unidades en cada eje.

La simulación tiene un fondo oscuro, lo que mejora la visibilidad de los elementos.

Una leyenda en la esquina superior derecha identifica los colores asignados a cada cuerpo celeste.

3. **Impresión general:**

La simulación parece ser un modelo simplificado de un sistema planetario. Podría ser útil para demostrar cómo orbitan los planetas alrededor de una estrella central en un espacio tridimensional.

Usando estas fórmulas:

a) Matriz de escalado (S):

En 3D, el escalado afecta los ejes x, y y z:

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

b) Matriz de rotación sobre el eje z ($R_z(\theta)$):

Para rotar un punto en el plano xy, usamos:

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Matriz de rotación sobre el eje x ($R_x(\phi)$):

Para inclinar la órbita (cambiar el plano), rotamos alrededor del eje x:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

- **Observaciones.**

- **Error en el ícono del programa**: Aunque se intentó establecer un planeta como símbolo representativo del programa, al instalarlo aparece una pluma de ave como ícono predeterminado, tanto en el escritorio o menú de aplicaciones como dentro de la pestaña del programa. No se logró reemplazar correctamente la imagen por defecto durante la configuración del programa. Este error puede que afecte la consistencia visual un poco en cuanto al tema del programa.
- **Necesidad de una actualización para mejorar la optimización**: Se observó que el programa podría beneficiarse de una actualización que optimice su rendimiento. Durante las pruebas, se identificaron áreas que requieren ajustes para hacer un uso más eficiente de los recursos computacionales, especialmente en la simulación en tiempo real. Mejorar la optimización garantizará una experiencia de usuario más fluida y eficiente, además de permitir la versatilidad para futuros desarrollos.

Conclusiones.

- Cerón Samperio Lizeth Montserrat.

Este proyecto destaca la importancia de integrar álgebra lineal y programación para abordar problemas científicos, como el estudio de órbitas planetarias. Nos permite a nosotros, los estudiantes aplicar conocimientos de matemáticas, física y desarrollo de software, fomentando una comprensión interdisciplinaria clave para resolver problemas complejos. La simulación orbital mediante matrices de transformación y visualización en tiempo real demuestra cómo los fundamentos matemáticos y computacionales se aplican a fenómenos reales, desarrollando habilidades técnicas esenciales para áreas emergentes como IA, visualización de datos y simulación física.

- Higuera Pineda Angel Abraham.

El uso de matrices de rotación y escalado es esencial para simular trayectorias orbitales de manera precisa y eficiente, modelando movimientos elípticos y variaciones angulares en tiempo real. Este enfoque matemático optimiza cálculos computacionales, evitando métodos más complejos como ecuaciones diferenciales. Además, demuestra cómo los conceptos básicos de álgebra lineal se aplican a problemas complejos, facilitando la comprensión de la dinámica orbital para desarrolladores y entusiastas de la física espacial.

- Lorenzo Silva Abad Rey.

Este proyecto destaca por su visualización en tiempo real, que permite analizar dinámicamente las trayectorias planetarias y observar cómo los cambios en parámetros, como la velocidad angular, afectan las órbitas. Esta funcionalidad tiene aplicaciones prácticas más allá del ámbito educativo, como la simulación de órbitas de satélites o el diseño de trayectorias espaciales. Además, facilita la identificación de patrones orbitales difíciles de analizar con métodos estáticos, ofreciendo una herramienta valiosa para validar teorías matemáticas, probar modelos físicos y desarrollar aplicaciones en navegación y exploración espacial.

- Moya Rivera Mia Paulina

La precisión numérica es crucial en simulaciones orbitales con matrices de transformación lineal, garantizando que las órbitas sean estables y periódicas. Este proyecto resalta cómo la acumulación de errores numéricos, como en rotaciones y escalados iterativos, puede distorsionar trayectorias, subrayando la importancia de

algoritmos eficientes y representaciones adecuadas. Además, permite explorar cómo las limitaciones computacionales, como la precisión de punto flotante, afectan los resultados y fomenta estrategias para minimizar errores, como normalización de vectores o uso de bibliotecas optimizadas, introduciendo a los estudiantes a desafíos del desarrollo científico en software.

Bibliografía.

Tim, T. W. [@TechWithTim]. (s/f). Planet simulation in python - tutorial. Youtube. Recuperado el 4 de enero de 2025, de https://www.youtube.com/watch?v=WTLPmUHTPqo

del Misterio, E. T. [@eltuneldelmisterio2022]. (s/f). *Python: Movimiento DE planetas*. Youtube. Recuperado el 4 de enero de 2025, de https://www.youtube.com/watch?v=WMYVVjq8HT0

Podcast, A. G.-A. &. [@alfonsogonzalez-astrodynam2207]. (s/f). *Propagar Órbitas con Runge-Kutta en Python, Ecuaciones Diferenciales | Mecánica Orbital con Python 4*. Youtube. Recuperado el 4 de enero de 2025, de https://www.youtube.com/watch?v=Kti7dskCuE0

Piogram [@Piogram]. (s/f). *1.1 Instalación y Configuración de Python en Visual Studio Code | Programar desde cero en Python*. Youtube. Recuperado el 4 de enero de 2025, de https://www.youtube.com/watch?v=-IyA_Yvs8IQ

The Code City [@TheCodeCity]. (s/f). *How to install PIP in visual studio code | PIP in VSCode (2024)*. Youtube. Recuperado el 4 de enero de 2025, de https://www.youtube.com/watch?v=ENHnfQ3cBQM