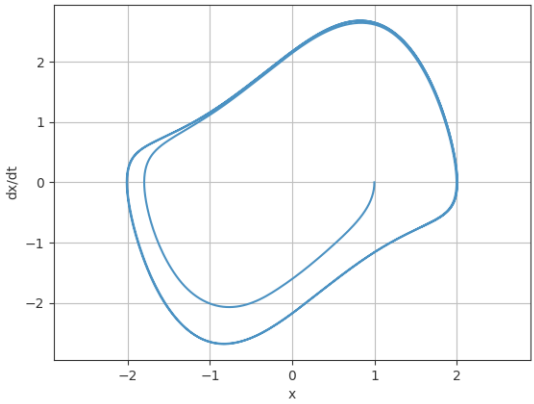
Resolución de ecuaciones diferenciales con SciPy

Simplificando la integración numérica de ecuaciones diferenciales ordinarias en Python

Ildeberto de los Santos Ruiz



Julio, 2025

Introducción

Las ecuaciones diferenciales son herramientas imprescindibles para describir el comportamiento y la evolución de sistemas dinámicos en campos tan diversos como la física, la ingeniería, la biología, la economía y las ciencias sociales. Estas ecuaciones establecen relaciones entre una función desconocida y sus derivadas, permitiendo modelar procesos continuos como el crecimiento poblacional, la propagación de ondas, la transmisión de calor o la oscilación de circuitos eléctricos. Mientras que en casos sencillos —por ejemplo, ecuaciones lineales de primer orden o sistemas con coeficientes constantes— es posible encontrar soluciones analíticas en forma de expresiones cerradas, la gran mayoría de los problemas reales conduce a ecuaciones no lineales o de orden superior que no admiten una solución exacta. Para enfrentar esta complejidad se emplean métodos numéricos, tales como el método de Euler, los métodos de Runge-Kutta o las técnicas de diferencias finitas, que permiten aproximar la solución con el grado de precisión deseado. Gracias a la potencia de cálculo de computadoras modernas, hoy es factible simular modelos con millones de ecuaciones diferenciales acopladas, lo que ha ampliado enormemente nuestra capacidad para predecir y controlar fenómenos complejos.

A la resolución numérica de las ecuaciones diferenciales que modelan un sistema físico se le nombra frecuentemente “simulación” del sistema.

En este artículo aprenderás cuándo es necesario recurrir a soluciones numéricas, cómo resolver ecuaciones diferenciales ordinarias (ODEs, del inglés Ordinary Differential Equations) usando el paquete SciPy mediante ejemplos prácticos paso a paso, y recibirás recomendaciones de buenas prácticas para garantizar resultados confiables.

¿Por qué recurrir a soluciones numéricas?

Las soluciones analíticas de las ecuaciones diferenciales —es decir, aquellas expresiones cerradas que describen exactamente el comportamiento del sistema— son muy valiosas, pero en la práctica presentan varias limitaciones:

- **Ausencia de una fórmula cerrada**: Muchas EDs, especialmente las no lineales o de orden superior, no admiten una solución exacta en términos de funciones elementales o especiales. Incluso cuando existen, dichas soluciones pueden ser tan complejas que resultan poco prácticas para su interpretación o

implementación directa.

- **Complejidad creciente**: Los modelos reales suelen incluir coeficientes variables, términos dependientes de múltiples variables o acoplamientos con otras ecuaciones, lo que hace inviable el cálculo simbólico. Condiciones de frontera o iniciales específicas (por ejemplo, datos experimentales ruidosos) pueden invalidar las hipótesis necesarias para obtener una solución exacta.
- **Flexibilidad y versatilidad**: Los métodos numéricos permiten adaptar el grado de aproximación al nivel de precisión requerido, ajustando tamaños de paso o tolerancias de error. Facilitan la incorporación de factores adicionales (retardos, discontinuidades, forzamientos externos) sin necesidad de reformular la teoría.
- **Potencia computacional disponible**: Con el hardware actual es posible resolver millones de ecuaciones simultáneamente, lo que abre la puerta al estudio de fenómenos de gran escala (modelos climáticos, circuitos distribuidos, redes biológicas). La paralelización y el uso de GPUs aceleran drásticamente el proceso de simulación.

Aquí es donde los métodos numéricos cobran protagonismo: emplean algoritmos iterativos que aproximan la solución de una ED con control de error, ajuste dinámico del paso y detección de eventos. Bibliotecas como SciPy —a través de funciones como `solve_ivp` y `odeint`— incorporan implementaciones optimizadas de esquemas de Runge–Kutta, Euler implícito/explicito y tamaño de paso adaptativo, permitiendo resolver sistemas rígidos y no rígidos de forma rápida, precisa y con un mínimo esfuerzo de codificación.

Requisitos previos

Antes de comenzar, asegúrate de tener instaladas las bibliotecas **NumPy**, **SciPy** y **Matplotlib**:

>_ Terminal

```
pip install numpy scipy matplotlib
```

Los integradores numéricos `solve_ivp` y `odeint` se acceden a través del subpaquete o módulo integrate de scipy. Mientras que odeint es una interfaz “clásica” a LSODA de ODEPACK, que ofrece una llamada sencilla pero con opciones limitadas (tolerancias fijas, sin soporte nativo para detección de eventos o elección de métodos), `solve_ivp` es la API moderna de SciPy que permite escoger entre varios integradores

(Runge–Kutta, BDF, ...), ajustar dinámicamente tolerancias, definir funciones de evento y recibir resultados estructurados, lo que ofrece mayor flexibilidad y control en la resolución numérica de ecuaciones diferenciales.

LSODA (*Livermore Solver for Ordinary Differential equations with Automatic method switching*) es un integrador de la librería ODEPACK que detecta dinámicamente si el sistema es "rígido" y cambia automáticamente entre el método de Adams (para sistemas no rígidos) y las fórmulas de diferencias backward (BDF, para sistemas rígidos), ofreciendo así robustez y eficiencia sin que el usuario tenga que elegir el método manualmente.

Un sistema *rígido* es aquel donde coexisten escalas de tiempo muy dispares, de modo que los métodos explícitos tradicionales (como el método de Euler) deben tomar pasos de integración extremadamente pequeños para mantener la estabilidad. Un ejemplo simple es un circuito RLC serie con una inductancia muy pequeña y una capacitancia muy grande, ya que las constantes de tiempo L/R y RC difieren varios órdenes de magnitud.

Para la resolución numérica de ODEs y el posterior análisis de sus soluciones se requerirá que en el script Python se importen las librerías y funciones necesarias:

```
import numpy as np
from scipy.integrate import solve_ivp, odeint
import matplotlib.pyplot as plt
```

A continuación se ejemplifica el uso de las funciones `solve_ivp` y `odeint` para resolver ODEs simples y sistemas de ODEs.

Ejemplo 1: Ecuación diferencial ordinaria simple

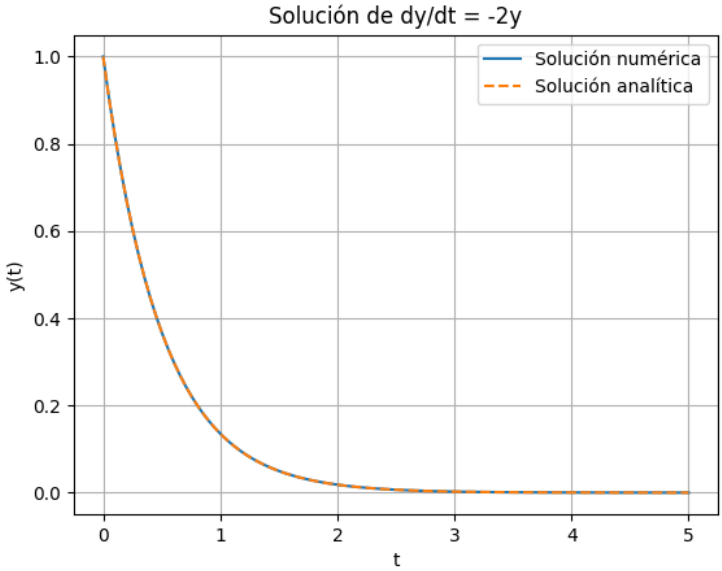
Resolvamos la siguiente ODE:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -2y, \text{ con la condición inicial } y(0) = 1.$$

Esta es una ecuación lineal de primer orden con solución analítica fácil de obtener, $y(t) = \exp(-2t)$, pero usaremos un método numérico para aproximarla.

ejemplo1.py

```python
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def dydt(t, y):
    return -2*y

t_span = [0, 5]  # Intervalo de tiempo
y0 = [1]         # Condición inicial

# Resolver
sol = solve_ivp(dydt, t_span, y0, t_eval=np.linspace(0, 5, 100))

# Graficar
plt.plot(sol.t, sol.y[0], label='Solución numérica')
plt.plot(sol.t, np.exp(-2*sol.t), '--', label='Solución analítica')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend()
plt.title('Solución de dy/dt = -2y')
plt.grid()
plt.show()
```



Una vez que tengas el objeto sol de `solve_ivp` (o el array de tiempos y soluciones de `odeint`), además de graficar la solución, puedes mostrar los datos en forma tabular:

```
tabla = np.column_stack((sol.t, sol.y[0]))
print(tabla)
```

Salida

```
[[0.00000000e+00 1.00000000e+00]
 [5.05050505e-02 9.03923887e-01]
 [1.01010101e-01 8.17053791e-01]
 ...
 [4.89898990e+00 5.59518455e-05]
 [4.94949495e+00 5.05809898e-05]
 [5.00000000e+00 4.57237894e-05]]
```

La llamada a `solve_ivp` genera dos arrays: `sol.t` (los tiempos) y `sol.y[0]` (los valores de la solución y). Con `np.column_stack` combinamos ambos en una sola matriz de dos columnas para visualizar la solución en forma tabular.

Para evaluar la solución en un valor específico de t, se puede buscar en los

resultados de la integración numérica mediante `np.interp` y devolver una estimación por interpolación lineal cuando el valor de t no coincide con los tiempos donde se muestreó la solución:

```
t_new = 2.345
y_new = np.interp(t_new, sol.t, sol.y[0])
print(f"Para t = {t_new:.3f}, y ≈ {y_new:.6f}")
```

Salida

```
Para t = 2.345, y ≈ 0.009222
```

Ejemplo 2: Sistema de ecuaciones diferenciales

Veamos ahora un sistema de dos ecuaciones diferenciales, como el modelo depredador-presa de Lotka-Volterra:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \alpha x - \beta x y$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\gamma y + \delta x y$$

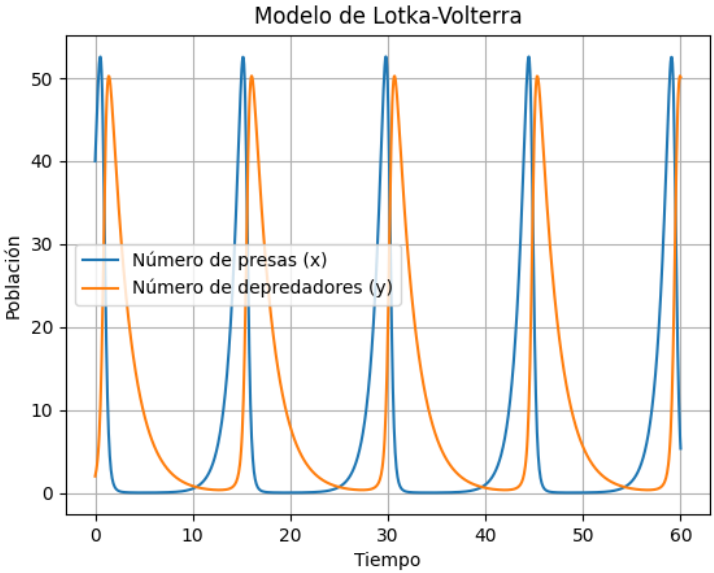
donde x es el número de presas (por ejemplo, gacelas en una reserva natural) mientras que y es el número de depredadores (por ejemplo, leones).

Solución con `odeint`:

ejemplo2.py

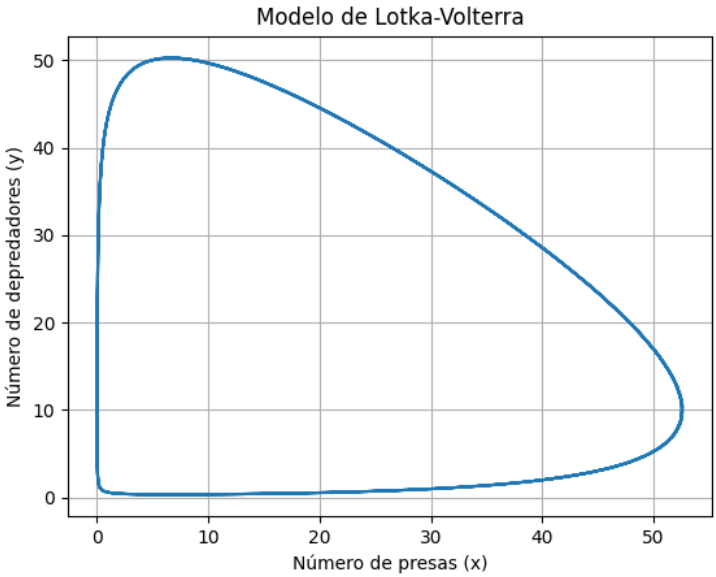
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def dydt(z, t, alpha, beta, gamma, delta):
    x, y = z
    dxdt = alpha*x - beta*x*y
    dydt = -gamma*y + delta*x*y
    return [dxdt, dydt]

# Parámetros
alpha = 1.0
beta = 0.1
```

```
gamma = 0.5
delta = 0.075

# Condiciones iniciales
z0 = [40, 2]

# Tiempo
t = np.linspace(0, 60, 1000)

# Resolver
sol = odeint(dydt, z0, t, args=(alpha, beta, gamma, delta))

# Graficar
plt.plot(t, sol[:,0], label='Número de presas (x)')
plt.plot(t, sol[:,1], label='Número de depredadores (y)')
plt.xlabel('Tiempo')
plt.ylabel('Población')
plt.legend()
plt.title('Modelo de Lotka-Volterra')
plt.grid()
plt.show()
```



En sistemas dinámicos donde el tiempo es la variable independiente, las soluciones de sistemas de ecuaciones diferenciales con dos funciones desconocidas (i.e., dos variables dependientes), frecuentemente se grafican en el plano de fase, además de las típicas gráficas respecto al tiempo. El **plano de fase** es una representación bidimensional en la que cada punto (x, y) refleja el estado simultáneo de las dos variables del sistema dinámico, trazando la trayectoria que recorre el par ordenado a medida que evoluciona en el tiempo. Al graficar estas trayectorias, los comportamientos periódicos se manifiestan como órbitas cerradas o ciclos límite, lo que permite identificar de un vistazo la amplitud, la frecuencia y la estabilidad de las oscilaciones. De este modo, el plano de fase facilita la comprensión visual de cómo interactúan las dos variables y cómo el sistema retorna a sus estados iniciales de forma repetitiva. Para mostrar el plano de fase con las soluciones del ejemplo anterior:

```
plt.plot(sol[:,0], sol[:,1])
plt.xlabel('Número de presas (x)')
plt.ylabel('Número de depredadores (y)')
plt.title('Modelo de Lotka-Volterra')
plt.grid()
plt.show()
```



Visualización de resultados

La visualización de resultados es fundamental para comprender y comunicar el comportamiento de tus soluciones numéricas de manera intuitiva. Con `matplotlib` puedes trazar la evolución temporal de cada variable —configurando colores, estilos de línea y leyendas claras— para identificar tendencias, oscilaciones o transitorios; generar diagramas de fase (representando y vs. x en sistemas bidimensionales) que revelan atractores, ciclos límite o puntos de equilibrio; superponer la solución numérica con la analítica cuando esté disponible, de modo que puedas evaluar el error y validar tu implementación; e incluso combinar varios subgráficos en una misma figura para comparar diferentes métodos, parámetros o condiciones iniciales. Además, puedes añadir anotaciones en puntos críticos, marcas de tiempo específicas, rejillas y exportar gráficos en alta resolución para informes o presentaciones, facilitando así la interpretación y el análisis riguroso de tus modelos.

Recomendaciones y buenas prácticas

Al efectuar tus simulaciones, considera estas recomendaciones y buenas prácticas para maximizar la precisión, la eficiencia y la reproducibilidad de tus resultados.

- **Documenta tu código:** Comenta claramente cada paso del modelo y la interpretación física o matemática.
- **Valida tu modelo:** Si conoces la solución analítica, compárala con la numérica. Usa `np.allclose()` para verificar diferencias pequeñas.
- **Ajusta la precisión:** Usa los parámetros `rtol` y `atol` en `solve_ivp` para controlar el error. Ejemplo: `solve_ivp(..., rtol=1e-6, atol=1e-9)`
- **Maneja las condiciones iniciales cuidadosamente:** Pequeños cambios en las condiciones iniciales pueden provocar grandes diferencias en sistemas caóticos.
- **Evita funciones discontinuas:** Si tu modelo incluye saltos o funciones no suaves, considera usar eventos o dividir el dominio.
- **Usa eventos para detectar condiciones:** `solve_ivp` permite definir funciones de evento para detener la integración bajo ciertas condiciones. Para que el integrador tenga en cuenta tu evento, pásalo al parámetro events de `solve_ivp`. Por ejemplo:

ejemplo3.py

```python
from scipy.integrate import solve_ivp

def dydt(t, y):
    return -y + t

def event(t, y):
    return y[0] - 0.5  # se dispara cuando y[0] == 0.5
event.terminal = True  # detiene la integración al ocurrir

t_span = (0, 10)
y0 = [0]
sol = solve_ivp(dydt, t_span, y0, events=event,
    dense_output=True)

print("Evento en t =", sol.t_events[0])
```

Salida

```
Evento en t = [1.19834189]
```

Comentarios finales

Además de `solve_ivp` y `odeint`, el módulo `scipy.integrate` ofrece herramientas adicionales para distintos tipos de integración numérica, como `quad` y `solve_bvp`, que quedan fuera del alcance de este tutorial. La función `quad` no está diseñada para resolver ecuaciones diferenciales, sino que actúa como un integrador de cuadratura para aproximar el valor de integrales definidas de funciones de una variable con alta precisión; por su parte, `solve_bvp` está especializada en problemas de valores en la frontera (*boundary value problems*), resolviendo sistemas de ODEs cuando se imponen condiciones en los extremos del intervalo —por ejemplo, para la distribución de temperatura en una barra o el perfil de velocidad en un flujo— y empleando métodos de colocación o diferencias finitas para hallar la solución que satisfaga simultáneamente todas las condiciones de contorno.

Para profundizar en la resolución numérica de ODEs con SciPy, explora la sección de `scipy.integrate` en la documentación oficial, prueba ejemplos interactivos en notebooks —cambiando métodos, tolerancias y detección de eventos— y consulta recursos como el *SciPy Cookbook* para ver casos de uso reales; así irás conociendo a fondo las capacidades de `solve_ivp`, `odeint` y otros solvers para aplicarlos con confianza en tus proyectos.

Conclusiones

La resolución numérica de ecuaciones diferenciales con Python es una competencia fundamental para científicos e ingenieros, pues permite simular con pocas líneas de código desde sistemas sencillos hasta modelos complejos, aprovechando la potencia de **NumPy** y **SciPy**. Emplear `solve_ivp` con métodos adaptativos suele ofrecer la mayor versatilidad, y contrastar los resultados con soluciones analíticas o datos experimentales ayudará a validar tu implementación. La visualización de trayectorias y diagramas de fase facilita la detección de comportamientos críticos, y ajustar las tolerancias de los algoritmos permite equilibrar precisión y velocidad de cálculo. Escribir código modular y bien documentado garantiza que tus simulaciones sean reproducibles, fáciles de mantener y estén preparadas para modificaciones futuras.

Ildeberto de los Santos Ruiz es originario de Tonalá, Chiapas (1973). Ingeniero en Electrónica, Maestro en Ciencias en Ingeniería Mecatrónica y Doctor en Ciencias de la Ingeniería por el Instituto Tecnológico de Tuxtla Gutiérrez (ITTG); Doctor en Automática, Robótica y Visión por la Universidad Politécnica de Cataluña. Miembro de la *Association for Computing Machinery* (ACM) y del *Institute of Electrical and Electronics Engineers* (IEEE). Miembro afiliado de la *International Federation of Automatic Control* (IFAC). Profesor de tiempo completo en el ITTG desde 1995, adscrito al Departamento de Ingeniería Eléctrica y Electrónica, Jefe de Proyectos de Investigación de Ingeniería Mecatrónica y Presidente del Claustro del Doctorado en Ciencias de la Ingeniería. Dirige proyectos de investigación en las áreas de diagnóstico y control inteligente, específicamente en robótica y en detección/localización de fugas en redes de distribución de agua.