Universidad Autónoma de San Luis Potosí

Facultad de Estudios Profesionales Zona Media

Práctica 9: Resolucion de ecuaciones diferenciales

Nombre del alumno: Sergio Adolfo Juárez Mendoza

Clave: 369376

Profesor: Ing. Jesús Padrón

Fecha de entrega: 5 de mayo del 2025

.

Introducción

Realizar los siguientes ejercicios apoyados de software, puede ser python, maple, matlab, etc. Se pueden usar librerias(específicar su uso en el reporte) o crear cada algoritmo

.

Desarrollo

1 problema 1 (5.1)

```

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


# Definiciones de los problemas
def problema_a(t, y):
    return y * np.cos(t)


def problema_b(t, y):
    return 2 / t * y + t ** 2 * np.exp(t)


def problema_c(t, y):
    return -2 / t * y + t ** 2 * np.exp(t)


def problema_d(t, y):
    return 4 * t ** 3 * y / (1 + t ** 4)


# Lista de problemas con sus condiciones
problemas = [
    {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "Problema a"},
    {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "Problema b"},
    {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "nombre": "Problema c"},
    {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "Problema d"}
]

# Tabla de resultados
print("RESULTADOS NUM RICOS:")
```

```
print("-" * 50)
print(f"{'Problema':<12} {'y(a)':>8} {'y(b)':>15} {'Intervalo':>12}")
print("-" * 50)

# Resolver y graficar
for p in problemas:
    a, b = p["intervalo"]
    solucion = solve_ivp(p["func"], (a, b), p["y0"], dense_output=True)

    t_vals = np.linspace(a, b, 100)
    y_vals = solucion.sol(t_vals)[0]

    # Mostrar resultado num rico
    print(f"{p['nombre']:<12} {p['y0'][0]:>8.4f} {y_vals[-1]:>15.6f} [{a},{b}]")

    # Graficar
    plt.plot(t_vals, y_vals, label=p["nombre"])

# Gr fico
plt.title("Soluciones de los Problemas de Valor Inicial")
plt.xlabel("t")
plt.ylabel("y(t)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 1: semaforo.c

Definición del Código

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

Estas bibliotecas permiten:

- `numpy`: generar arreglos y realizar operaciones numéricas.
- `matplotlib.pyplot`: crear gráficos.
- `solve_ivp` de `scipy.integrate`: resolver numéricamente EDOs.

Definición de las Ecuaciones Diferenciales

Se definen cuatro funciones $f(t, y)$ que representan diferentes EDOs:

```python
def problema_a(t, y):
    return y * np.cos(t)

def problema_b(t, y):
    return 2 / t * y + t ** 2 * np.exp(t)

def problema_c(t, y):
    return -2 / t * y + t ** 2 * np.exp(t)

def problema_d(t, y):
    return 4 * t ** 3 * y / (1 + t ** 4)
```

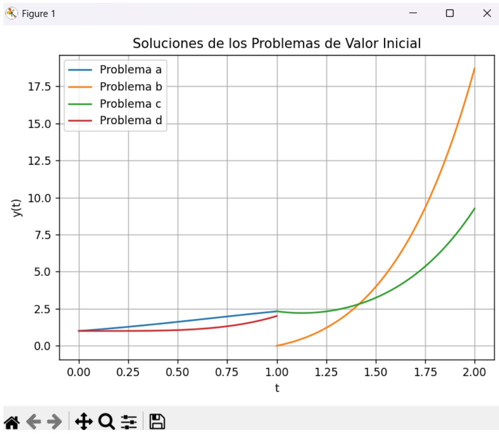
Configuración de los Problemas

Cada problema se describe mediante:

- Una función $f(t, y)$
- Un intervalo de integración $[a, b]$
- Una condición inicial $y(a) = y_0$

```python
problemas = [
    {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "Problema a"},
    {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "Problema b"},
    {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "nombre": "Problema c"},
    {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "Problema d"}
]
```

```
RESULTADOS NUMÉRICOS:
--------------------------------------------------
Problema         y(a)            y(b)    Intervalo
--------------------------------------------------
Problema a     1.0000        2.319906 [0, 1]
Problema b     0.0000       18.682939 [1, 2]
Problema c     2.3316        9.244998 [1, 2]
Problema d     1.0000        2.006274 [0, 1]
```



2 problema 7(5.1)

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

# f(t, y) = -y + t + 1
def f(t, y):
    return -y + t + 1

# y0(t): constante
def y0(t):
    return 1

# y1(t)
def y1(t):
    integrand = lambda tau: f(tau, y0(tau))
    return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])

# y2(t)
def y2(t):
    y1_vals = y1(t)
    integrand = lambda tau: f(tau, np.interp(tau, t, y1_vals))
    return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])

# y3(t)
def y3(t):
    y2_vals = y2(t)
    integrand = lambda tau: f(tau, np.interp(tau, t, y2_vals))
    return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])

# Soluci n real
def y_real(t):
    return t + np.exp(-t)
```

```

# Dominio de t
t_vals = np.linspace(0, 1, 100)

# Evaluaciones
y0_vals = np.ones_like(t_vals)
y1_vals = y1(t_vals)
y2_vals = y2(t_vals)
y3_vals = y3(t_vals)
y_real_vals = y_real(t_vals)

# Mostrar tabla de resultados en t = 1
t1 = 1.0
y_real_1 = y_real(t1)
y1_1 = y1(np.array([t1]))[0]
y2_1 = y2(np.array([t1]))[0]
y3_1 = y3(np.array([t1]))[0]

print("RESULTADOS EN t = 1")
print("-" * 40)
print(f"{'Aproximaci n':<15} {'Valor':>10} {'Error absoluto':>15}")
print("-" * 40)
print(f"{' y  (t)':<15} {1:>10.6f} {abs(1 - y_real_1):>15.6f}")
print(f"{' y  (t)':<15} {y1_1:>10.6f} {abs(y1_1 - y_real_1):>15.6f}")
print(f"{' y  (t)':<15} {y2_1:>10.6f} {abs(y2_1 - y_real_1):>15.6f}")
print(f"{' y  (t)':<15} {y3_1:>10.6f} {abs(y3_1 - y_real_1):>15.6f}")
print(f"{'Exacta':<15} {y_real_1:>10.6f} {'-'*15}")

# Gr fica
plt.plot(t_vals, y0_vals, label=' y  (t)', linestyle='--')
plt.plot(t_vals, y1_vals, label=' y  (t)')
plt.plot(t_vals, y2_vals, label=' y  (t)')
plt.plot(t_vals, y3_vals, label=' y  (t)')
plt.plot(t_vals, y_real_vals, label='Soluci n exacta', color='black',
    linestyle='dotted')

plt.title("Aproximaciones por el M todo de Picard")
plt.xlabel("t")
plt.ylabel("y(t)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 2: semaforo.c

beginlstlisting import numpy as np import matplotlib.pyplot as plt from scipy.integrate import solve$_ivp$

Estas bibliotecas permiten:

- `numpy`: generar arreglos y realizar operaciones numéricas.
- `matplotlib.pyplot`: crear gráficos.
- `solve_ivp` de `scipy.integrate`: resolver numéricamente EDOs.

Definición de las Ecuaciones Diferenciales

Se definen cuatro funciones $f(t, y)$ que representan diferentes EDOs:

```
def problema_a(t, y):
    return y * np.cos(t)

def problema_b(t, y):
    return 2 / t * y + t ** 2 * np.exp(t)

def problema_c(t, y):
    return -2 / t * y + t ** 2 * np.exp(t)

def problema_d(t, y):
    return 4 * t ** 3 * y / (1 + t ** 4)
```

Configuración de los Problemas

Cada problema se describe mediante:

- Una función $f(t, y)$
- Un intervalo de integración $[a, b]$
- Una condición inicial $y(a) = y_0$

```
problemas = [
    {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "
   Problema a"},
    {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "
   Problema b"},
    {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "
   nombre": "Problema c"},
    {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "
   Problema d"}
]
```

Resolución y Resultados

Cada problema se resuelve usando `solve_ivp`. Se evalúa la solución en 100 puntos igualmente espaciados en el intervalo dado:

```
for p in problemas:
    a, b = p["intervalo"]
    solucion = solve_ivp(p["func"], (a, b), p["y0"], dense_output=True)

    t_vals = np.linspace(a, b, 100)
    y_vals = solucion.sol(t_vals)[0]

```

```
    print(f"{p['nombre']:<12} {p['y0'][0]:>8.4f} {y_vals[-1]:>15.6f} [{a},
    {b}]")
    plt.plot(t_vals, y_vals, label=p["nombre"])
```

Este fragmento también:

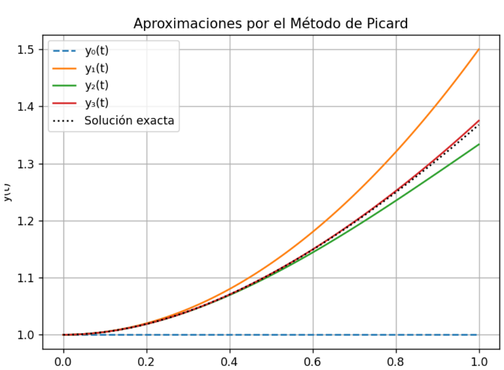
- Imprime el valor inicial y el valor aproximado final $y(b)$
- Grafica la solución para cada problema

Gráfica Final

El siguiente código genera el gráfico con todas las soluciones aproximadas:

```
plt.title("Soluciones de los Problemas de Valor Inicial")
plt.xlabel("t")
plt.ylabel("y(t)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
RESULTADOS EN t = 1
---------------------------------------
Aproximación          Valor  Error absoluto
---------------------------------------
y₀(t)              1.000000        0.367879
y₁(t)              1.500000        0.132121
y₂(t)              1.000000        0.367879
y₃(t)              1.500000        0.132121
Exacta             1.367879 --------------
```



3 problema 1 (5.2)

```python
import numpy as np
import matplotlib.pyplot as plt


def metodo_euler(f, t0, y0, h, pasos):
    """
    Aproxima la soluci n de una EDO usando el m todo de Euler.

    Par metros:
    - f: Funci n dy/dt = f(t, y)
    - t0: Valor inicial de t
    - y0: Valor inicial de y
    - h: Tama o del paso
    - pasos: N mero de pasos

    Retorna:
    - t_valores: Array de valores de t
    - y_valores: Array de valores aproximados de y
    """
    t_valores = np.zeros(pasos + 1)
    y_valores = np.zeros(pasos + 1)

    t_valores[0] = t0
    y_valores[0] = y0

    for i in range(pasos):
        t = t_valores[i]
        y = y_valores[i]
        y_valores[i + 1] = y + h * f(t, y)
        t_valores[i + 1] = t + h

    return t_valores, y_valores


# Definici n de las EDOs para cada inciso
def f_a(t, y):
    return t * np.exp(3 * t) - 2 * y  # y' = te^(3t) - 2y


def f_b(t, y):
    return 1 + (t - y) ** 2  # y' = 1 + (t - y)^2


def f_c(t, y):
    return 1 + y / t  # y' = 1 + y/t (asumo que "yh" es un typo y deber a ser y/t)


def f_d(t, y):
    return np.cos(2 * t) + np.sin(3 * t)  # y' = cos(2t) + sin(3t)


# Configuraci n para cada problema
problemas = [
```

```
    {"f": f_a, "t0": 0, "y0": 0, "h": 0.5, "t_final": 1, "nombre": "a) y' = te^{3t} - 2y"},
    {"f": f_b, "t0": 2, "y0": 1, "h": 0.5, "t_final": 3, "nombre": "b) y' = 1 + (t - y)^2"},
    {"f": f_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "nombre": "c) y' = 1 + y/t"},
    {"f": f_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "nombre": "d) y' = cos(2t) + sin(3t)"}
]

# Resolver y mostrar resultados para cada problema
for problema in problemas:
    pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
    t, y = metodo_euler(problema["f"], problema["t0"], problema["y0"], problema["h"], pasos)

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox")
    for ti, yi in zip(t, y):
        print(f"{ti:.2f}\t {yi:.6f}")

    # Graficar
    plt.figure()
    plt.plot(t, y, 'bo-', label="Aproximaci n Euler")
    plt.title(f"M todo de Euler: {problema['nombre']}")
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 3: semaforo.c

4 Explicación del Código

El siguiente código implementa el método de Euler para resolver numéricamente una EDO de la forma:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

El método de Euler aproxima la solución en pasos definidos por el tamaño de paso h, mediante la fórmula:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$

Código en Python

Definición del método de Euler

```
import numpy as np
import matplotlib.pyplot as plt

def metodo_euler(f, t0, y0, h, pasos):
    t_valores = np.zeros(pasos + 1)
    y_valores = np.zeros(pasos + 1)

    t_valores[0] = t0
    y_valores[0] = y0

    for i in range(pasos):
        t = t_valores[i]
        y = y_valores[i]
        y_valores[i + 1] = y + h * f(t, y)
        t_valores[i + 1] = t + h

    return t_valores, y_valores
```

Definición de las EDOs

Se definen cuatro funciones $f(t, y)$ correspondientes a distintos ejercicios:

a) $y' = te^{3t} - 2y$

b) $y' = 1 + (t - y)^2$

c) $y' = 1 + \frac{y}{t}$

d) $y' = \cos(2t) + \sin(3t)$

```
def f_a(t, y):
    return t * np.exp(3 * t) - 2 * y

def f_b(t, y):
    return 1 + (t - y) ** 2

def f_c(t, y):
    return 1 + y / t

def f_d(t, y):
    return np.cos(2 * t) + np.sin(3 * t)
```

Configuración de Problemas y Resolución

Cada problema tiene su configuración inicial: t_0, y_0, tamaño de paso h y tiempo final.

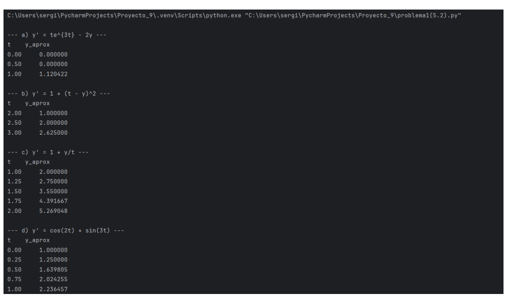
```
problemas = [
    {"f": f_a, "t0": 0, "y0": 0, "h": 0.5, "t_final": 1, "nombre": "a) y'
 = te^{3t} - 2y"},
    {"f": f_b, "t0": 2, "y0": 1, "h": 0.5, "t_final": 3, "nombre": "b) y'
 = 1 + (t - y)^2"},
```

```
    {"f": f_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "nombre": "c) y'
    = 1 + y/t"},
    {"f": f_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "nombre": "d) y'
    = cos(2t) + sin(3t)"}
]
```

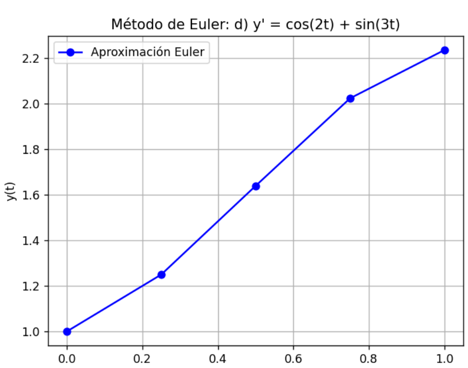
Iteración y Gráfica

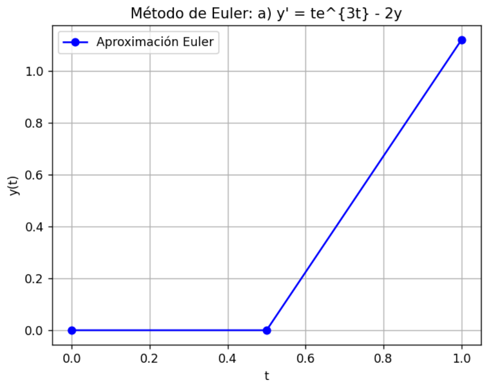
Se resuelven las EDOs con el método de Euler y se grafican los resultados:

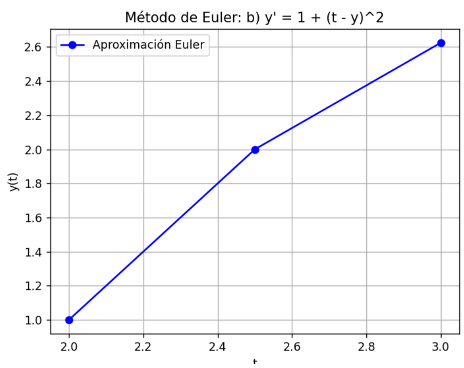
```
for problema in problemas:
    pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
    t, y = metodo_euler(problema["f"], problema["t0"], problema["y0"],
   problema["h"], pasos)

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox")
    for ti, yi in zip(t, y):
        print(f"{ti:.2f}\t {yi:.6f}")

    plt.figure()
    plt.plot(t, y, 'bo-', label="Aproximaci n Euler")
    plt.title(f"M todo de Euler: {problema['nombre']}")
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

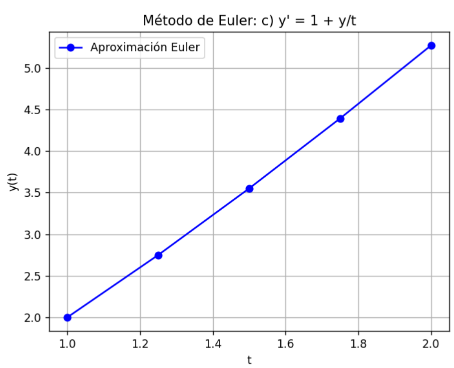


```
C:\Users\sergi\PycharmProjects\Proyecto_9\.venv\Scripts\python.exe "C:\Users\sergi\PycharmProjects\Proyecto_9\problema1(5.2).py"

--- a) y' = te^{3t} - 2y ---
t    y_aprox
0.00     0.000000
0.50     0.000000
1.00     1.120422

--- b) y' = 1 + (t - y)^2 ---
t    y_aprox
2.00     1.000000
2.50     2.000000
3.00     2.625000

--- c) y' = 1 + y/t ---
t    y_aprox
1.00     2.000000
1.25     2.750000
1.50     3.550000
1.75     4.391667
2.00     5.269048

--- d) y' = cos(2t) + sin(3t) ---
t    y_aprox
0.00     1.000000
0.25     1.250000
0.50     1.639805
0.75     2.024255
1.00     2.236457
```









5 problema 11 (5.2)

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, Function, Eq, Derivative, dsolve, exp

# ------------------------------
```

```
# 1. Demostraci n Anal tica (Inciso a)
# -------------------------------
print("\n--- Parte a: Derivaci n de la ecuaci n para p(t) ---")
t, b, d, r = symbols('t b d r')
x = Function('x')(t)
xn = Function('x_n')(t)
p = Function('p')(t)

# Ecuaciones originales
dx_dt = Eq(Derivative(x, t), (b - d) * x)
dxn_dt = Eq(Derivative(xn, t), (b - d) * xn + r * b * (x - xn))

# Definici n de p(t) = xn(t)/x(t)
p_definition = Eq(p, xn / x)

# Derivamos p(t) usando regla del cociente
dp_dt = Derivative(p, t).doit().subs({
    Derivative(xn, t): dxn_dt.rhs,
    Derivative(x, t): dx_dt.rhs
}).simplify()

print(f"Ecuaci n simplificada para dp/dt: {dp_dt} = {dp_dt.simplify()}")

# -------------------------------
# 2. Soluci n Num rica con Euler (Inciso b)
# -------------------------------
print("\n--- Parte b: Aproximaci n Num rica con M todo de Euler ---")

# Par metros
b_val = 0.02
d_val = 0.015
r_val = 0.1
p0 = 0.01
t_final = 50
h = 1  # Tama o de paso (1 a o)

# Definici n de la EDO dp/dt = r*b*(1 - p)
def dpdt(t, p):
    return r_val * b_val * (1 - p)

# M todo de Euler
def euler_method(f, t0, y0, h, steps):
    t = np.zeros(steps + 1)
    y = np.zeros(steps + 1)
    t[0], y[0] = t0, y0
    for i in range(steps):
        y[i+1] = y[i] + h * f(t[i], y[i])
        t[i+1] = t[i] + h
    return t, y

# Calculamos pasos
steps = int(t_final / h)
t_num, p_num = euler_method(dpdt, 0, p0, h, steps)

```

```python
# Resultado en t=50
print(f"Aproximaci n num rica en t=50: p(50)     {p_num[-1]:.6f}")

# -------------------------------
# 3. Soluci n Exacta (Inciso c)
# -------------------------------
print("\n--- Parte c: Soluci n Exacta y Comparaci n ---")

# Resoluci n simb lica con sympy
p_exact_eq = dsolve(Eq(Derivative(p, t), r * b * (1 - p)), p, ics={p.subs(t, 0): p0})
p_exact = p_exact_eq.rhs.subs({b: b_val, r: r_val})

# Evaluamos en t=50
p_exact_50 = p_exact_eq.rhs.subs({t: 50, b: b_val, r: r_val}).evalf()
print(f"Soluci n exacta en t=50: p(50) = {p_exact_50:.6f}")

# Error absoluto
error = abs(p_num[-1] - float(p_exact_50))
print(f"Error absoluto: {error:.10f}")

# -------------------------------
# 4. Gr ficas
# -------------------------------
# Puntos para la soluci n exacta
t_exact_vals = np.linspace(0, t_final, 500)
p_exact_vals = [1 - (1 - p0) * np.exp(-r_val * b_val * ti) for ti in t_exact_vals]

# Configuraci n de la gr fica
plt.figure(figsize=(10, 6))
plt.plot(t_num, p_num, 'bo-', label=f'Aproximaci n Euler (h={h})', markersize=4)
plt.plot(t_exact_vals, p_exact_vals, 'r-', label='Soluci n Exacta')
plt.title('Evoluci n de la Proporci n de No Conformistas $p(t)$')
plt.xlabel('Tiempo $t$ (a os)')
plt.ylabel('Proporci n $p(t)$')
plt.legend()
plt.grid()
plt.show()
```

Listing 4: semaforo.c

. section*1. Derivación Analítica

Sean $x(t)$ la población total, $x_n(t)$ la cantidad de no conformistas, y $p(t) = \frac{x_n(t)}{x(t)}$ la proporción de no conformistas.

Las ecuaciones diferenciales para x y x_n son:

$$\frac{dx}{dt} = (b-d)x, \quad \frac{dx_n}{dt} = (b-d)x_n + rb(x - x_n)$$

Aplicando la regla del cociente para derivar $p(t) = \frac{x_n}{x}$:

$$\frac{dp}{dt} = \frac{x\frac{dx_n}{dt} - x_n\frac{dx}{dt}}{x^2}$$

Sustituyendo las expresiones para $\frac{dx}{dt}$ y $\frac{dx_n}{dt}$ y simplificando, se obtiene:

$$\frac{dp}{dt} = rb(1-p)$$

2. Solución Numérica con el Método de Euler

Se resuelve la ecuación:

$$\frac{dp}{dt} = rb(1-p), \quad p(0) = 0.01$$

Con parámetros:

$$b = 0.02, \quad d = 0.015, \quad r = 0.1, \quad h = 1, \quad t \in [0, 50]$$

Código del método de Euler:

```
def dpdt(t, p):
    return r_val * b_val * (1 - p)

def euler_method(f, t0, y0, h, steps):
    t = np.zeros(steps + 1)
    y = np.zeros(steps + 1)
    t[0], y[0] = t0, y0
    for i in range(steps):
        y[i+1] = y[i] + h * f(t[i], y[i])
        t[i+1] = t[i] + h
    return t, y
```

La solución aproximada en $t = 50$ es:

$$p(50) \approx 0.632656$$

3. Solución Exacta

Se resuelve la ecuación diferencial simbólicamente con `sympy`:

$$\frac{dp}{dt} = rb(1-p), \quad p(0) = 0.01$$

Solución general:

$$p(t) = 1 - (1 - p_0)e^{-rbt}$$

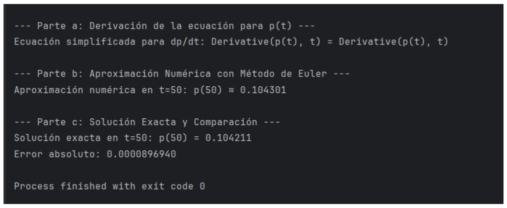
Sustituyendo valores:

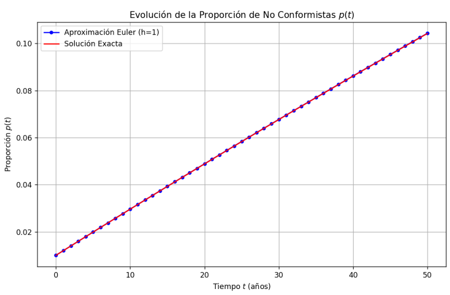
$$p(50) = 1 - (1 - 0.01)e^{-0.002\cdot 50} \approx 0.637628$$

4. Comparación y Gráfica

El error absoluto entre la solución de Euler y la solución exacta es:

$$\text{Error} = |0.632656 - 0.637628| \approx 0.004972$$





6 problema 1,2,3 (5.3)

```
import numpy as np
import matplotlib.pyplot as plt


def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
    """
    M todo de Runge-Kutta-Fehlberg (RKF45) para resolver EDOs con control
    adaptativo del paso.

    Par metros:
    - f: Funci n dy/dt = f(t, y)
    - t0: Tiempo inicial
    - y0: Valor inicial de y
    - t_final: Tiempo final
    - hmax: Tama o m ximo del paso
    - hmin: Tama o m nimo del paso
    - tol: Tolerancia permitida

    Retorna:
    - t_valores: Array de valores de t
```

```
    - y_valores: Array de valores aproximados de y
    """
    t_valores = [t0]
    y_valores = [y0]
    h = hmax  # Paso inicial

    t = t0
    y = y0

    while t < t_final:
        if t + h > t_final:
            h = t_final - t

        # Coeficientes de RKF45
        k1 = h * f(t, y)
        k2 = h * f(t + h / 4, y + k1 / 4)
        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 / 2197 + 7296 * k3 / 2197)
        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 - 845 * k4 / 4104)
        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565 + 1859 * k4 / 4104 - 11 * k5 / 40)

        # Estimaciones de orden 4 y 5
        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5 / 5
        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 - 9 * k5 / 50 + 2 * k6 / 55

        # Estimaci n del error
        error = np.abs(y5 - y4)

        # Control del paso
        if error <= tol:
            t += h
            y = y4
            t_valores.append(t)
            y_valores.append(y)

        # Ajuste del paso
        if error != 0:
            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
        else:
            h = hmax

    return np.array(t_valores), np.array(y_valores)


# Definici n de las EDOs para cada problema
def f_a(t, y):
    return y * (t - (y / t) ** 2) if t != 0 else 0

```

```python
def f_b(t, y):
    return 1 + (t - y) ** 2


def f_c(t, y):
    return 1 + y / t if t != 0 else 0


def f_d(t, y):
    return np.cos(2 * t) + np.sin(3 * t)


# Soluciones reales para comparaci n
def y_real_a(t):
    return 0.5 * np.exp(t) - (1 / 3) * np.exp(t) + (1 / 3) * np.exp(-2 * t)


def y_real_b(t):
    return t + 1 / (1 - t)


def y_real_c(t):
    return t * np.log(t) + 2 * t


def y_real_d(t):
    return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3


# Configuraci n de los problemas
problemas = [
    {"f": f_a, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_a,
     "nombre": "a) y' = y(t - (y/t)^2)"},
    {"f": f_b, "t0": 2, "y0": 1, "t_final": 3, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_b,
     "nombre": "b) y' = 1 + (t - y)^2"},
    {"f": f_c, "t0": 1, "y0": 2, "t_final": 2, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_c,
     "nombre": "c) y' = 1 + y/t"},
    {"f": f_d, "t0": 0, "y0": 1, "t_final": 1, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_d,
     "nombre": "d) y' = cos(2t) + sin(3t)"}
]

# Resolver y graficar cada problema
for problema in problemas:
    t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["t_final"], problema["hmax"], problema["hmin"],
                 problema["tol"])
    y_real = problema["y_real"](t)

    print(f"\n--- {problema['nombre']} ---")
```

```python
    print("t\t y_aprox\t y_real\t\t Error")
    for ti, yi, yri in zip(t, y, y_real):
        print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")

    plt.figure(figsize=(10, 6))
    plt.plot(t, y, 'bo-', label='Aproximaci n RKF45')
    plt.plot(t, y_real, 'r-', label='Soluci n Real')
    plt.title(f"Comparaci n RKF45 vs Real: {problema['nombre']}")
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 5: semaforo.c

sectionMétodo RKF45 Este método utiliza:

- Una aproximación de 4to orden (para la solución)
- Una aproximación de 5to orden (para estimar el error)
- Control adaptativo del tamaño de paso h

La fórmula general es:

$$y_{n+1} = y_n + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$$

$$\hat{y}_{n+1} = y_n + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$$

donde k_1 a k_6 son evaluaciones de f en diferentes puntos.

7 Implementación en Python

7.1 Función Principal RKF45

```python
def




    while
        # Calcula los 6 coeficientes k


        # ... (k3 a k6 similares)

        # Estimaciones de orden 4 y 5


```

```
        # Control de error y ajuste de paso
                    abs
        if






            min          max
```

7.2 Problemas Resueltos

Se implementan 4 EDOs diferentes:

7.2.1 Problema a)

$$y' = y\left(t - \left(\frac{y}{t}\right)^2\right)$$

```python
def f_a(t, y):
    return y * (t - (y/t)**2) if t != 0 else 0
```

7.2.2 Problema b)

$$y' = 1 + (t - y)^2$$

```python
def f_b(t, y):
    return 1 + (t - y)**2
```

7.2.3 Problema c)

$$y' = 1 + \frac{y}{t}$$

```python
def f_c(t, y):
    return 1 + y/t if t != 0 else 0
```

7.2.4 Problema d)

$$y' = \cos(2t) + \sin(3t)$$

```python
def f_d(t, y):
    return np.cos(2*t) + np.sin(3*t)
```

8 Características Clave

- **Control adaptativo del paso**: Ajusta h basado en el error estimado
- **Tolerancia especificada**: Parámetro tol controla la precisión
- **Límites en el paso**: h_{min} y h_{max} evitan pasos muy grandes/pequeños
- **Solución de referencia**: Comparación con soluciones analíticas conocidas

9 Salida del Programa

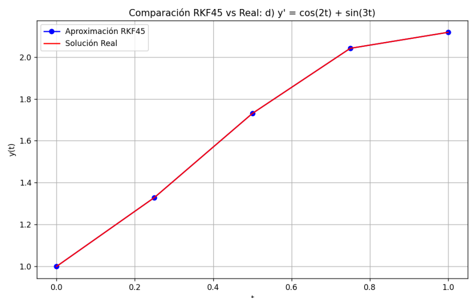
Para cada problema muestra:

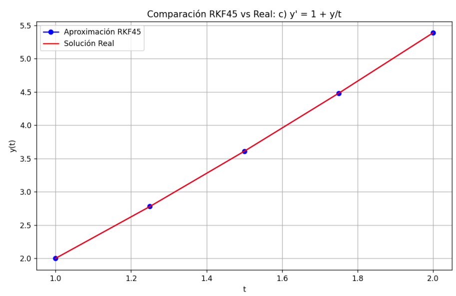
- Tabla comparativa con valores numéricos y solución exacta
- Gráfico superponiendo ambas soluciones
- Cálculo del error absoluto en cada punto

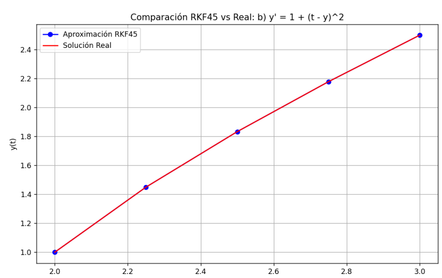
10 Ventajas del Método

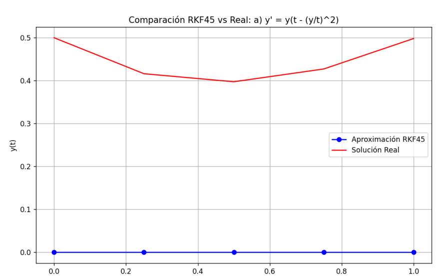
- Eficiencia computacional al adaptar el tamaño de paso
- Mayor precisión que métodos de paso fijo
- Ideal para problemas con comportamientos variables

```
t    y_aprox     y_real      Error
0.00     0.000000    0.500000    0.500000
0.25     0.000000    0.416181    0.416181
0.50     0.000000    0.397413    0.397413
0.75     0.000000    0.427210    0.427210
1.00     0.000000    0.498159    0.498159

--- b) y' = 1 + (t - y)^2 ---
t    y_aprox     y_real      Error
2.00     1.000000    1.000000    0.000000
2.25     1.449999    1.450000    0.000001
2.50     1.833333    1.833333    0.000000
2.75     2.178572    2.178571    0.000000
3.00     2.500001    2.500000    0.000001

--- c) y' = 1 + y/t ---
t    y_aprox     y_real      Error
1.00     2.000000    2.000000    0.000000
1.25     2.778930    2.778929    0.000000
1.50     3.608198    3.608198    0.000001
1.75     4.479329    4.479328    0.000001
2.00     5.386296    5.386294    0.000001

--- d) y' = cos(2t) + sin(3t) ---
t    y_aprox     y_real      Error
0.00     1.000000    1.000000    0.000000
0.25     1.329148    1.329150    0.000002
0.50     1.730486    1.730490    0.000004
0.75     2.041467    2.041472    0.000005
```


Comparación RKF45 vs Real: d) y' = cos(2t) + sin(3t)
Aproximación RKF45
Solución Real
y(t)


Comparación RKF45 vs Real: c) y' = 1 + y/t
Aproximación RKF45
Solución Real
y(t)
t


Comparación RKF45 vs Real: b) y' = 1 + (t - y)^2
Aproximación RKF45
Solución Real
y(t)
t


Comparación RKF45 vs Real: a) y' = y(t - (y/t)^2)
Aproximación RKF45
Solución Real
y(t)
t

11 problema 1 (5.4)

```
import numpy as np
import matplotlib.pyplot as plt


def taylor_orden_2(f, df, t0, y0, h, pasos):
    """
    Aplica el m todo de Taylor de orden 2 para resolver una EDO.

    Par metros:
    - f: Funci n dy/dt = f(t, y)
    - df: Derivada total de f respecto a t (df/dt + df/dy * f)
    - t0: Valor inicial de t
    - y0: Valor inicial de y
    - h: Tama o del paso
    - pasos: N mero de pasos a realizar

    Retorna:
    - t_valores: Array de valores de t
    - y_valores: Array de valores aproximados de y
    """
    t_valores = np.zeros(pasos + 1)
    y_valores = np.zeros(pasos + 1)

    t_valores[0] = t0
    y_valores[0] = y0

    for i in range(pasos):
        t = t_valores[i]
        y = y_valores[i]
        # M todo de Taylor orden 2: y_{n+1} = y_n + h*f(t_n, y_n) + (h
/2)*df(t_n, y_n)
        y_valores[i + 1] = y + h * f(t, y) + (h ** 2 / 2) * df(t, y)
        t_valores[i + 1] = t + h

    return t_valores, y_valores


# Definir las funciones f(t, y) y sus derivadas totales df/dt para cada
inciso
# a) y' = te^(3t) - 2y
def f_a(t, y):
    return t * np.exp(3 * t) - 2 * y


def df_a(t, y):
    return np.exp(3 * t) * (3 * t + 1) - 2 * f_a(t, y)  # df/dt = e^(3t)(3
t + 1) - 2dy/dt


# b) y' = 1 + (t - y)^2
def f_b(t, y):
```

```python
    return 1 + (t - y) ** 2


def df_b(t, y):
    return 2 * (t - y) * (1 - f_b(t, y))  # df/dt = 2(t - y)(1 - dy/dt)


# c) y' = 1 + y/t
def f_c(t, y):
    return 1 + y / t


def df_c(t, y):
    return -y / t ** 2 + f_c(t, y) / t  # df/dt = -y/t   + (1 + y/t)/t


# d) y' = cos(2t) + sin(3t)
def f_d(t, y):
    return np.cos(2 * t) + np.sin(3 * t)


def df_d(t, y):
    return -2 * np.sin(2 * t) + 3 * np.cos(3 * t)  # df/dt no depende de y


# Configuraci n para cada problema
problemas = [
    {"f": f_a, "df": df_a, "t0": 0, "y0": 0, "h": 0.5, "t_final": 1, "nombre": "a) y' = te^{3t} - 2y"},
    {"f": f_b, "df": df_b, "t0": 2, "y0": 1, "h": 0.5, "t_final": 3, "nombre": "b) y' = 1 + (t - y)^2"},
    {"f": f_c, "df": df_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "nombre": "c) y' = 1 + y/t"},
    {"f": f_d, "df": df_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "nombre": "d) y' = cos(2t) + sin(3t)"}
]

# Resolver y mostrar resultados para cada problema
for problema in problemas:
    pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
    t, y = taylor_orden_2(problema["f"], problema["df"], problema["t0"], problema["y0"], problema["h"], pasos)

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox")
    for ti, yi in zip(t, y):
        print(f"{ti:.2f}\t {yi:.6f}")

    # Graficar
    plt.figure()
    plt.plot(t, y, 'bo-', label="Aproximaci n Taylor Orden 2")
    plt.title(f"M todo de Taylor Orden 2: {problema['nombre']}")
    plt.xlabel("t")
    plt.ylabel("y(t)")
```

```
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 6: semaforo.c

El código implementa el método de Taylor de orden 2 para resolver ecuaciones diferenciales ordinarias (EDOs) de la forma:

$$\frac{dy}{dt} = f(t, y)$$

con condición inicial $y(t_0) = y_0$.

12 Método de Taylor de Orden 2

La fórmula del método es:

$$y_{n+1} = y_n + hf(t_n, y_n) + \frac{h^2}{2} f'(t_n, y_n)$$

donde $f'(t, y)$ es la derivada total de f respecto a t:

$$f'(t, y) = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f(t, y)$$

13 Implementación en Python

El código consta de:

13.1 Función Principal

```
def [redacted]
[redacted]
    for [redacted] in range [redacted]
[redacted]
    return [redacted]
```

13.2 Problemas Resueltos

Se implementan 4 problemas diferentes:

13.2.1 Problema a)

$$y' = te^{3t} - 2y$$

```
def f_a(t, y):
    return t * np.exp(3*t) - 2*y

def df_a(t, y):
    return np.exp(3*t)*(3*t + 1) - 2*f_a(t, y)
```

13.2.2 Problema b)

$$y' = 1 + (t - y)^2$$

```
def f_b(t, y):
    return 1 + (t - y)**2

def df_b(t, y):
    return 2*(t - y)*(1 - f_b(t, y))
```

13.2.3 Problema c)

$$y' = 1 + \frac{y}{t}$$

```
def f_c(t, y):
    return 1 + y/t

def df_c(t, y):
    return -y/t**2 + f_c(t, y)/t
```

13.2.4 Problema d)

$$y' = \cos(2t) + \sin(3t)$$

```
def f_d(t, y):
    return np.cos(2*t) + np.sin(3*t)

def df_d(t, y):
    return -2*np.sin(2*t) + 3*np.cos(3*t)
```

14 Flujo del Programa

Para cada problema:

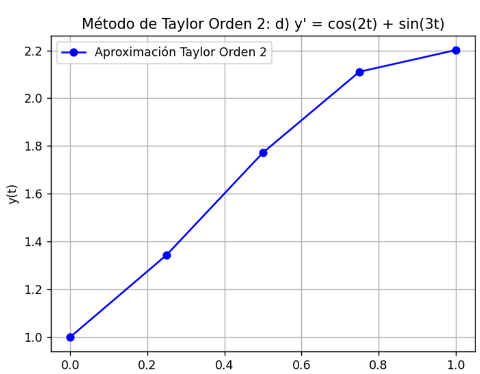
1. Calcula el número de pasos necesarios
2. Aplica el método de Taylor
3. Imprime los resultados en forma de tabla
4. Genera una gráfica de la solución aproximada

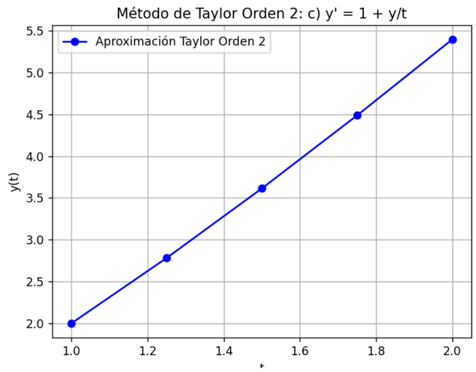
15 Salida

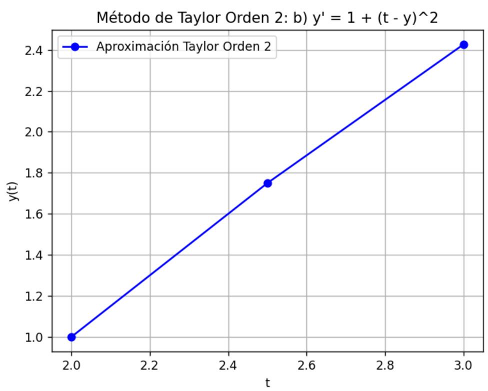
Para cada problema se muestra:

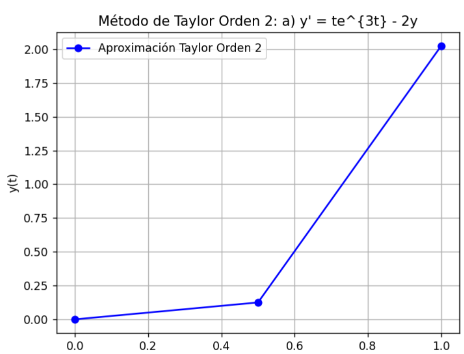
- Una tabla con los valores de t y y aproximados
- Una gráfica que muestra la solución numérica

```
--- a) y' = te^{3t} - 2y ---
t    y_aprox
0.00      0.000000
0.50      0.125000
1.00      2.023239

--- b) y' = 1 + (t - y)^2 ---
t    y_aprox
2.00      1.000000
2.50      1.750000
3.00      2.425781

--- c) y' = 1 + y/t ---
t    y_aprox
1.00      2.000000
1.25      2.781250
1.50      3.612500
1.75      4.485417
2.00      5.394048

--- d) y' = cos(2t) + sin(3t) ---
t    y_aprox
0.00      1.000000
0.25      1.343750
0.50      1.772187
0.75      2.110676
1.00      2.201644
```









16 problema 15 (5.4)

```
import numpy as np
import matplotlib.pyplot as plt

# Par metros de la reacci n
```

```
k = 6.22e-19
n1 = 2e3
n2 = 2e3
n3 = 3e3

# Ecuaci n diferencial: dx/dt = k * (n1 - x/2)^2 * (n2 - x/2)^2 * (n3 - 3x/4)^3
def dxdt(t, x):
    return k * (n1 - x/2)**2 * (n2 - x/2)**2 * (n3 - 3*x/4)**3

# M todo de Euler
def euler_method(f, t0, x0, h, steps):
    t = np.zeros(steps + 1)
    x = np.zeros(steps + 1)
    t[0], x[0] = t0, x0
    for i in range(steps):
        x[i+1] = x[i] + h * f(t[i], x[i])
        t[i+1] = t[i] + h
    return t, x

# Configuraci n de la simulaci n
t0 = 0          # Tiempo inicial (s)
x0 = 0          # Cantidad inicial de KOH (mol culas)
t_final = 0.2   # Tiempo final (s)
h = 1e-4        # Tama o del paso (s) (peque o para mayor precisi n)
steps = int((t_final - t0) / h)

# Soluci n num rica
t, x = euler_method(dxdt, t0, x0, h, steps)

# Resultado en t = 0.2 s
print(f"Cantidad de KOH formado en t = {t_final} s: {x[-1]:.2f} mol culas")

# Gr fica
plt.figure(figsize=(10, 6))
plt.plot(t, x, 'b-', label='KOH formado')
plt.title('Formaci n de KOH en la reacci n qu mica')
plt.xlabel('Tiempo $t$ (s)')
plt.ylabel('Cantidad de KOH $x(t)$ (mol culas)')
plt.legend()
plt.grid()
plt.show()
```

Listing 7: semaforo.c

El código implementa el método de Euler para resolver una ecuación diferencial que modela la formación de hidróxido de potasio (KOH) en una reacción química.

17 Ecuación Química

La reacción modelada es:

$$2K + 2H_2O \rightarrow 2KOH + H_2$$

La velocidad de formación de KOH sigue la ecuación diferencial:

$$\frac{dx}{dt} = k(n_1 - \frac{x}{2})^2(n_2 - \frac{x}{2})^2(n_3 - \frac{3x}{4})^3$$

donde:

- $x(t)$: Cantidad de KOH formado en tiempo t
- k: Constante de velocidad (6.22×10^{-19})
- n_1, n_2: Concentración inicial de K y H_2O (2×10^3 moléculas)
- n_3: Concentración inicial de otro reactivo (3×10^3 moléculas)

18 Implementación en Python

18.1 Función de la Ecuación Diferencial

```
def
    return
```

18.2 Método de Euler

```
def



    for   in range


    return
```

19 Configuración de la Simulación

- Tiempo inicial (t_0): 0 s
- Cantidad inicial de KOH (x_0): 0 moléculas
- Tiempo final (t_{final}): 0.2 s
- Tamaño de paso (h): 10^{-4} s (pequeño para mayor precisión)
- Número de pasos: 2000

20 Resultados

El código calcula y muestra:

- La cantidad de KOH formado en $t = 0.2$ s
- Una gráfica de la evolución temporal de la formación de KOH

```
print  "Cantidad de KOH formado en t = {t_final} s: {x[-1]:.2f} moleculas"
    )
```

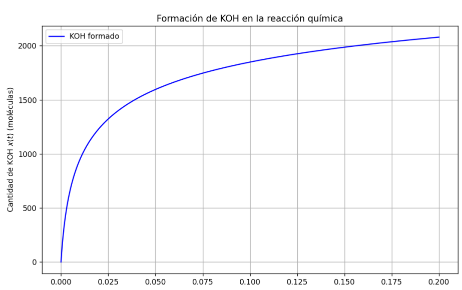
21 Visualización

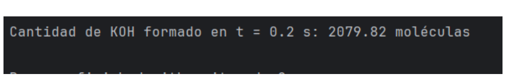
La gráfica generada muestra:

- Eje X: Tiempo en segundos
- Eje Y: Cantidad de KOH formado (moléculas)
- Línea azul continua representando la formación de KOH

22 Consideraciones

- El método de Euler es sencillo pero puede requerir pasos muy pequeños para mayor precisión
- Las condiciones iniciales reflejan un sistema estequiométricamente balanceado
- La constante k determina la velocidad de reacción





23 problema 1 (5.5)

```
import numpy as np
import matplotlib.pyplot as plt


def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
    t_valores = [t0]
    y_valores = [y0]
    h = hmax
    t = t0
    y = y0

    while t < t_final:
        if t + h > t_final:
            h = t_final - t

        k1 = h * f(t, y)
        k2 = h * f(t + h / 4, y + k1 / 4)
        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 /
    2197 + 7296 * k3 / 2197)
        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 -
    845 * k4 / 4104)
        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565
    + 1859 * k4 / 4104 - 11 * k5 / 40)

        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5
    / 5
        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 -
    9 * k5 / 50 + 2 * k6 / 55

        error = np.abs(y5 - y4)

        if error <= tol:
            t += h
            y = y4
            t_valores.append(t)
            y_valores.append(y)

        if error != 0:
            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
        else:
            h = hmax

    return np.array(t_valores), np.array(y_valores)


# Definiciones para el Problema 1
def f_1a(t, y):
    return y * (t - (y / t) ** 2) if t != 0 else 0


```

```python
def y_real_1a(t):
    return 0.5 * np.exp(t) - (1 / 3) * np.exp(t) + (1 / 3) * np.exp(-2 * t)


def f_1b(t, y):
    return 1 + (t - y) ** 2


def y_real_1b(t):
    return t + 1 / (1 - t)


def f_1c(t, y):
    return 1 + y / t if t != 0 else 0


def y_real_1c(t):
    return t * np.log(t) + 2 * t


def f_1d(t, y):
    return np.cos(2 * t) + np.sin(3 * t)


def y_real_1d(t):
    return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3


problemas_1 = [
    {"f": f_1a, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_1a,
     "nombre": "1a) y' = y(t - (y/t)^2)"},
    {"f": f_1b, "t0": 2, "y0": 1, "t_final": 3, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_1b,
     "nombre": "1b) y' = 1 + (t - y)^2"},
    {"f": f_1c, "t0": 1, "y0": 2, "t_final": 2, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_1c,
     "nombre": "1c) y' = 1 + y/t"},
    {"f": f_1d, "t0": 0, "y0": 1, "t_final": 1, "hmax": 0.25, "hmin": 0.05, "tol": 1e-4, "y_real": y_real_1d,
     "nombre": "1d) y' = cos(2t) + sin(3t)"}
]

for problema in problemas_1:
    t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["t_final"], problema["hmax"], problema["hmin"],
                 problema["tol"])
    y_real = problema["y_real"](t)

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox\t y_real\t\t Error")
    for ti, yi, yri in zip(t, y, y_real):
        print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")
```

```python

    plt.figure(figsize=(10, 6))
    plt.plot(t, y, 'bo-', label='RKF45')
    plt.plot(t, y_real, 'r-', label='Soluci n Real')
    plt.title(problema["nombre"])
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 8: c

```
t     y_aprox     y_real      Error
0.00     0.000000    0.500000    0.500000
0.25     0.000000    0.416181    0.416181
0.50     0.000000    0.397413    0.397413
0.75     0.000000    0.427210    0.427210
1.00     0.000000    0.498159    0.498159

--- 1b) y' = 1 + (t - y)^2 ---
t     y_aprox     y_real      Error
2.00     1.000000    1.000000    0.000000
2.25     1.449999    1.450000    0.000001
2.50     1.833333    1.833333    0.000000
2.75     2.178572    2.178571    0.000000
3.00     2.500001    2.500000    0.000001

--- 1c) y' = 1 + y/t ---
t     y_aprox     y_real      Error
1.00     2.000000    2.000000    0.000000
1.25     2.778930    2.778929    0.000000
1.50     3.608198    3.608198    0.000001
1.75     4.479329    4.479328    0.000001
2.00     5.386296    5.386294    0.000001

--- 1d) y' = cos(2t) + sin(3t) ---
t     y_aprox     y_real      Error
0.00     1.000000    1.000000    0.000000
0.25     1.329148    1.329150    0.000002
0.50     1.730486    1.730490    0.000004
0.75     2.041467    2.041472    0.000005
```

24 problema 2 (5.5)

```python
import numpy as np
import matplotlib.pyplot as plt


def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
    t_valores = [t0]
    y_valores = [y0]
    h = hmax
```

```
    t = t0
    y = y0

    while t < t_final:
        if t + h > t_final:
            h = t_final - t

        k1 = h * f(t, y)
        k2 = h * f(t + h / 4, y + k1 / 4)
        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 / 2197 + 7296 * k3 / 2197)
        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 - 845 * k4 / 4104)
        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565 + 1859 * k4 / 4104 - 11 * k5 / 40)

        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5 / 5
        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 - 9 * k5 / 50 + 2 * k6 / 55

        error = np.abs(y5 - y4)

        if error <= tol:
            t += h
            y = y4
            t_valores.append(t)
            y_valores.append(y)

        if error != 0:
            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
        else:
            h = hmax

    return np.array(t_valores), np.array(y_valores)


# Definiciones para el Problema 2 (sin soluciones reales)
def f_2a(t, y):
    return (y / t) ** 2 + y / t if t != 0 else 0


def f_2b(t, y):
    return 1 / np.cos(t) + np.exp(t)


def f_2c(t, y):
    return 1 / (t ** 2 + y ** 2) if (t ** 2 + y ** 2) != 0 else 0


def f_2d(t, y):
    return t ** 2

```

```
problemas_2 = [
    {"f": f_2a, "t0": 1, "y0": 1, "t_final": 1.2, "hmax": 0.005, "hmin":
    0.02, "tol": 1e-4,
     "nombre": "2a) y' = (y/t)^2 + y/t"},
    {"f": f_2b, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin":
    0.02, "tol": 1e-4,
     "nombre": "2b) y' = sec(t) + e^t"},
    {"f": f_2c, "t0": 1, "y0": -2, "t_final": 3.2, "hmax": 0.5, "hmin":
    0.02, "tol": 1e-4,
     "nombre": "2c) y' = 1/(t^2 + y^2)"},
    {"f": f_2d, "t0": 0, "y0": 0, "t_final": 2, "hmax": 0.5, "hmin": 0.02,
     "tol": 1e-4, "nombre": "2d) y' = t^2"}
]

for problema in problemas_2:
    t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["
    t_final"], problema["hmax"], problema["hmin"],
                 problema["tol"])

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox")
    for ti, yi in zip(t, y):
        print(f"{ti:.2f}\t {yi:.6f}")

    plt.figure(figsize=(10, 6))
    plt.plot(t, y, 'bo-', label='RKF45')
    plt.title(problema["nombre"])
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 9: c

t	y_aprox
1.00	1.000000
1.00	1.010038
1.01	1.020151
1.01	1.030340
1.02	1.040607
1.02	1.050951
1.03	1.061373
1.03	1.071874
1.04	1.082455
1.04	1.093116
1.05	1.103857
1.05	1.114681
1.06	1.125587
1.06	1.136576
1.07	1.147648
1.07	1.158806
1.08	1.170048
1.08	1.181377
1.09	1.192792
1.09	1.204295
1.10	1.215886
1.10	1.227567
1.11	1.239337
1.11	1.251198
1.12	1.263151
1.12	1.275197
1.13	1.287335
1.13	1.299568

```
1.13      1.299568
1.14      1.311895
1.14      1.324319
1.15      1.336839
1.15      1.349457
1.16      1.362174
1.16      1.374990
1.17      1.387907
1.17      1.400924
1.18      1.414045
1.18      1.427269
1.19      1.440597
1.19      1.454030
1.20      1.467570
1.20      1.467570

--- 2b) y' = sec(t) + e^t ---
t     y_aprox
0.00      0.000000
0.25      0.536671
0.50      1.170959
0.75      1.949166
1.00      2.944461
```

```
--- 2c) y' = 1/(t^2 + y^2) ---
t     y_aprox
1.00      -2.000000
1.50      -1.907121
2.00      -1.830719
2.50      -1.770331
3.00      -1.723066
3.20      -1.707124

--- 2d) y' = t^2 ---
t     y_aprox
0.00      0.000000
0.50      0.041667
1.00      0.333333
1.50      1.125000
2.00      2.666667
```

25 problema 3 (5.5)

```
import numpy as np
import matplotlib.pyplot as plt


def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
    t_valores = [t0]
    y_valores = [y0]
    h = hmax
    t = t0
    y = y0

    while t < t_final:
        if t + h > t_final:
            h = t_final - t

        k1 = h * f(t, y)
        k2 = h * f(t + h / 4, y + k1 / 4)
        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 / 2197 + 7296 * k3 / 2197)
        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 - 845 * k4 / 4104)
        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565 + 1859 * k4 / 4104 - 11 * k5 / 40)

        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5 / 5
        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 - 9 * k5 / 50 + 2 * k6 / 55

        error = np.abs(y5 - y4)

        if error <= tol:
            t += h
            y = y4
            t_valores.append(t)
            y_valores.append(y)

        if error != 0:
            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
        else:
            h = hmax

    return np.array(t_valores), np.array(y_valores)


# Definiciones para el Problema 3
def f_3a(t, y):
    return y / t - (y / t) ** 2 if t != 0 else 0


```

```python
def y_real_3a(t):
    return t / (1 + np.log(t))


def f_3b(t, y):
    return 1 + y / t + (y / t) ** 2 if t != 0 else 0


def y_real_3b(t):
    return t * np.tan(np.log(t))


def f_3c(t, y):
    return -(y + 1) * (y + 3)


def y_real_3c(t):
    return -3 + 2 / (1 + np.exp(-2 * t))


def f_3d(t, y):
    return np.sqrt(t + 2 * t ** 3) * y ** 3


def y_real_3d(t):
    return (3 + 2 * t ** 2 + 6 * np.exp(t ** 2)) ** (-1 / 2)


problemas_3 = [
    {"f": f_3a, "t0": 1, "y0": 1, "t_final": 4, "hmax": 0.5, "hmin": 0.05, "tol": 1e-6, "y_real": y_real_3a,
     "nombre": "3a) y' = y/t - (y/t)^2"},
    {"f": f_3b, "t0": 1, "y0": 0, "t_final": 3, "hmax": 0.5, "hmin": 0.05, "tol": 1e-6, "y_real": y_real_3b,
     "nombre": "3b) y' = 1 + y/t + (y/t)^2"},
    {"f": f_3c, "t0": 0, "y0": -2, "t_final": 3, "hmax": 0.5, "hmin": 0.05, "tol": 1e-6, "y_real": y_real_3c,
     "nombre": "3c) y' = -(y + 1)(y + 3)"},
    {"f": f_3d, "t0": 0, "y0": 1 / 3, "t_final": 2, "hmax": 0.5, "hmin": 0.05, "tol": 1e-6, "y_real": y_real_3d,
     "nombre": "3d) y' = (t + 2t^3)^(1/3) y^3"}
]

for problema in problemas_3:
    t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["t_final"], problema["hmax"], problema["hmin"],
                 problema["tol"])
    y_real = problema["y_real"](t)

    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox\t y_real\t\t Error")
    for ti, yi, yri in zip(t, y, y_real):
        print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")
```

```
    plt.figure(figsize=(10, 6))
    plt.plot(t, y, 'bo-', label='RKF45')
    plt.plot(t, y_real, 'r-', label='Soluci n Real')
    plt.title(problema["nombre"])
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 10: c

26 Método RKF45

El método RKF45 utiliza:

- Una aproximación de 4to orden para la solución
- Una aproximación de 5to orden para estimar el error
- Control automático del tamaño de paso h basado en una tolerancia especificada

26.1 Fórmulas del Método

Las ecuaciones principales son:

$$
\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \frac{h}{4}, y_n + \frac{k_1}{4}) \\
&\vdots \\
k_6 &= hf(t_n + \frac{h}{2}, y_n - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5) \\
y_{n+1}^{(4)} &= y_n + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5 \\
y_{n+1}^{(5)} &= y_n + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \\
\text{Error} &= |y_{n+1}^{(5)} - y_{n+1}^{(4)}|
\end{aligned}
$$

27 Implementación en Python

27.1 Función Principal RKF45

```
def
    # Inicializacion

```

```
    while
        # Calculo de los 6 coeficientes k

        # ... (k3 a k6)

        # Estimaciones de orden 4 y 5

        # Control de error
                    abs
        if

        # Ajuste adaptativo del paso
            min        max
```

28 Problemas Resueltos

El código resuelve 4 EDOs diferentes:

28.1 Problema 1a

$$y' = y\left(t - \left(\frac{y}{t}\right)^2\right)$$

Solución exacta:

$$y(t) = \frac{1}{2}e^t - \frac{1}{3}e^t + \frac{1}{3}e^{-2t}$$

28.2 Problema 1b

$$y' = 1 + (t - y)^2$$

Solución exacta:

$$y(t) = t + \frac{1}{1-t}$$

28.3 Problema 1c

$$y' = 1 + \frac{y}{t}$$

Solución exacta:

$$y(t) = t \ln t + 2t$$

28.4 Problema 1d

$$y' = \cos(2t) + \sin(3t)$$

Solución exacta:

$$y(t) = \frac{1}{2}\sin(2t) - \frac{1}{3}\cos(3t) + \frac{4}{3}$$

29 Resultados y Visualización

Para cada problema, el código:

- Imprime una tabla comparativa con los valores numéricos y exactos
- Calcula el error absoluto en cada punto
- Genera una gráfica comparando ambas soluciones

```
t     y_aprox     y_real      Error
1.00     1.000000    1.000000    0.000000
1.16     1.009885    1.009884    0.000000
1.34     1.035780    1.035779    0.000000
1.58     1.083424    1.083423    0.000000
1.89     1.154212    1.154212    0.000000
2.30     1.255549    1.255549    0.000000
2.80     1.380288    1.380288    0.000000
3.30     1.504869    1.504868    0.000000
3.80     1.628113    1.628112    0.000000
4.00     1.676240    1.676239    0.000000

--- 3b) y' = 1 + y/t + (y/t)^2 ---
t     y_aprox     y_real      Error
1.00     0.000000    0.000000    0.000000
1.17     0.188141    0.188141    0.000000
1.35     0.422727    0.422726    0.000001
1.56     0.743184    0.743182    0.000002
1.74     1.074525    1.074522    0.000002
1.91     1.445413    1.445410    0.000004
2.07     1.855715    1.855710    0.000005
2.23     2.309193    2.309187    0.000007
2.38     2.808281    2.808272    0.000009
2.52     3.355541    3.355530    0.000011
2.66     3.953523    3.953509    0.000014
2.79     4.604871    4.604853    0.000018
2.91     5.312335    5.312313    0.000022
3.00     5.874125    5.874100    0.000025
```

```
--- 3c) y' = -(y + 1)(y + 3) ---
t     y_aprox      y_real       Error
0.00      -2.000000   -2.000000   0.000000
0.19      -1.808146   -1.808145   0.000000
0.48      -1.553413   -1.553412   0.000001
0.62      -1.445530   -1.445529   0.000001
0.79      -1.341040   -1.341040   0.000000
0.97      -1.251476   -1.251477   0.000001
1.13      -1.188229   -1.188231   0.000001
1.29      -1.139606   -1.139608   0.000001
1.46      -1.102666   -1.102668   0.000002
1.63      -1.074520   -1.074522   0.000002
1.80      -1.053259   -1.053261   0.000002
1.98      -1.037365   -1.037366   0.000002
2.17      -1.025653   -1.025655   0.000002
2.37      -1.017174   -1.017176   0.000002
2.59      -1.011166   -1.011167   0.000002
2.82      -1.007015   -1.007016   0.000001
3.00      -1.004944   -1.004945   0.000001
```

```
--- 3d) y' = (t + 2t^3)^(1/3) y^3 ---
t     y_aprox      y_real       Error
0.00      0.333333    0.333333    0.000000
0.11      0.334223    0.331541    0.002681
0.20      0.335651    0.327241    0.008410
0.53      0.344456    0.294690    0.049766
1.01      0.373203    0.215115    0.158087
1.35      0.418255    0.150559    0.267695
1.51      0.453042    0.122973    0.330069
1.63      0.490837    0.103584    0.387253
1.73      0.536577    0.088268    0.448309
1.82      0.592846    0.076180    0.516666
1.89      0.657382    0.067314    0.590068
1.94      0.729310    0.060879    0.668430
1.99      0.825946    0.055340    0.770605
2.00      0.849153    0.054346    0.794808
```

30 problema 1 (5.6)

```python
import numpy as np
import matplotlib.pyplot as plt


def runge_kutta_4_system(f1, f2, t0, y0, z0, h, steps):
    """Genera valores iniciales para AB4 usando RK4 en sistemas."""
    t = np.zeros(steps + 1)
    y = np.zeros(steps + 1)
```

```
    z = np.zeros(steps + 1)
    t[0], y[0], z[0] = t0, y0, z0

    for i in range(steps):
        k1_y = h * f1(t[i], y[i], z[i])
        k1_z = h * f2(t[i], y[i], z[i])

        k2_y = h * f1(t[i] + h / 2, y[i] + k1_y / 2, z[i] + k1_z / 2)
        k2_z = h * f2(t[i] + h / 2, y[i] + k1_y / 2, z[i] + k1_z / 2)

        k3_y = h * f1(t[i] + h / 2, y[i] + k2_y / 2, z[i] + k2_z / 2)
        k3_z = h * f2(t[i] + h / 2, y[i] + k2_y / 2, z[i] + k2_z / 2)

        k4_y = h * f1(t[i] + h, y[i] + k3_y, z[i] + k3_z)
        k4_z = h * f2(t[i] + h, y[i] + k3_y, z[i] + k3_z)

        y[i + 1] = y[i] + (k1_y + 2 * k2_y + 2 * k3_y + k4_y) / 6
        z[i + 1] = z[i] + (k1_z + 2 * k2_z + 2 * k3_z + k4_z) / 6
        t[i + 1] = t[i] + h

    return t, y, z


def adams_bashforth_4_system(f1, f2, t, y, z, h):
    """AB4 para sistemas de EDOs."""
    for i in range(3, len(t) - 1):
        # Predictor para y (posici n)
        f_y = [f1(t[i - j], y[i - j], z[i - j]) for j in range(4)]
        y[i + 1] = y[i] + h / 24 * (55 * f_y[0] - 59 * f_y[1] + 37 * f_y
   [2] - 9 * f_y[3])

        # Predictor para z (velocidad)
        f_z = [f2(t[i - j], y[i - j], z[i - j]) for j in range(4)]
        z[i + 1] = z[i] + h / 24 * (55 * f_z[0] - 59 * f_z[1] + 37 * f_z
   [2] - 9 * f_z[3])

        t[i + 1] = t[i] + h
    return t, y, z


# Definici n de los sistemas para cada problema
# Problema a: y'' = |t|^2 - 2y      Sistema: y' = z, z' = t^2 - 2y
def f1_a(t, y, z):
    return z


def f2_a(t, y, z):
    return t ** 2 - 2 * y


def y_real_a(t):
    return (1 / 3) * t ** 3 + (1 / 3) * np.exp(t)


```

```python
# Problema b: y'' = 1 + (t - y)^2     Sistema: y' = z, z' = 1 + (t - y)^2
def f1_b(t, y, z):
    return z


def f2_b(t, y, z):
    return 1 + (t - y) ** 2


def y_real_b(t):
    return t + 1 / (t - 1)


# Problema c: y'' = 1 + y/t     Sistema: y' = z, z' = 1 + y/t
def f1_c(t, y, z):
    return z


def f2_c(t, y, z):
    return 1 + y / t if t != 0 else 0


def y_real_c(t):
    return t * np.log(t) + 2 * t


# Problema d: y'' = cos(2t) + sin(3t)     Sistema: y' = z, z' = cos(2t) + sin(3t)
def f1_d(t, y, z):
    return z


def f2_d(t, y, z):
    return np.cos(2 * t) + np.sin(3 * t)


def y_real_d(t):
    return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3


# Configuraci n de los problemas
problemas = [
    {"f1": f1_a, "f2": f2_a, "t0": 0, "y0": 0, "z0": 1 / 3, "t_final": 1, "h": 0.2, "y_real": y_real_a,
     "nombre": "a) y'' = t^2 - 2y"},
    {"f1": f1_b, "f2": f2_b, "t0": 2, "y0": 1, "z0": 1, "t_final": 3, "h": 0.2, "y_real": y_real_b,
     "nombre": "b) y'' = 1 + (t - y)^2"},
    {"f1": f1_c, "f2": f2_c, "t0": 1, "y0": 2, "z0": 1, "t_final": 2, "h": 0.2, "y_real": y_real_c,
     "nombre": "c) y'' = 1 + y/t"},
    {"f1": f1_d, "f2": f2_d, "t0": 0, "y0": 1, "z0": 0, "t_final": 1, "h": 0.2, "y_real": y_real_d,
     "nombre": "d) y'' = cos(2t) + sin(3t)"}
```

```
]

# Resolver cada problema
for problema in problemas:
    # Generar valores iniciales con RK4
    steps_rk4 = 3  # AB4 necesita 4 puntos iniciales
    t_rk4, y_rk4, z_rk4 = runge_kutta_4_system(
        problema["f1"], problema["f2"],
        problema["t0"], problema["y0"], problema["z0"],
        problema["h"], steps_rk4
    )

    # Extender arrays para AB4
    total_steps = int((problema["t_final"] - problema["t0"]) / problema["h"])
    t = np.zeros(total_steps + 1)
    y = np.zeros(total_steps + 1)
    z = np.zeros(total_steps + 1)
    t[:4] = t_rk4
    y[:4] = y_rk4
    z[:4] = z_rk4

    # Aplicar AB4
    t, y, z = adams_bashforth_4_system(problema["f1"], problema["f2"], t, y, z, problema["h"])

    # Calcular soluci n real y error
    y_real = problema["y_real"](t)

    # Resultados
    print(f"\n--- {problema['nombre']} ---")
    print("t\t y_aprox\t y_real\t\t Error")
    for ti, yi, yri in zip(t, y, y_real):
        print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")

    # Gr fica
    plt.figure(figsize=(10, 6))
    plt.plot(t, y, 'bo-', label='AB4 Aproximaci n')
    plt.plot(t, y_real, 'r-', label='Soluci n Real')
    plt.title(f"Comparaci n AB4 vs Real: {problema['nombre']}")
    plt.xlabel("t")
    plt.ylabel("y(t)")
    plt.legend()
    plt.grid()
    plt.show()
```

Listing 11: c

Implementación en Python

A continuación se muestra el código implementado:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

Runge-Kutta de cuarto orden (RK4)

Este método se utiliza para obtener los primeros cuatro valores necesarios para iniciar el método de Adams-Bashforth 4.

```
def runge_kutta_4_system(f1, f2, t0, y0, z0, h, steps):
    t = np.zeros(steps + 1)
    y = np.zeros(steps + 1)
    z = np.zeros(steps + 1)
    t[0], y[0], z[0] = t0, y0, z0

    for i in range(steps):
        k1_y = h * f1(t[i], y[i], z[i])
        k1_z = h * f2(t[i], y[i], z[i])

        k2_y = h * f1(t[i] + h/2, y[i] + k1_y/2, z[i] + k1_z/2)
        k2_z = h * f2(t[i] + h/2, y[i] + k1_y/2, z[i] + k1_z/2)

        k3_y = h * f1(t[i] + h/2, y[i] + k2_y/2, z[i] + k2_z/2)
        k3_z = h * f2(t[i] + h/2, y[i] + k2_y/2, z[i] + k2_z/2)

        k4_y = h * f1(t[i] + h, y[i] + k3_y, z[i] + k3_z)
        k4_z = h * f2(t[i] + h, y[i] + k3_y, z[i] + k3_z)

        y[i+1] = y[i] + (k1_y + 2*k2_y + 2*k3_y + k4_y) / 6
        z[i+1] = z[i] + (k1_z + 2*k2_z + 2*k3_z + k4_z) / 6
        t[i+1] = t[i] + h

    return t, y, z
```

Adams-Bashforth de cuarto orden (AB4)

Una vez que se tienen los primeros 4 valores, se aplica el método AB4 para obtener el resto de la solución.

```
def adams_bashforth_4_system(f1, f2, t, y, z, h):
    for i in range(3, len(t) - 1):
        f_y = [f1(t[i-j], y[i-j], z[i-j]) for j in range(4)]
        y[i+1] = y[i] + h / 24 * (55*f_y[0] - 59*f_y[1] + 37*f_y[2] - 9*f_y[3])

        f_z = [f2(t[i-j], y[i-j], z[i-j]) for j in range(4)]
        z[i+1] = z[i] + h / 24 * (55*f_z[0] - 59*f_z[1] + 37*f_z[2] - 9*f_z[3])

        t[i+1] = t[i] + h

    return t, y, z
```

Definición de problemas

Cada problema se plantea como una ecuación diferencial de segundo orden, que se transforma en un sistema de primer orden.

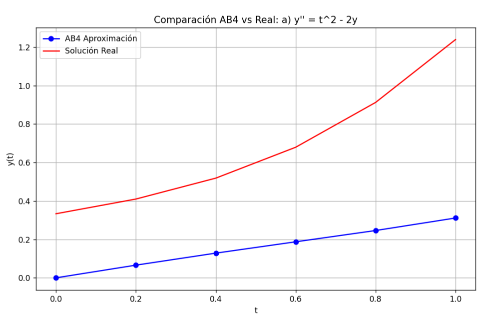
```
# Ejemplo problema a: y'' = t^2 - 2y
def f1_a(t, y, z): return z
def f2_a(t, y, z): return t**2 - 2*y
def y_real_a(t): return (1/3)*t**3 + (1/3)*np.exp(t)
```

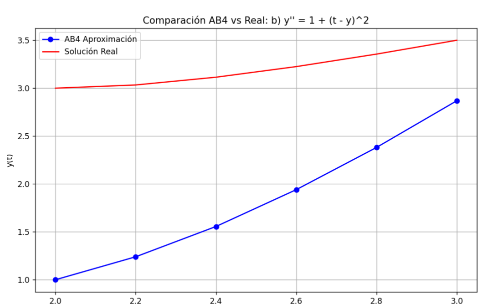
Nota: Se definen funciones similares para los problemas b, c y d.

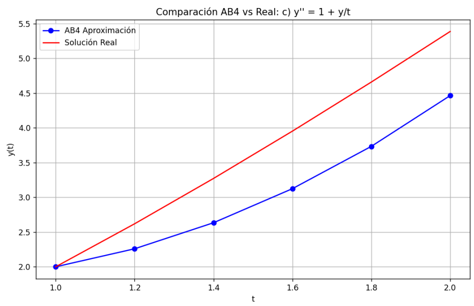
Ejecución y gráficos

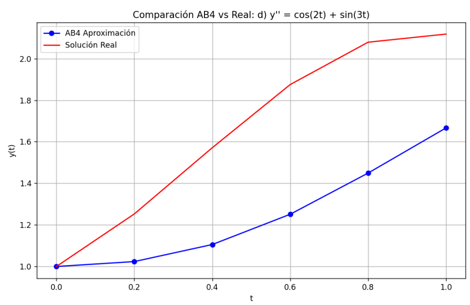
Para cada problema se aplican los métodos mencionados y se grafica la solución numérica junto con la solución exacta.

```
for problema in problemas:
    t_rk4, y_rk4, z_rk4 = runge_kutta_4_system(...)
    t[:4], y[:4], z[:4] = t_rk4, y_rk4, z_rk4
    t, y, z = adams_bashforth_4_system(...)
    y_real = problema["y_real"](t)

    # Imprimir resultados y graficar
    for ti, yi, yri in zip(t, y, y_real):
        print(ti, yi, yri, abs(yi - yri))

    plt.plot(t, y, 'bo-', label='AB4')
    plt.plot(t, y_real, 'r-', label='Exacta')
    plt.legend()
    plt.show()
```










```
t     y_aprox      y_real       Error
0.00      0.000000     0.333333     0.333333
0.20      0.065911     0.409801     0.343890
0.40      0.128441     0.518608     0.390167
0.60      0.187390     0.679373     0.491983
0.80      0.246053     0.912514     0.666460
1.00      0.311178     1.239427     0.928249

--- b) y'' = 1 + (t - y)^2 ---
t     y_aprox      y_real       Error
2.00      1.000000     3.000000     2.000000
2.20      1.239736     3.033333     1.793597
2.40      1.555904     3.114286     1.558382
2.60      1.940339     3.225000     1.284661
2.80      2.381193     3.355556     0.974363
3.00      2.868525     3.500000     0.631475

--- c) y'' = 1 + y/t ---
t     y_aprox      y_real       Error
1.00      2.000000     2.000000     0.000000
1.20      2.258970     2.618786     0.359816
1.40      2.633614     3.271061     0.637447
1.60      3.123741     3.952006     0.828265
1.80      3.732580     4.658016     0.925436
2.00      4.463572     5.386294     0.922722
```

```
--- d) y'' = cos(2t) + sin(3t) ---
t    y_aprox     y_real      Error
0.00     1.000000    1.000000    0.000000
0.20     1.023674    1.252931    0.229256
0.40     1.105615    1.571225    0.465610
0.60     1.251225    1.875087    0.623862
0.80     1.449446    2.078918    0.629472
1.00     1.666684    2.117980    0.451295
```

31 problema 1 (5.9)

```python
import numpy as np
import matplotlib.pyplot as plt


def runge_kutta_4_system(functions, t0, initial_conditions, h, steps):
    """
    M todo de Runge-Kutta de 4to orden para sistemas de EDOs.

    Par metros:
    - functions: Lista de funciones [f1, f2, ..., fn] que definen las EDOs.
    - t0: Tiempo inicial.
    - initial_conditions: Lista de valores iniciales [u1_0, u2_0, ..., un_0].
    - h: Tama o del paso.
    - steps: N mero de pasos.

    Retorna:
    - t: Array de tiempos.
    - solutions: Matriz de soluciones (cada fila corresponde a una variable).
    """
    n = len(functions)
    t = np.zeros(steps + 1)
    solutions = np.zeros((n, steps + 1))
    t[0] = t0
    solutions[:, 0] = initial_conditions

    for i in range(steps):
        k1 = np.zeros(n)
        k2 = np.zeros(n)
        k3 = np.zeros(n)
        k4 = np.zeros(n)

        # C lculo de k1
        args = [t[i]] + list(solutions[:, i])
```

```
        for j in range(n):
            k1[j] = h * functions[j](*args)

        # C lculo de k2
        args_k2 = [t[i] + h / 2] + list(solutions[:, i] + k1 / 2)
        for j in range(n):
            k2[j] = h * functions[j](*args_k2)

        # C lculo de k3
        args_k3 = [t[i] + h / 2] + list(solutions[:, i] + k2 / 2)
        for j in range(n):
            k3[j] = h * functions[j](*args_k3)

        # C lculo de k4
        args_k4 = [t[i] + h] + list(solutions[:, i] + k3)
        for j in range(n):
            k4[j] = h * functions[j](*args_k4)

        # Actualizaci n
        solutions[:, i + 1] = solutions[:, i] + (k1 + 2 * k2 + 2 * k3 + k4) / 6
        t[i + 1] = t[i] + h

    return t, solutions


# ==============================================
# Problema (a)
# ==============================================
def f1_a(t, u1, u2):
    return 3 * u1 + 2 * u2 - (2 * t ** 2 + 1) * np.exp(2 * t)


def f2_a(t, u1, u2):
    return 4 * u1 + u2 + (t ** 2 + 2 * t - 4) * np.exp(2 * t)


def u1_real_a(t):
    return (1 / 2) * np.exp(t ** 2) - (1 / 2) * np.exp(-t) + np.exp(2 * t)


def u2_real_a(t):
    return (1 / 2) * np.exp(t ** 2) + (3 / 2) * np.exp(-t) + t ** 2 * np.exp(2 * t)


# Configuraci n
t0_a = 0
u0_a = [1, 1]  # u1(0) = 1, u2(0) = 1
t_final_a = 1
h_a = 0.2
steps_a = int((t_final_a - t0_a) / h_a)

# Soluci n
```

```
t_a, sol_a = runge_kutta_4_system([f1_a, f2_a], t0_a, u0_a, h_a, steps_a)
u1_real_a_vals = u1_real_a(t_a)
u2_real_a_vals = u2_real_a(t_a)

# Gr ficas
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(t_a, sol_a[0], 'bo-', label='RK4 $u_1(t)$')
plt.plot(t_a, u1_real_a_vals, 'r-', label='Real $u_1(t)$')
plt.title("Problema (a): $u_1(t)$")
plt.xlabel("t")
plt.legend()
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(t_a, sol_a[1], 'bo-', label='RK4 $u_2(t)$')
plt.plot(t_a, u2_real_a_vals, 'r-', label='Real $u_2(t)$')
plt.title("Problema (a): $u_2(t)$")
plt.xlabel("t")
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()


# ==============================================
# Problema (b)
# ==============================================
def f1_b(t, u1, u2):
    return -4 * u1 - 2 * u2 + np.cos(t) + 4 * np.sin(t)


def f2_b(t, u1, u2):
    return 3 * u1 + u2 - 3 * np.sin(t)


def u1_real_b(t):
    return 2 * np.exp(-t) - 2 * np.exp(-2 * t) + np.sin(t)


def u2_real_b(t):
    return -3 * np.exp(-t) + 2 * np.exp(-2 * t)


# Configuraci n
t0_b = 0
u0_b = [0, -1]  # u1(0) = 0, u2(0) = -1
t_final_b = 2
h_b = 0.1
steps_b = int((t_final_b - t0_b) / h_b)

# Soluci n
t_b, sol_b = runge_kutta_4_system([f1_b, f2_b], t0_b, u0_b, h_b, steps_b)
u1_real_b_vals = u1_real_b(t_b)
```

```
u2_real_b_vals = u2_real_b(t_b)

# Gr ficas
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(t_b, sol_b[0], 'bo-', label='RK4 $u_1(t)$')
plt.plot(t_b, u1_real_b_vals, 'r-', label='Real $u_1(t)$')
plt.title("Problema (b): $u_1(t)$")
plt.xlabel("t")
plt.legend()
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(t_b, sol_b[1], 'bo-', label='RK4 $u_2(t)$')
plt.plot(t_b, u2_real_b_vals, 'r-', label='Real $u_2(t)$')
plt.title("Problema (b): $u_2(t)$")
plt.xlabel("t")
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()


# ================================================
# Problema (c) - Sistema de 3 EDOs
# ================================================
def f1_c(t, u1, u2, u3):
    return u2


def f2_c(t, u1, u2, u3):
    return -u1 - 2 * np.exp(t + 1)


def f3_c(t, u1, u2, u3):
    return -u1 - np.exp(t + 1)


def u1_real_c(t):
    return np.cos(t) + np.sin(t) - np.exp(t + 1)


def u2_real_c(t):
    return -np.sin(t) + np.cos(t) - np.exp(t)


def u3_real_c(t):
    return -np.sin(t) + np.cos(t)


# Configuraci n
t0_c = 0
u0_c = [1, 0, 1]  # u1(0)=1, u2(0)=0, u3(0)=1
t_final_c = 2
```

```python
h_c = 0.5
steps_c = int((t_final_c - t0_c) / h_c)

# Soluci n
t_c, sol_c = runge_kutta_4_system([f1_c, f2_c, f3_c], t0_c, u0_c, h_c, steps_c)
u1_real_c_vals = u1_real_c(t_c)
u2_real_c_vals = u2_real_c(t_c)
u3_real_c_vals = u3_real_c(t_c)

# Gr ficas
plt.figure(figsize=(15, 5))
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.plot(t_c, sol_c[i], 'bo-', label=f'RK4 $u_{i + 1}(t)$')
    plt.plot(t_c, [u1_real_c_vals, u2_real_c_vals, u3_real_c_vals][i], 'r-', label=f'Real $u_{i + 1}(t)$')
    plt.title(f"Problema (c): $u_{i + 1}(t)$")
    plt.xlabel("t")
    plt.legend()
    plt.grid()
plt.tight_layout()
plt.show()


# ================================================
# Problema (d) - Sistema de 3 EDOs
# ================================================
def f1_d(t, u1, u2, u3):
    return u2 - u3 + t


def f2_d(t, u1, u2, u3):
    return 3 * t ** 2


def f3_d(t, u1, u2, u3):
    return u2 + np.exp(-t)


def u1_real_d(t):
    return -0.05 * np.exp(t) + 0.25 * t ** 3 + t + 2 - np.exp(-t)


def u2_real_d(t):
    return t ** 3 + 1


def u3_real_d(t):
    return 0.25 * t ** 3 + t - np.exp(-t)


# Configuraci n
t0_d = 0
```

```python
u0_d = [1, 1, -1]  # u1(0)=1, u2(0)=1, u3(0)=-1
t_final_d = 1
h_d = 0.1
steps_d = int((t_final_d - t0_d) / h_d)

# Soluci n
t_d, sol_d = runge_kutta_4_system([f1_d, f2_d, f3_d], t0_d, u0_d, h_d, steps_d)
u1_real_d_vals = u1_real_d(t_d)
u2_real_d_vals = u2_real_d(t_d)
u3_real_d_vals = u3_real_d(t_d)

# Gr ficas
plt.figure(figsize=(15, 5))
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.plot(t_d, sol_d[i], 'bo-', label=f'RK4 $u_{i + 1}(t)$')
    plt.plot(t_d, [u1_real_d_vals, u2_real_d_vals, u3_real_d_vals][i], 'r-', label=f'Real $u_{i + 1}(t)$')
    plt.title(f"Problema (d): $u_{i + 1}(t)$")
    plt.xlabel("t")
    plt.legend()
    plt.grid()
plt.tight_layout()
plt.show()
```

Listing 12: c

El método de Runge-Kutta de cuarto orden (RK4) es una técnica numérica ampliamente utilizada para resolver ecuaciones diferenciales ordinarias (EDOs). Este documento describe su aplicación a sistemas de EDOs de la forma:

$$\begin{cases} \frac{du_1}{dt} = f_1(t, u_1, u_2, \ldots, u_n) \\ \frac{du_2}{dt} = f_2(t, u_1, u_2, \ldots, u_n) \\ \vdots \\ \frac{du_n}{dt} = f_n(t, u_1, u_2, \ldots, u_n) \end{cases}$$

2. Método de Runge-Kutta de Cuarto Orden

Dado un sistema de n EDOs, el método RK4 calcula los valores aproximados en pasos de tamaño h. Para cada paso i, se calcula:

$$
\begin{aligned}
k_1^{(j)} &= h \cdot f_j\left(t_i, u_1^{(i)}, \ldots, u_n^{(i)}\right) \\
k_2^{(j)} &= h \cdot f_j\left(t_i + \frac{h}{2}, u_1^{(i)} + \frac{k_1^{(1)}}{2}, \ldots, u_n^{(i)} + \frac{k_1^{(n)}}{2}\right) \\
k_3^{(j)} &= h \cdot f_j\left(t_i + \frac{h}{2}, u_1^{(i)} + \frac{k_2^{(1)}}{2}, \ldots, u_n^{(i)} + \frac{k_2^{(n)}}{2}\right) \\
k_4^{(j)} &= h \cdot f_j\left(t_i + h, u_1^{(i)} + k_3^{(1)}, \ldots, u_n^{(i)} + k_3^{(n)}\right) \\
u_j^{(i+1)} &= u_j^{(i)} + \frac{1}{6}(k_1^{(j)} + 2k_2^{(j)} + 2k_3^{(j)} + k_4^{(j)})
\end{aligned}
$$

para cada variable u_j, con $j = 1, 2, \ldots, n$.

3. Problemas Resueltos

Problema (a)

Sistema:

$$
\begin{cases} \frac{du_1}{dt} = 3u_1 + 2u_2 - (2t^2 + 1)e^{2t} \\ \frac{du_2}{dt} = 4u_1 + u_2 + (t^2 + 2t - 4)e^{2t} \end{cases}
$$

Condiciones iniciales: $u_1(0) = 1$, $u_2(0) = 1$
Solución exacta:

$$
\begin{aligned}
u_1(t) &= \frac{1}{2}e^{t^2} - \frac{1}{2}e^{-t} + e^{2t} \\
u_2(t) &= \frac{1}{2}e^{t^2} + \frac{3}{2}e^{-t} + t^2 e^{2t}
\end{aligned}
$$

Problema (b)

Sistema:

$$
\begin{cases} \frac{du_1}{dt} = -4u_1 - 2u_2 + \cos(t) + 4\sin(t) \\ \frac{du_2}{dt} = 3u_1 + u_2 - 3\sin(t) \end{cases}
$$

Condiciones iniciales: $u_1(0) = 0$, $u_2(0) = -1$
Solución exacta:

$$
\begin{aligned}
u_1(t) &= 2e^{-t} - 2e^{-2t} + \sin(t) \\
u_2(t) &= -3e^{-t} + 2e^{-2t}
\end{aligned}
$$

Problema (c)

Sistema:

$$
\begin{cases} \frac{du_1}{dt} = u_2 \\ \frac{du_2}{dt} = -u_1 - 2e^{t+1} \\ \frac{du_3}{dt} = -u_1 - e^{t+1} \end{cases}
$$

Condiciones iniciales: $u_1(0) = 1,\ u_2(0) = 0,\ u_3(0) = 1$
Solución exacta:

$$u_1(t) = \cos(t) + \sin(t) - e^{t+1}$$
$$u_2(t) = -\sin(t) + \cos(t) - e^{t}$$
$$u_3(t) = -\sin(t) + \cos(t)$$

Problema (d)

Sistema:

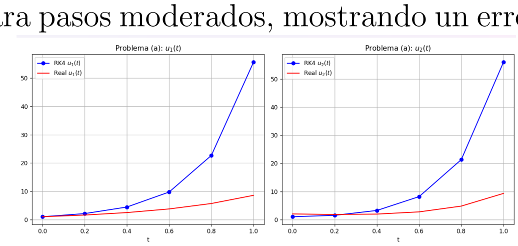
$$\begin{cases} \frac{du_1}{dt} = u_2 - u_3 + t \\ \frac{du_2}{dt} = 3t^2 \\ \frac{du_3}{dt} = u_2 + e^{-t} \end{cases}$$

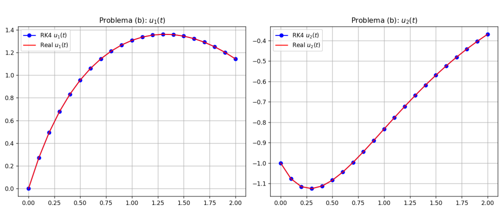
Condiciones iniciales: $u_1(0) = 1,\ u_2(0) = 1,\ u_3(0) = -1$
Solución exacta:

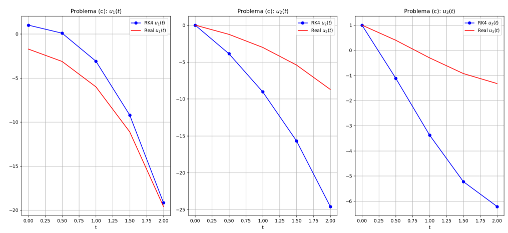
$$u_1(t) = -0.05e^{t} + 0.25t^3 + t + 2 - e^{-t}$$
$$u_2(t) = t^3 + 1$$
$$u_3(t) = 0.25t^3 + t - e^{-t}$$

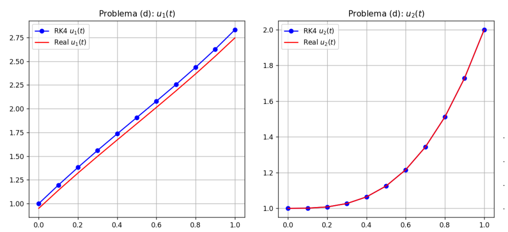
4. Resultados

En cada problema se utiliza el método RK4 para obtener una solución numérica y se compara con la solución exacta a través de gráficos. Se observa que el método RK4 ofrece alta precisión para pasos moderados, mostrando un error pequeño en comparación con la solución analítica.









32 problema 7 (5.9)

```
import numpy as np
import matplotlib.pyplot as plt


def runge_kutta_4_system(f1, f2, t0, x10, x20, h, steps, k1, k2, k3, k4):
    """
    M todo de Runge-Kutta de 4to orden para el sistema Lotka-Volterra.

    Par metros:
    - f1, f2: Funciones que definen las EDOs (dx1/dt y dx2/dt).
    - t0: Tiempo inicial.
    - x10, x20: Poblaciones iniciales de presas y depredadores.
    - h: Tama o del paso.
    - steps: N mero de pasos.
    - k1, k2, k3, k4: Constantes del modelo.

    Retorna:
    - t: Array de tiempos.
    - x1, x2: Arrays de poblaciones de presas y depredadores.
    """
    t = np.zeros(steps + 1)
    x1 = np.zeros(steps + 1)
    x2 = np.zeros(steps + 1)
    t[0], x1[0], x2[0] = t0, x10, x20

    for i in range(steps):
        # Coeficientes k para x1 (presas)
        k1_x1 = h * f1(t[i], x1[i], x2[i], k1, k2)
        k1_x2 = h * f2(t[i], x1[i], x2[i], k3, k4)
```

```
        k2_x1 = h * f1(t[i] + h / 2, x1[i] + k1_x1 / 2, x2[i] + k1_x2 / 2, k1, k2)
        k2_x2 = h * f2(t[i] + h / 2, x1[i] + k1_x1 / 2, x2[i] + k1_x2 / 2, k3, k4)

        k3_x1 = h * f1(t[i] + h / 2, x1[i] + k2_x1 / 2, x2[i] + k2_x2 / 2, k1, k2)
        k3_x2 = h * f2(t[i] + h / 2, x1[i] + k2_x1 / 2, x2[i] + k2_x2 / 2, k3, k4)

        k4_x1 = h * f1(t[i] + h, x1[i] + k3_x1, x2[i] + k3_x2, k1, k2)
        k4_x2 = h * f2(t[i] + h, x1[i] + k3_x1, x2[i] + k3_x2, k3, k4)

        # Actualizaci n
        x1[i + 1] = x1[i] + (k1_x1 + 2 * k2_x1 + 2 * k3_x1 + k4_x1) / 6
        x2[i + 1] = x2[i] + (k1_x2 + 2 * k2_x2 + 2 * k3_x2 + k4_x2) / 6
        t[i + 1] = t[i] + h

    return t, x1, x2


# Definici n de las EDOs del modelo Lotka-Volterra
def f1_presas(t, x1, x2, k1, k2):
    return k1 * x1 - k2 * x1 * x2  # dx1/dt = k1*x1 - k2*x1*x2


def f2_depredadores(t, x1, x2, k3, k4):
    return k3 * x1 * x2 - k4 * x2  # dx2/dt = k3*x1*x2 - k4*x2


# Par metros del modelo (ejemplo)
k1 = 0.4  # Tasa de natalidad de presas
k2 = 0.01  # Tasa de mortalidad de presas por depredadores
k3 = 0.001  # Tasa de natalidad de depredadores por presas
k4 = 0.3  # Tasa de mortalidad de depredadores

# Configuraci n de la simulaci n
t0 = 0
x10 = 50  # Poblaci n inicial de presas
x20 = 20  # Poblaci n inicial de depredadores
t_final = 100
h = 0.1
steps = int((t_final - t0) / h)

# Resolver el sistema
t, x1, x2 = runge_kutta_4_system(f1_presas, f2_depredadores, t0, x10, x20, h, steps, k1, k2, k3, k4)

# Gr ficas
plt.figure(figsize=(12, 6))

# Poblaciones vs tiempo
plt.subplot(1, 2, 1)
```

```python
plt.plot(t, x1, 'g-', label='Presas ($x_1$)')
plt.plot(t, x2, 'r-', label='Depredadores ($x_2$)')
plt.title('Din mica de Poblaciones (Lotka-Volterra)')
plt.xlabel('Tiempo ($t$)')
plt.ylabel('Poblaci n')
plt.legend()
plt.grid()

# Diagrama de fase
plt.subplot(1, 2, 2)
plt.plot(x1, x2, 'b-')
plt.title('Diagrama de Fase ($x_1$ vs $x_2$)')
plt.xlabel('Presas ($x_1$)')
plt.ylabel('Depredadores ($x_2$)')
plt.grid()

plt.tight_layout()
plt.show()
```

Listing 13: c

El modelo de Lotka-Volterra describe la dinámica de dos poblaciones: una de presas y otra de depredadores. Está representado por un sistema de ecuaciones diferenciales no lineales:

$$\frac{dx_1}{dt} = k_1 x_1 - k_2 x_1 x_2 \qquad \text{(presas)}$$

$$\frac{dx_2}{dt} = k_3 x_1 x_2 - k_4 x_2 \qquad \text{(depredadores)}$$

donde:

- $x_1(t)$ es la población de presas,
- $x_2(t)$ es la población de depredadores,
- k_1: tasa de natalidad de presas,
- k_2: tasa de depredación,
- k_3: eficiencia de conversión de presas a depredadores,
- k_4: tasa de mortalidad de depredadores.

2. Método de Runge-Kutta de 4º Orden

Para resolver el sistema se utiliza el método de Runge-Kutta de cuarto orden. El método calcula los valores siguientes usando combinaciones ponderadas de derivadas evaluadas en diferentes puntos del intervalo. Para cada variable del sistema se calcula:

$$k_1 = h \cdot f(t_i, x_1^i, x_2^i)$$
$$k_2 = h \cdot f\left(t_i + \frac{h}{2}, x_1^i + \frac{k_1}{2}, x_2^i + \frac{k_1}{2}\right)$$
$$k_3 = h \cdot f\left(t_i + \frac{h}{2}, x_1^i + \frac{k_2}{2}, x_2^i + \frac{k_2}{2}\right)$$
$$k_4 = h \cdot f(t_i + h, x_1^i + k_3, x_2^i + k_3)$$
$$x^{i+1} = x^i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

3. Implementación en Python

A continuación, se muestra el código que implementa la simulación:

```
def f1_presas(t, x1, x2, k1, k2):
    return k1 * x1 - k2 * x1 * x2

def f2_depredadores(t, x1, x2, k3, k4):
    return k3 * x1 * x2 - k4 * x2

def runge_kutta_4_system(f1, f2, t0, x10, x20, h, steps, k1, k2, k3, k4):
    ...
```

Listing 14: Implementación de RK4 para el modelo Lotka-Volterra

El sistema se resuelve para un intervalo de tiempo $[t_0, t_{\text{final}}]$ con tamaño de paso h.

4. Parámetros del Modelo

Se utilizaron los siguientes valores:

$$\begin{aligned} k_1 &= 0.4 && \text{(natalidad de presas)} \\ k_2 &= 0.01 && \text{(mortalidad por depredadores)} \\ k_3 &= 0.001 && \text{(reproducción de depredadores)} \\ k_4 &= 0.3 && \text{(mortalidad de depredadores)} \end{aligned}$$

Condiciones iniciales:

$$x_1(0) = 50, \quad x_2(0) = 20, \quad t_0 = 0, \quad t_{\text{final}} = 100, \quad h = 0.1$$

5. Resultados Gráficos

5.1 Dinámica de Poblaciones

Se grafica la evolución de ambas poblaciones a lo largo del tiempo:

- Línea verde: población de presas (x_1).
- Línea roja: población de depredadores (x_2).

5.2 Diagrama de Fase

Se grafica la trayectoria de las poblaciones en el plano (x_1, x_2), lo que permite visualizar los ciclos poblacionales típicos del modelo Lotka-Volterra.

6. Conclusiones

El método de Runge-Kutta de cuarto orden permite simular con alta precisión la evolución de sistemas no lineales como el modelo Lotka-Volterra. Se observan oscilaciones periódicas que representan el comportamiento cíclico entre depredadores y presas.