



UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ  
FACULTAD DE ESTUDIOS PROFESIONALES ZONA MEDIA

---

## Práctica 9: Resolución 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 librerías(especificar su uso en el reporte) o crear cada algoritmo

## Desarrollo

### 1 problema 1 (5.1)

```
1
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.integrate import solve_ivp
5
6
7 # Definiciones de los problemas
8 def problema_a(t, y):
9     return y * np.cos(t)
10
11
12 def problema_b(t, y):
13     return 2 / t * y + t ** 2 * np.exp(t)
14
15
16 def problema_c(t, y):
17     return -2 / t * y + t ** 2 * np.exp(t)
18
19
20 def problema_d(t, y):
21     return 4 * t ** 3 * y / (1 + t ** 4)
22
23
24 # Lista de problemas con sus condiciones
25 problemas = [
26     {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "
27     Problema a"},
28     {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "
29     Problema b"},
30     {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "
31     nombre": "Problema c"},
32     {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "
33     Problema d"}
34 ]
35
36 # Tabla de resultados
37 print("RESULTADOS NUM RICOS:")
```

```

34 print("-" * 50)
35 print(f{'Problema':<12} {'y(a)':>8} {'y(b)':>15} {'Intervalo':>12})
36 print("-" * 50)
37
38 # Resolver y graficar
39 for p in problemas:
40     a, b = p["intervalo"]
41     solucion = solve_ivp(p["func"], (a, b), p["y0"], dense_output=True)
42
43     t_vals = np.linspace(a, b, 100)
44     y_vals = solucion.sol(t_vals)[0]
45
46     # Mostrar resultado num rico
47     print(f"{p['nombre']:<12} {p['y0'][0]:>8.4f} {y_vals[-1]:>15.6f} [{a},
48           {b}]")
49
50     # Graficar
51     plt.plot(t_vals, y_vals, label=p["nombre"])
52
53 # Gr fico
54 plt.title("Soluciones de los Problemas de Valor Inicial")
55 plt.xlabel("t")
56 plt.ylabel("y(t)")
57 plt.legend()
58 plt.grid(True)
59 plt.tight_layout()
60 plt.show()
61
62

```

Listing 1: semaforo.c

## Definición del Código

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 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:

```

1 def problema_a(t, y):
2     return y * np.cos(t)
3
4 def problema_b(t, y):
5     return 2 / t * y + t ** 2 * np.exp(t)
6
7 def problema_c(t, y):
8     return -2 / t * y + t ** 2 * np.exp(t)
9
10 def problema_d(t, y):
11     return 4 * t ** 3 * y / (1 + t ** 4)

```

## Configuración de los Problemas

Cada problema se describe mediante:

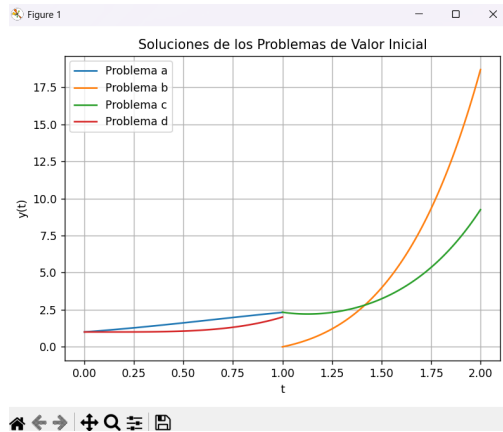
- Una función  $f(t, y)$
- Un intervalo de integración  $[a, b]$
- Una condición inicial  $y(a) = y_0$

```

1 problemas = [
2     {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "
3     Problema a"},
4     {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "
5     Problema b"},
6     {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "
7     nombre": "Problema c"},
8     {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "
9     Problema d"}
10 ]

```

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)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import quad
4
5 # f(t, y) = -y + t + 1
6 def f(t, y):
7     return -y + t + 1
8
9 # y0(t): constante
10 def y0(t):
11     return 1
12
13 # y1(t)
14 def y1(t):
15     integrand = lambda tau: f(tau, y0(tau))
16     return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])
17
18 # y2(t)
19 def y2(t):
20     y1_vals = y1(t)
21     integrand = lambda tau: f(tau, np.interp(tau, t, y1_vals))
22     return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])
23
24 # y3(t)
25 def y3(t):
26     y2_vals = y2(t)
27     integrand = lambda tau: f(tau, np.interp(tau, t, y2_vals))
28     return 1 + np.array([quad(integrand, 0, ti)[0] for ti in t])
29
30 # Soluci n real
31 def y_real(t):
32     return t + np.exp(-t)

```

```

33
34 # Dominio de t
35 t_vals = np.linspace(0, 1, 100)
36
37 # Evaluaciones
38 y0_vals = np.ones_like(t_vals)
39 y1_vals = y1(t_vals)
40 y2_vals = y2(t_vals)
41 y3_vals = y3(t_vals)
42 y_real_vals = y_real(t_vals)
43
44 # Mostrar tabla de resultados en t = 1
45 t1 = 1.0
46 y_real_1 = y_real(t1)
47 y1_1 = y1(np.array([t1]))[0]
48 y2_1 = y2(np.array([t1]))[0]
49 y3_1 = y3(np.array([t1]))[0]
50
51 print("RESULTADOS EN t = 1")
52 print("-" * 40)
53 print(f"{'Aproximaci n':<15} {'Valor':>10} {'Error absoluto':>15}")
54 print("-" * 40)
55 print(f"{' y (t)':<15} {1:>10.6f} {abs(1 - y_real_1):>15.6f}")
56 print(f"{' y (t)':<15} {y1_1:>10.6f} {abs(y1_1 - y_real_1):>15.6f}")
57 print(f"{' y (t)':<15} {y2_1:>10.6f} {abs(y2_1 - y_real_1):>15.6f}")
58 print(f"{' y (t)':<15} {y3_1:>10.6f} {abs(y3_1 - y_real_1):>15.6f}")
59 print(f"{'Exacta':<15} {y_real_1:>10.6f} {'-'*15}")
60
61 # Gr fica
62 plt.plot(t_vals, y0_vals, label=' y (t)', linestyle='--')
63 plt.plot(t_vals, y1_vals, label=' y (t)')
64 plt.plot(t_vals, y2_vals, label=' y (t)')
65 plt.plot(t_vals, y3_vals, label=' y (t)')
66 plt.plot(t_vals, y_real_vals, label='Soluci n exacta', color='black',
        linestyle='dotted')
67
68 plt.title("Aproximaciones por el M todo de Picard")
69 plt.xlabel("t")
70 plt.ylabel("y(t)")
71 plt.legend()
72 plt.grid(True)
73 plt.tight_layout()
74 plt.show()

```

Listing 2: semaforo.c

beginlstlisting import numpy as np import matplotlib.pyplot as plt from scipy.integrate  
import solve<sub>i</sub>vp

Estas bibliotecas permiten:

- numpy: generar arreglos y realizar operaciones numéricas.
- matplotlib.pyplot: crear gráficos.
- solve<sub>i</sub>vp de scipy.integrate: resolver numéricamente EDOs.

## Definición de las Ecuaciones Diferenciales

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

```
1 def problema_a(t, y):
2     return y * np.cos(t)
3
4 def problema_b(t, y):
5     return 2 / t * y + t ** 2 * np.exp(t)
6
7 def problema_c(t, y):
8     return -2 / t * y + t ** 2 * np.exp(t)
9
10 def problema_d(t, y):
11     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  $y(a) = y_0$

```
1 problemas = [
2     {"func": problema_a, "intervalo": (0, 1), "y0": [1], "nombre": "
3     Problema a"},
4     {"func": problema_b, "intervalo": (1, 2), "y0": [0], "nombre": "
5     Problema b"},
6     {"func": problema_c, "intervalo": (1, 2), "y0": [np.sqrt(2 * np.e)], "
7     nombre": "Problema c"},
8     {"func": problema_d, "intervalo": (0, 1), "y0": [1], "nombre": "
9     Problema d"}
10 ]
```

## Resolución y Resultados

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

```
1 for p in problemas:
2     a, b = p["intervalo"]
3     solucion = solve_ivp(p["func"], (a, b), p["y0"], dense_output=True)
4
5     t_vals = np.linspace(a, b, 100)
6     y_vals = solucion.sol(t_vals)[0]
7
```

```

8     print(f"{p['nombre']:<12} {p['y0'][0]:>8.4f} {y_vals[-1]:>15.6f} [{a},
9     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

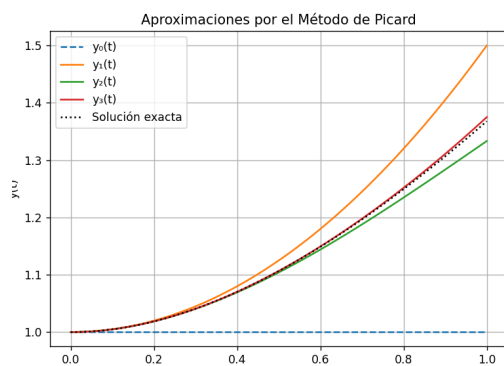
El siguiente código genera el gráfico con todas las soluciones aproximadas:

```

1 plt.title("Soluciones de los Problemas de Valor Inicial")
2 plt.xlabel("t")
3 plt.ylabel("y(t)")
4 plt.legend()
5 plt.grid(True)
6 plt.tight_layout()
7 plt.show()

```

RESULTADOS EN $t = 1$		
Aproximación	Valor	Error absoluto
$y_0(t)$	1.000000	0.367879
$y_1(t)$	1.500000	0.132121
$y_2(t)$	1.000000	0.367879
$y_3(t)$	1.500000	0.132121
Exacta	1.367879	-----



### 3 problema 1 (5.2)



```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def metodo_euler(f, t0, y0, h, pasos):
6     """
7     Aproxima la soluci n de una EDO usando el m todo de Euler.
8
9     Par metros:
10    - f: Funci n  $dy/dt = f(t, y)$ 
11    - t0: Valor inicial de t
12    - y0: Valor inicial de y
13    - h: Tama o del paso
14    - pasos: N mero de pasos
15
16    Retorna:
17    - t_valores: Array de valores de t
18    - y_valores: Array de valores aproximados de y
19    """
20    t_valores = np.zeros(pasos + 1)
21    y_valores = np.zeros(pasos + 1)
22
23    t_valores[0] = t0
24    y_valores[0] = y0
25
26    for i in range(pasos):
27        t = t_valores[i]
28        y = y_valores[i]
29        y_valores[i + 1] = y + h * f(t, y)
30        t_valores[i + 1] = t + h
31
32    return t_valores, y_valores
33
34
35 # Definici n de las EDOs para cada inciso
36 def f_a(t, y):
37     return t * np.exp(3 * t) - 2 * y #  $y' = te^{(3t)} - 2y$ 
38
39
40 def f_b(t, y):
41     return 1 + (t - y) ** 2 #  $y' = 1 + (t - y)^2$ 
42
43
44 def f_c(t, y):
45     return 1 + y / t #  $y' = 1 + y/t$  (asumo que "yh" es un typo y deber a
46     ser y/t)
47
48 def f_d(t, y):
49     return np.cos(2 * t) + np.sin(3 * t) #  $y' = \cos(2t) + \sin(3t)$ 
50
51
52 # Configuraci n para cada problema
53 problemas = [

```

```

54     {"f": f_a, "t0": 0, "y0": 0, "h": 0.5, "t_final": 1, "nombre": "a) y'
      = te^{3t} - 2y"},
55     {"f": f_b, "t0": 2, "y0": 1, "h": 0.5, "t_final": 3, "nombre": "b) y'
      = 1 + (t - y)^2"},
56     {"f": f_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "nombre": "c) y'
      = 1 + y/t"},
57     {"f": f_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "nombre": "d) y'
      = cos(2t) + sin(3t)"}
58 ]
59
60 # Resolver y mostrar resultados para cada problema
61 for problema in problemas:
62     pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
63     t, y = metodo_euler(problema["f"], problema["t0"], problema["y0"],
64                          problema["h"], pasos)
65
66     print(f"\n--- {problema['nombre']} ---")
67     print("t\t y_aprox")
68     for ti, yi in zip(t, y):
69         print(f"{ti:.2f}\t {yi:.6f}")
70
71     # Graficar
72     plt.figure()
73     plt.plot(t, y, 'bo-', label="Aproximaci n Euler")
74     plt.title(f"M todo de Euler: {problema['nombre']}")
75     plt.xlabel("t")
76     plt.ylabel("y(t)")
77     plt.legend()
78     plt.grid()
79     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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def metodo_euler(f, t0, y0, h, pasos):
5     t_valores = np.zeros(pasos + 1)
6     y_valores = np.zeros(pasos + 1)
7
8     t_valores[0] = t0
9     y_valores[0] = y0
10
11     for i in range(pasos):
12         t = t_valores[i]
13         y = y_valores[i]
14         y_valores[i + 1] = y + h * f(t, y)
15         t_valores[i + 1] = t + h
16
17     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)  $y' = \cos(2t) + \sin(3t)$

```

1 def f_a(t, y):
2     return t * np.exp(3 * t) - 2 * y
3
4 def f_b(t, y):
5     return 1 + (t - y) ** 2
6
7 def f_c(t, y):
8     return 1 + y / t
9
10 def f_d(t, y):
11     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.

```

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

```

```

4     {"f": f_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "nombre": "c) y'
      = 1 + y/t"},
5     {"f": f_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "nombre": "d) y'
      = cos(2t) + sin(3t)"}
6 ]

```

## Iteración y Gráfica

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

```

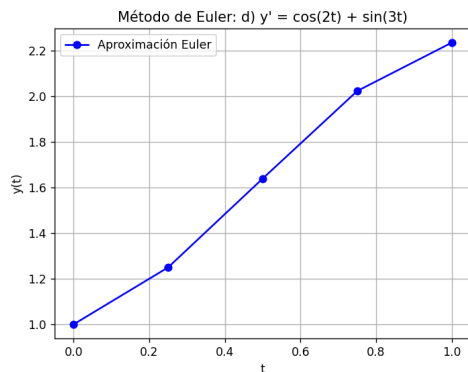
1 for problema in problemas:
2     pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
3     t, y = metodo_euler(problema["f"], problema["t0"], problema["y0"],
4                         problema["h"], pasos)
5
6     print(f"\n--- {problema['nombre']} ---")
7     print("t\t y_aprox")
8     for ti, yi in zip(t, y):
9         print(f"{ti:.2f}\t {yi:.6f}")
10
11     plt.figure()
12     plt.plot(t, y, 'bo-', label="Aproximación Euler")
13     plt.title(f"Método de Euler: {problema['nombre']}")
14     plt.xlabel("t")
15     plt.ylabel("y(t)")
16     plt.legend()
17     plt.grid()
18     plt.show()

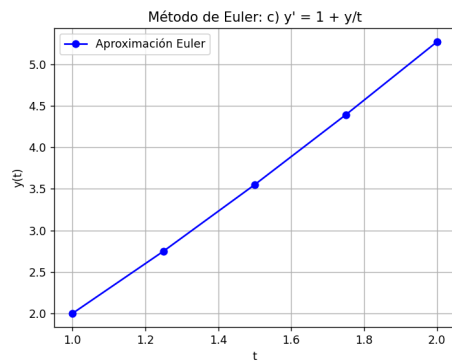
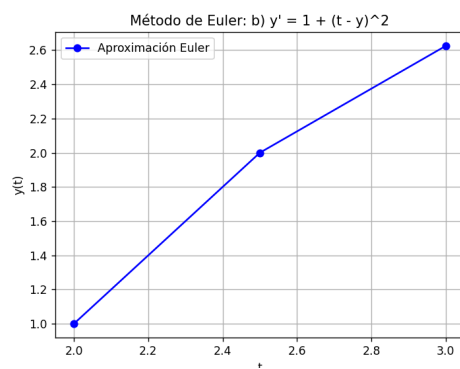
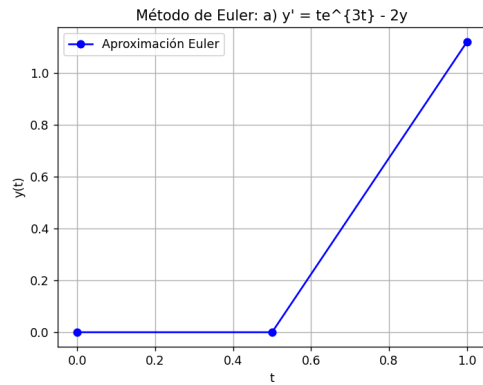
```

```

C:\Users\argi\PycharmProjects\Proyecto_3\venv\Scripts\python.exe "C:\Users\argi\PycharmProjects\Proyecto_3\problemas\1.2.py"
--- a) y' = 5e^(2t) - 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   2.420000
--- c) y' = 1 + y^2 ---
t      y_aprox
1.00   2.000000
1.50   3.500000
1.90   4.800000
2.00   5.200000
--- d) y' = cos(2t) + sin(3t) ---
t      y_aprox
0.00   1.000000
0.25   1.250000
0.50   1.625000
0.75   2.025000
1.00   2.250000

```





## 5 problema 11 (5.2)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sympy import symbols, Function, Eq, Derivative, dsolve, exp
4
5 # -----

```

```

6 # 1. Demostraci n Anal tica (Inciso a)
7 # -----
8 print("\n--- Parte a: Derivaci n de la ecuaci n para p(t) ---")
9 t, b, d, r = symbols('t b d r')
10 x = Function('x')(t)
11 xn = Function('x_n')(t)
12 p = Function('p')(t)
13
14 # Ecuaciones originales
15 dx_dt = Eq(Derivative(x, t), (b - d) * x)
16 dxn_dt = Eq(Derivative(xn, t), (b - d) * xn + r * b * (x - xn))
17
18 # Definici n de p(t) = xn(t)/x(t)
19 p_definition = Eq(p, xn / x)
20
21 # Derivamos p(t) usando regla del cociente
22 dp_dt = Derivative(p, t).doit().subs({
23     Derivative(xn, t): dxn_dt.rhs,
24     Derivative(x, t): dx_dt.rhs
25 }).simplify()
26
27 print(f"Ecuaci n simplificada para dp/dt: {dp_dt} = {dp_dt.simplify()}")
28
29 # -----
30 # 2. Soluci n Num rica con Euler (Inciso b)
31 # -----
32 print("\n--- Parte b: Aproximaci n Num rica con M todo de Euler ---")
33
34 # Par metros
35 b_val = 0.02
36 d_val = 0.015
37 r_val = 0.1
38 p0 = 0.01
39 t_final = 50
40 h = 1 # Tama o de paso (1 a o)
41
42 # Definici n de la EDO dp/dt = r*b*(1 - p)
43 def dpdt(t, p):
44     return r_val * b_val * (1 - p)
45
46 # M todo de Euler
47 def euler_method(f, t0, y0, h, steps):
48     t = np.zeros(steps + 1)
49     y = np.zeros(steps + 1)
50     t[0], y[0] = t0, y0
51     for i in range(steps):
52         y[i+1] = y[i] + h * f(t[i], y[i])
53         t[i+1] = t[i] + h
54     return t, y
55
56 # Calculamos pasos
57 steps = int(t_final / h)
58 t_num, p_num = euler_method(dpdt, 0, p0, h, steps)
59

```

```

60 # Resultado en t=50
61 print(f"Aproximaci n num rica en t=50: p(50)      {p_num[-1]:.6f}")
62
63 # -----
64 # 3. Soluci n Exacta (Inciso c)
65 # -----
66 print("\n--- Parte c: Soluci n Exacta y Comparaci n ---")
67
68 # Resoluci n simb lica con sympy
69 p_exact_eq = dsolve(Eq(Derivative(p, t), r * b * (1 - p)), p, ics={p.subs(
    t, 0): p0})
70 p_exact = p_exact_eq.rhs.subs({b: b_val, r: r_val})
71
72 # Evaluamos en t=50
73 p_exact_50 = p_exact_eq.rhs.subs({t: 50, b: b_val, r: r_val}).evalf()
74 print(f"Soluci n exacta en t=50: p(50) = {p_exact_50:.6f}")
75
76 # Error absoluto
77 error = abs(p_num[-1] - float(p_exact_50))
78 print(f"Error absoluto: {error:.10f}")
79
80 # -----
81 # 4. Gr ficas
82 # -----
83 # Puntos para la soluci n exacta
84 t_exact_vals = np.linspace(0, t_final, 500)
85 p_exact_vals = [1 - (1 - p0) * np.exp(-r_val * b_val * ti) for ti in
    t_exact_vals]
86
87 # Configuraci n de la gr fica
88 plt.figure(figsize=(10, 6))
89 plt.plot(t_num, p_num, 'bo-', label=f'Aproximaci n Euler (h={h})',
    markersize=4)
90 plt.plot(t_exact_vals, p_exact_vals, 'r-', label='Soluci n Exacta')
91 plt.title('Evoluci n de la Proporci n de No Conformistas $p(t)$')
92 plt.xlabel('Tiempo $t$ (a os)')
93 plt.ylabel('Proporci n $p(t)$')
94 plt.legend()
95 plt.grid()
96 plt.show()

```

Listing 4: semaforo.c

. section\*1. Derivaci3n Anal3tica

Sean  $x(t)$  la poblaci3n total,  $x_n(t)$  la cantidad de no conformistas, y  $p(t) = \frac{x_n(t)}{x(t)}$  la proporci3n 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:**

```

1 def dpdt(t, p):
2     return r_val * b_val * (1 - p)
3
4 def euler_method(f, t0, y0, h, steps):
5     t = np.zeros(steps + 1)
6     y = np.zeros(steps + 1)
7     t[0], y[0] = t0, y0
8     for i in range(steps):
9         y[i+1] = y[i] + h * f(t[i], y[i])
10        t[i+1] = t[i] + h
11    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:

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

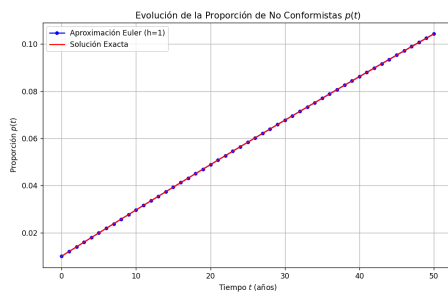


## 4. Comparación y Gráfica

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

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

```
--- Parte a: Derivación de la ecuación para p(t) ---  
Ecuación simplificada para dp/dt: Derivative(p(t), t) = Derivative(p(t), t)  
  
--- Parte b: Aproximación Numérica con Método de Euler ---  
Aproximación numérica en t=50: p(50) = 0.104301  
  
--- Parte c: Solución Exacta y Comparación ---  
Solución exacta en t=50: p(50) = 0.104211  
Error absoluto: 0.0000896940  
  
Process finished with exit code 0
```



## 6 problema 1,2,3 (5.3)

```
1 import numpy as np  
2 import matplotlib.pyplot as plt  
3  
4  
5 def rkf45(f, t0, y0, t_final, hmax, hmin, tol):  
6     """  
7     M todo de Runge-Kutta-Fehlberg (RKF45) para resolver EDOs con control  
8     adaptativo del paso.  
9  
10    Par metros:  
11    - f: Funci n dy/dt = f(t, y)  
12    - t0: Tiempo inicial  
13    - y0: Valor inicial de y  
14    - t_final: Tiempo final  
15    - hmax: Tama o m ximo del paso  
16    - hmin: Tama o m nimo del paso  
17    - tol: Tolerancia permitida  
18  
19    Retorna:  
20    - t_valores: Array de valores de t
```

```

20 - y_valores: Array de valores aproximados de y
21 """
22 t_valores = [t0]
23 y_valores = [y0]
24 h = hmax # Paso inicial
25
26 t = t0
27 y = y0
28
29 while t < t_final:
30     if t + h > t_final:
31         h = t_final - t
32
33     # Coeficientes de RKF45
34     k1 = h * f(t, y)
35     k2 = h * f(t + h / 4, y + k1 / 4)
36     k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
37     k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 /
2197 + 7296 * k3 / 2197)
38     k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 -
845 * k4 / 4104)
39     k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565
+ 1859 * k4 / 4104 - 11 * k5 / 40)
40
41     # Estimaciones de orden 4 y 5
42     y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5
/ 5
43     y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 -
9 * k5 / 50 + 2 * k6 / 55
44
45     # Estimaci n del error
46     error = np.abs(y5 - y4)
47
48     # Control del paso
49     if error <= tol:
50         t += h
51         y = y4
52         t_valores.append(t)
53         y_valores.append(y)
54
55     # Ajuste del paso
56     if error != 0:
57         h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
58     else:
59         h = hmax
60
61     return np.array(t_valores), np.array(y_valores)
62
63
64 # Definici n de las EDOs para cada problema
65 def f_a(t, y):
66     return y * (t - (y / t) ** 2) if t != 0 else 0
67
68

```

```

69 def f_b(t, y):
70     return 1 + (t - y) ** 2
71
72
73 def f_c(t, y):
74     return 1 + y / t if t != 0 else 0
75
76
77 def f_d(t, y):
78     return np.cos(2 * t) + np.sin(3 * t)
79
80
81 # Soluciones reales para comparaci n
82 def y_real_a(t):
83     return 0.5 * np.exp(t) - (1 / 3) * np.exp(t) + (1 / 3) * np.exp(-2 * t
84 )
85
86 def y_real_b(t):
87     return t + 1 / (1 - t)
88
89
90 def y_real_c(t):
91     return t * np.log(t) + 2 * t
92
93
94 def y_real_d(t):
95     return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3
96
97
98 # Configuraci n de los problemas
99 problemas = [
100     {"f": f_a, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin": 0.05,
101      "tol": 1e-4, "y_real": y_real_a,
102      "nombre": "a)  $y' = y(t - (y/t)^2)$ "},
103     {"f": f_b, "t0": 2, "y0": 1, "t_final": 3, "hmax": 0.25, "hmin": 0.05,
104      "tol": 1e-4, "y_real": y_real_b,
105      "nombre": "b)  $y' = 1 + (t - y)^2$ "},
106     {"f": f_c, "t0": 1, "y0": 2, "t_final": 2, "hmax": 0.25, "hmin": 0.05,
107      "tol": 1e-4, "y_real": y_real_c,
108      "nombre": "c)  $y' = 1 + y/t$ "},
109     {"f": f_d, "t0": 0, "y0": 1, "t_final": 1, "hmax": 0.25, "hmin": 0.05,
110      "tol": 1e-4, "y_real": y_real_d,
111      "nombre": "d)  $y' = \cos(2t) + \sin(3t)$ "}
112 ]
113
114 # Resolver y graficar cada problema
115 for problema in problemas:
116     t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["
117 t_final"], problema["hmax"], problema["hmin"],
118                 problema["tol"])
119     y_real = problema["y_real"](t)
120     print(f"\n--- {problema['nombre']} ---")

```

```

117 print("\t\t y_aprox\t y_real\t\t Error")
118 for ti, yi, yri in zip(t, y, y_real):
119     print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")
120
121 plt.figure(figsize=(10, 6))
122 plt.plot(t, y, 'bo-', label='Aproximaci n RKF45')
123 plt.plot(t, y_real, 'r-', label='Soluci n Real')
124 plt.title(f"Comparaci n RKF45 vs Real: {problema['nombre']}")
125 plt.xlabel("t")
126 plt.ylabel("y(t)")
127 plt.legend()
128 plt.grid()
129 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

```

1 def
2
3
4
5
6 while
7     # Calcula los 6 coeficientes k
8
9
10    # ... (k3 a k6 similares)
11
12    # Estimaciones de orden 4 y 5
13
14

```

```

15
16     # Control de error y ajuste de paso
17         abs
18     if
19
20
21
22
23
24         min         max

```

## 7.2 Problemas Resueltos

Se implementan 4 EDOs diferentes:

### 7.2.1 Problema a)

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

```

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

```

### 7.2.2 Problema b)

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

```

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

```

### 7.2.3 Problema c)

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

```

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

```

### 7.2.4 Problema d)

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

```

1 def f_d(t, y):
2     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

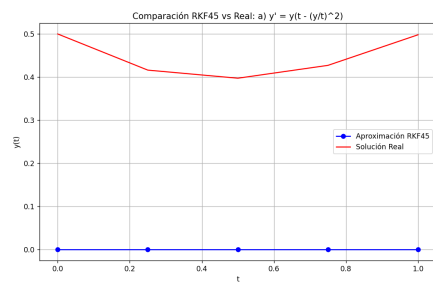
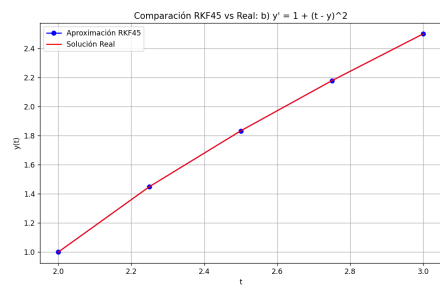
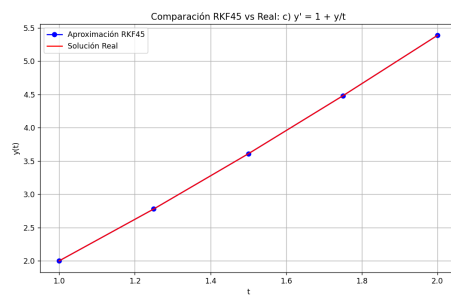
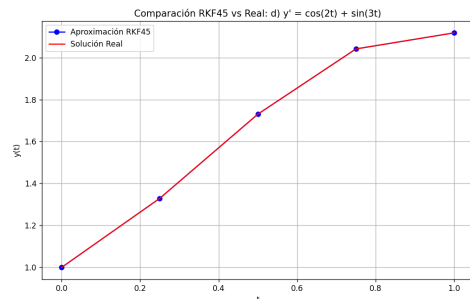
- Eficiencia computacional al adaptar el tamaño de paso
- Mayor precisión que métodos de paso fijo
- Ideal para problemas con comportamientos variables

```
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
```



## 11 problema 1 (5.4)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def taylor_orden_2(f, df, t0, y0, h, pasos):
6     """
7     Aplica el m todo de Taylor de orden 2 para resolver una EDO.
8
9     Par metros:
10    - f: Funci n dy/dt = f(t, y)
11    - df: Derivada total de f respecto a t (df/dt + df/dy * f)
12    - t0: Valor inicial de t
13    - y0: Valor inicial de y
14    - h: Tama o del paso
15    - pasos: N mero de pasos a realizar
16
17    Retorna:
18    - t_valores: Array de valores de t
19    - y_valores: Array de valores aproximados de y
20    """
21    t_valores = np.zeros(pasos + 1)
22    y_valores = np.zeros(pasos + 1)
23
24    t_valores[0] = t0
25    y_valores[0] = y0
26
27    for i in range(pasos):
28        t = t_valores[i]
29        y = y_valores[i]
30        # M todo de Taylor orden 2:  $y_{n+1} = y_n + h*f(t_n, y_n) + (h^2/2)*df(t_n, y_n)$ 
31        y_valores[i + 1] = y + h * f(t, y) + (h ** 2 / 2) * df(t, y)
32        t_valores[i + 1] = t + h
33
34    return t_valores, y_valores
35
36
37 # Definir las funciones f(t, y) y sus derivadas totales df/dt para cada
    inciso
38 # a)  $y' = te^{(3t)} - 2y$ 
39 def f_a(t, y):
40     return t * np.exp(3 * t) - 2 * y
41
42
43 def df_a(t, y):
44     return np.exp(3 * t) * (3 * t + 1) - 2 * f_a(t, y) # df/dt =  $e^{(3t)}(3t + 1) - 2dy/dt$ 
45
46
47 # b)  $y' = 1 + (t - y)^2$ 
48 def f_b(t, y):
```



```

49     return 1 + (t - y) ** 2
50
51
52 def df_b(t, y):
53     return 2 * (t - y) * (1 - f_b(t, y)) # df/dt = 2(t - y)(1 - dy/dt)
54
55
56 # c) y' = 1 + y/t
57 def f_c(t, y):
58     return 1 + y / t
59
60
61 def df_c(t, y):
62     return -y / t ** 2 + f_c(t, y) / t # df/dt = -y/t + (1 + y/t)/t
63
64
65 # d) y' = cos(2t) + sin(3t)
66 def f_d(t, y):
67     return np.cos(2 * t) + np.sin(3 * t)
68
69
70 def df_d(t, y):
71     return -2 * np.sin(2 * t) + 3 * np.cos(3 * t) # df/dt no depende de y
72
73
74 # Configuraci3n para cada problema
75 problemas = [
76     {"f": f_a, "df": df_a, "t0": 0, "y0": 0, "h": 0.5, "t_final": 1, "
77     nombre": "a) y' = t3 - 2y"},
78     {"f": f_b, "df": df_b, "t0": 2, "y0": 1, "h": 0.5, "t_final": 3, "
79     nombre": "b) y' = 1 + (t - y)2"},
80     {"f": f_c, "df": df_c, "t0": 1, "y0": 2, "h": 0.25, "t_final": 2, "
81     nombre": "c) y' = 1 + y/t"},
82     {"f": f_d, "df": df_d, "t0": 0, "y0": 1, "h": 0.25, "t_final": 1, "
83     nombre": "d) y' = cos(2t) + sin(3t)"}
84 ]
85
86 # Resolver y mostrar resultados para cada problema
87 for problema in problemas:
88     pasos = int((problema["t_final"] - problema["t0"]) / problema["h"])
89     t, y = taylor_orden_2(problema["f"], problema["df"], problema["t0"],
90     problema["y0"], problema["h"], pasos)
91
92     print(f"\n--- {problema['nombre']} ---")
93     print("t\t y_aprox")
94     for ti, yi in zip(t, y):
95         print(f"{ti:.2f}\t {yi:.6f}")
96
97     # Graficar
98     plt.figure()
99     plt.plot(t, y, 'bo-', label="Aproximaci3n Taylor Orden 2")
100    plt.title(f"M3todo de Taylor Orden 2: {problema['nombre']}")
101    plt.xlabel("t")
102    plt.ylabel("y(t)")

```

```

98 plt.legend()
99 plt.grid()
100 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

```

1 def
2
3
4
5
6
7
8     for     in range
9
10
11
12
13
14     return

```

### 13.2 Problemas Resueltos

Se implementan 4 problemas diferentes:

### 13.2.1 Problema a)

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

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

### 13.2.2 Problema b)

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

```
1 def f_b(t, y):
2     return 1 + (t - y)**2
3
4 def df_b(t, y):
5     return 2*(t - y)*(1 - f_b(t, y))
```

### 13.2.3 Problema c)

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

```
1 def f_c(t, y):
2     return 1 + y/t
3
4 def df_c(t, y):
5     return -y/t**2 + f_c(t, y)/t
```

### 13.2.4 Problema d)

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

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

## 14 Flujo del Programa

Para cada problema:

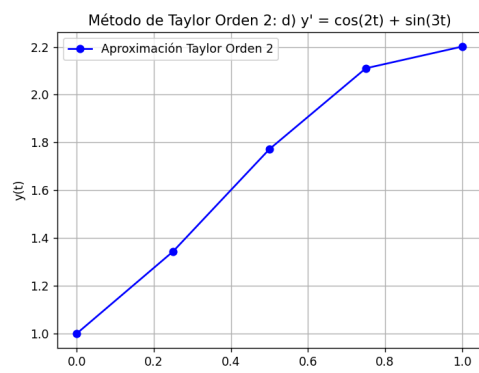
1. Calcula el número de pasos necesarios
2. Aplica el método de Taylor
3. Imprime los resultados en forma de tabla
4. Genera una gráfica de la solución aproximada

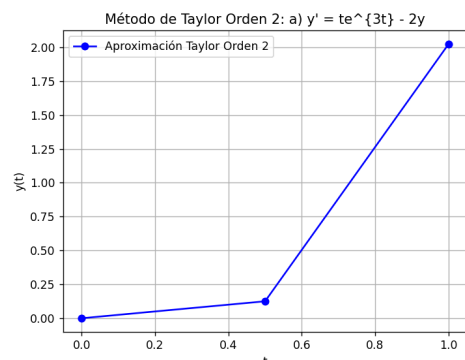
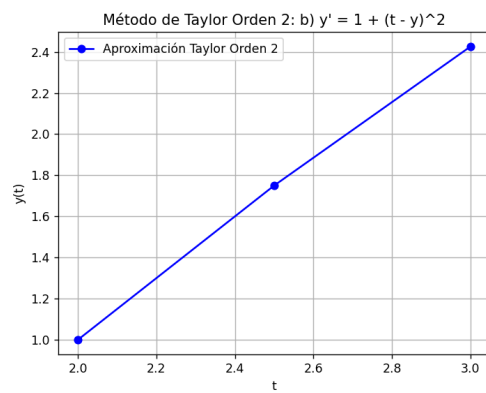
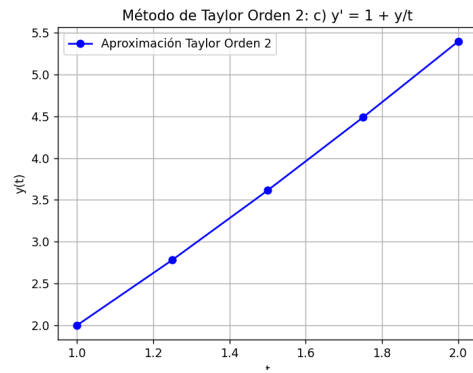
## 15 Salida

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)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Par metros de la reacci n
```

```

5 k = 6.22e-19
6 n1 = 2e3
7 n2 = 2e3
8 n3 = 3e3
9
10 # Ecuación diferencial: dx/dt = k * (n1 - x/2)^2 * (n2 - x/2)^2 * (n3 - 3
    x/4)^3
11 def dxdt(t, x):
12     return k * (n1 - x/2)**2 * (n2 - x/2)**2 * (n3 - 3*x/4)**3
13
14 # Método de Euler
15 def euler_method(f, t0, x0, h, steps):
16     t = np.zeros(steps + 1)
17     x = np.zeros(steps + 1)
18     t[0], x[0] = t0, x0
19     for i in range(steps):
20         x[i+1] = x[i] + h * f(t[i], x[i])
21         t[i+1] = t[i] + h
22     return t, x
23
24 # Configuración de la simulación
25 t0 = 0          # Tiempo inicial (s)
26 x0 = 0          # Cantidad inicial de KOH (molculas)
27 t_final = 0.2   # Tiempo final (s)
28 h = 1e-4        # Tamaño del paso (s) (pequeño para mayor precisión)
29 steps = int((t_final - t0) / h)
30
31 # Solución numérica
32 t, x = euler_method(dxdt, t0, x0, h, steps)
33
34 # Resultado en t = 0.2 s
35 print(f"Cantidad de KOH formado en t = {t_final} s: {x[-1]:.2f} molculas
    ")
36
37 # Gráfica
38 plt.figure(figsize=(10, 6))
39 plt.plot(t, x, 'b-', label='KOH formado')
40 plt.title('Formación de KOH en la reacción química')
41 plt.xlabel('Tiempo $t$ (s)')
42 plt.ylabel('Cantidad de KOH $x(t)$ (molculas)')
43 plt.legend()
44 plt.grid()
45 plt.show()

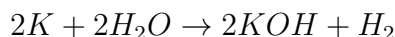
```

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:



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 ( $6.22 \times 10^{-19}$ )
- $n_1, n_2$ : Concentración inicial de K y H<sub>2</sub>O ( $2 \times 10^3$  moléculas)
- $n_3$ : Concentración inicial de otro reactivo ( $3 \times 10^3$  moléculas)

## 18 Implementación en Python

### 18.1 Función de la Ecuación Diferencial

```
1 def
2     return
```

### 18.2 Método de Euler

```
1 def
2
3
4
5     for     in range
6
7
8     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

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

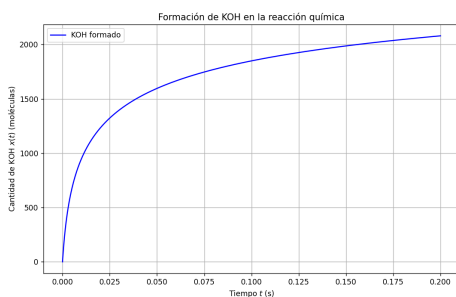
## 21 Visualización

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



```
Cantidad de KOH formado en t = 0.2 s: 2079.82 moléculas
```



## 23 problema 1 (5.5)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
6     t_valores = [t0]
7     y_valores = [y0]
8     h = hmax
9     t = t0
10    y = y0
11
12    while t < t_final:
13        if t + h > t_final:
14            h = t_final - t
15
16        k1 = h * f(t, y)
17        k2 = h * f(t + h / 4, y + k1 / 4)
18        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
19        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 /
20        2197 + 7296 * k3 / 2197)
21        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 -
22        845 * k4 / 4104)
23        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565
24        + 1859 * k4 / 4104 - 11 * k5 / 40)
25
26        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5
27        / 5
28        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 -
29        9 * k5 / 50 + 2 * k6 / 55
30
31        error = np.abs(y5 - y4)
32
33        if error <= tol:
34            t += h
35            y = y4
36            t_valores.append(t)
37            y_valores.append(y)
38
39        if error != 0:
40            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
41        else:
42            h = hmax
43
44    return np.array(t_valores), np.array(y_valores)
45
46
47 # Definiciones para el Problema 1
48 def f_1a(t, y):
49     return y * (t - (y / t) ** 2) if t != 0 else 0
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

```

47 def y_real_1a(t):
48     return 0.5 * np.exp(t) - (1 / 3) * np.exp(t) + (1 / 3) * np.exp(-2 * t
49 )
50
51 def f_1b(t, y):
52     return 1 + (t - y) ** 2
53
54
55 def y_real_1b(t):
56     return t + 1 / (1 - t)
57
58
59 def f_1c(t, y):
60     return 1 + y / t if t != 0 else 0
61
62
63 def y_real_1c(t):
64     return t * np.log(t) + 2 * t
65
66
67 def f_1d(t, y):
68     return np.cos(2 * t) + np.sin(3 * t)
69
70
71 def y_real_1d(t):
72     return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3
73
74
75 problemas_1 = [
76     {"f": f_1a, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin":
77     0.05, "tol": 1e-4, "y_real": y_real_1a,
78     "nombre": "1a)  $y' = y(t - (y/t)^2)$ "},
79     {"f": f_1b, "t0": 2, "y0": 1, "t_final": 3, "hmax": 0.25, "hmin":
80     0.05, "tol": 1e-4, "y_real": y_real_1b,
81     "nombre": "1b)  $y' = 1 + (t - y)^2$ "},
82     {"f": f_1c, "t0": 1, "y0": 2, "t_final": 2, "hmax": 0.25, "hmin":
83     0.05, "tol": 1e-4, "y_real": y_real_1c,
84     "nombre": "1c)  $y' = 1 + y/t$ "},
85     {"f": f_1d, "t0": 0, "y0": 1, "t_final": 1, "hmax": 0.25, "hmin":
86     0.05, "tol": 1e-4, "y_real": y_real_1d,
87     "nombre": "1d)  $y' = \cos(2t) + \sin(3t)$ "}
88 ]
89
90 for problema in problemas_1:
91     t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["
92     t_final"], problema["hmax"], problema["hmin"],
93     problema["tol"])
94     y_real = problema["y_real"](t)
95
96     print(f"\n--- {problema['nombre']} ---")
97     print("t\t y_aprox\t y_real\t\t Error")
98     for ti, yi, yri in zip(t, y, y_real):
99         print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")

```

```

95 plt.figure(figsize=(10, 6))
96 plt.plot(t, y, 'bo-', label='RKF45')
97 plt.plot(t, y_real, 'r-', label='Soluci n Real')
98 plt.title(problema["nombre"])
99 plt.xlabel("t")
100 plt.ylabel("y(t)")
101 plt.legend()
102 plt.grid()
103 plt.show()
104

```

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)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
6     t_valores = [t0]
7     y_valores = [y0]
8     h = hmax

```

```

9     t = t0
10    y = y0
11
12    while t < t_final:
13        if t + h > t_final:
14            h = t_final - t
15
16            k1 = h * f(t, y)
17            k2 = h * f(t + h / 4, y + k1 / 4)
18            k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
19            k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 /
20            2197 + 7296 * k3 / 2197)
21            k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 -
22            845 * k4 / 4104)
23            k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565
24            + 1859 * k4 / 4104 - 11 * k5 / 40)
25
26            y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5
27            / 5
28            y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 -
29            9 * k5 / 50 + 2 * k6 / 55
30
31            error = np.abs(y5 - y4)
32
33            if error <= tol:
34                t += h
35                y = y4
36                t_valores.append(t)
37                y_valores.append(y)
38
39            if error != 0:
40                h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
41            else:
42                h = hmax
43
44    return np.array(t_valores), np.array(y_valores)
45
46
47 # Definiciones para el Problema 2 (sin soluciones reales)
48 def f_2a(t, y):
49     return (y / t) ** 2 + y / t if t != 0 else 0
50
51 def f_2b(t, y):
52     return 1 / np.cos(t) + np.exp(t)
53
54 def f_2c(t, y):
55     return 1 / (t ** 2 + y ** 2) if (t ** 2 + y ** 2) != 0 else 0
56
57 def f_2d(t, y):
58     return t ** 2

```

```

58
59 problemas_2 = [
60     {"f": f_2a, "t0": 1, "y0": 1, "t_final": 1.2, "hmax": 0.005, "hmin":
61     0.02, "tol": 1e-4,
62     "nombre": "2a)  $y' = (y/t)^2 + y/t$ "},
63     {"f": f_2b, "t0": 0, "y0": 0, "t_final": 1, "hmax": 0.25, "hmin":
64     0.02, "tol": 1e-4,
65     "nombre": "2b)  $y' = \sec(t) + e^t$ "},
66     {"f": f_2c, "t0": 1, "y0": -2, "t_final": 3.2, "hmax": 0.5, "hmin":
67     0.02, "tol": 1e-4,
68     "nombre": "2c)  $y' = 1/(t^2 + y^2)$ "},
69     {"f": f_2d, "t0": 0, "y0": 0, "t_final": 2, "hmax": 0.5, "hmin": 0.02,
70     "tol": 1e-4, "nombre": "2d)  $y' = t^2$ "}
71 ]
72
73 for problema in problemas_2:
74     t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["
75     t_final"], problema["hmax"], problema["hmin"],
76     problema["tol"])
77
78     print(f"\n--- {problema['nombre']} ---")
79     print("t\t y_aprox")
80     for ti, yi in zip(t, y):
81         print(f"{ti:.2f}\t {yi:.6f}")
82
83     plt.figure(figsize=(10, 6))
84     plt.plot(t, y, 'bo-', label='RK45')
85     plt.title(problema["nombre"])
86     plt.xlabel("t")
87     plt.ylabel("y(t)")
88     plt.legend()
89     plt.grid()
90     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)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def rkf45(f, t0, y0, t_final, hmax, hmin, tol):
6     t_valores = [t0]
7     y_valores = [y0]
8     h = hmax
9     t = t0
10    y = y0
11
12    while t < t_final:
13        if t + h > t_final:
14            h = t_final - t
15
16        k1 = h * f(t, y)
17        k2 = h * f(t + h / 4, y + k1 / 4)
18        k3 = h * f(t + 3 * h / 8, y + 3 * k1 / 32 + 9 * k2 / 32)
19        k4 = h * f(t + 12 * h / 13, y + 1932 * k1 / 2197 - 7200 * k2 /
20        2197 + 7296 * k3 / 2197)
21        k5 = h * f(t + h, y + 439 * k1 / 216 - 8 * k2 + 3680 * k3 / 513 -
22        845 * k4 / 4104)
23        k6 = h * f(t + h / 2, y - 8 * k1 / 27 + 2 * k2 - 3544 * k3 / 2565
24        + 1859 * k4 / 4104 - 11 * k5 / 40)
25
26        y4 = y + 25 * k1 / 216 + 1408 * k3 / 2565 + 2197 * k4 / 4104 - k5
27        / 5
28        y5 = y + 16 * k1 / 135 + 6656 * k3 / 12825 + 28561 * k4 / 56430 -
29        9 * k5 / 50 + 2 * k6 / 55
30
31        error = np.abs(y5 - y4)
32
33        if error <= tol:
34            t += h
35            y = y4
36            t_valores.append(t)
37            y_valores.append(y)
38
39        if error != 0:
40            h = min(hmax, max(hmin, 0.84 * h * (tol / error) ** 0.25))
41        else:
42            h = hmax
43
44    return np.array(t_valores), np.array(y_valores)
45
46
47 # Definiciones para el Problema 3
48 def f_3a(t, y):
49     return y / t - (y / t) ** 2 if t != 0 else 0
50
51
52
```



```

47 def y_real_3a(t):
48     return t / (1 + np.log(t))
49
50
51 def f_3b(t, y):
52     return 1 + y / t + (y / t) ** 2 if t != 0 else 0
53
54
55 def y_real_3b(t):
56     return t * np.tan(np.log(t))
57
58
59 def f_3c(t, y):
60     return -(y + 1) * (y + 3)
61
62
63 def y_real_3c(t):
64     return -3 + 2 / (1 + np.exp(-2 * t))
65
66
67 def f_3d(t, y):
68     return np.sqrt(t + 2 * t ** 3) * y ** 3
69
70
71 def y_real_3d(t):
72     return (3 + 2 * t ** 2 + 6 * np.exp(t ** 2)) ** (-1 / 2)
73
74
75 problemas_3 = [
76     {"f": f_3a, "t0": 1, "y0": 1, "t_final": 4, "hmax": 0.5, "hmin": 0.05,
77      "tol": 1e-6, "y_real": y_real_3a,
78      "nombre": "3a)  $y' = y/t - (y/t)^2$ "},
79     {"f": f_3b, "t0": 1, "y0": 0, "t_final": 3, "hmax": 0.5, "hmin": 0.05,
80      "tol": 1e-6, "y_real": y_real_3b,
81      "nombre": "3b)  $y' = 1 + y/t + (y/t)^2$ "},
82     {"f": f_3c, "t0": 0, "y0": -2, "t_final": 3, "hmax": 0.5, "hmin":
83      0.05, "tol": 1e-6, "y_real": y_real_3c,
84      "nombre": "3c)  $y' = -(y + 1)(y + 3)$ "},
85     {"f": f_3d, "t0": 0, "y0": 1 / 3, "t_final": 2, "hmax": 0.5, "hmin":
86      0.05, "tol": 1e-6, "y_real": y_real_3d,
87      "nombre": "3d)  $y' = (t + 2t^3)^{(1/3)} y^3$ "}
88 ]
89
90 for problema in problemas_3:
91     t, y = rkf45(problema["f"], problema["t0"], problema["y0"], problema["
92     t_final"], problema["hmax"], problema["hmin"],
93     problema["tol"])
94     y_real = problema["y_real"](t)
95
96     print(f"\n--- {problema['nombre']} ---")
97     print("t\t y_aprox\t y_real\t\t Error")
98     for ti, yi, yri in zip(t, y, y_real):
99         print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")

```

```

96 plt.figure(figsize=(10, 6))
97 plt.plot(t, y, 'bo-', label='RKF45')
98 plt.plot(t, y_real, 'r-', label='Soluci n Real')
99 plt.title(problema["nombre"])
100 plt.xlabel("t")
101 plt.ylabel("y(t)")
102 plt.legend()
103 plt.grid()
104 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:

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

## 27 Implementación en Python

### 27.1 Función Principal RKF45

```

1 def
2     # Inicializacion
3

```

```

4
5
6
7 while
8     # Calculo de los 6 coeficientes k
9
10
11     # ... (k3 a k6)
12
13     # Estimaciones de orden 4 y 5
14
15
16
17     # Control de error
18         abs
19     if
20
21
22
23
24
25     # Ajuste adaptativo del paso
26         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)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def runge_kutta_4_system(f1, f2, t0, y0, z0, h, steps):
6     """Genera valores iniciales para AB4 usando RK4 en sistemas."""
7     t = np.zeros(steps + 1)
8     y = np.zeros(steps + 1)

```

```

9     z = np.zeros(steps + 1)
10    t[0], y[0], z[0] = t0, y0, z0
11
12    for i in range(steps):
13        k1_y = h * f1(t[i], y[i], z[i])
14        k1_z = h * f2(t[i], y[i], z[i])
15
16        k2_y = h * f1(t[i] + h / 2, y[i] + k1_y / 2, z[i] + k1_z / 2)
17        k2_z = h * f2(t[i] + h / 2, y[i] + k1_y / 2, z[i] + k1_z / 2)
18
19        k3_y = h * f1(t[i] + h / 2, y[i] + k2_y / 2, z[i] + k2_z / 2)
20        k3_z = h * f2(t[i] + h / 2, y[i] + k2_y / 2, z[i] + k2_z / 2)
21
22        k4_y = h * f1(t[i] + h, y[i] + k3_y, z[i] + k3_z)
23        k4_z = h * f2(t[i] + h, y[i] + k3_y, z[i] + k3_z)
24
25        y[i + 1] = y[i] + (k1_y + 2 * k2_y + 2 * k3_y + k4_y) / 6
26        z[i + 1] = z[i] + (k1_z + 2 * k2_z + 2 * k3_z + k4_z) / 6
27        t[i + 1] = t[i] + h
28
29    return t, y, z
30
31
32 def adams_bashforth_4_system(f1, f2, t, y, z, h):
33     """AB4 para sistemas de EDOs."""
34     for i in range(3, len(t) - 1):
35         # Predictor para y (posici n)
36         f_y = [f1(t[i - j], y[i - j], z[i - j]) for j in range(4)]
37         y[i + 1] = y[i] + h / 24 * (55 * f_y[0] - 59 * f_y[1] + 37 * f_y
38 [2] - 9 * f_y[3])
39
40         # Predictor para z (velocidad)
41         f_z = [f2(t[i - j], y[i - j], z[i - j]) for j in range(4)]
42         z[i + 1] = z[i] + h / 24 * (55 * f_z[0] - 59 * f_z[1] + 37 * f_z
43 [2] - 9 * f_z[3])
44
45         t[i + 1] = t[i] + h
46     return t, y, z
47
48 # Definici n de los sistemas para cada problema
49 # Problema a:  $y'' = |t|^2 - 2y$  Sistema:  $y' = z, z' = t^2 - 2y$ 
50 def f1_a(t, y, z):
51     return z
52
53 def f2_a(t, y, z):
54     return t ** 2 - 2 * y
55
56
57 def y_real_a(t):
58     return (1 / 3) * t ** 3 + (1 / 3) * np.exp(t)
59
60

```

```

61 # Problema b:  $y'' = 1 + (t - y)^2$       Sistema:  $y' = z, z' = 1 + (t - y)^2$ 
62 def f1_b(t, y, z):
63     return z
64
65
66 def f2_b(t, y, z):
67     return 1 + (t - y) ** 2
68
69
70 def y_real_b(t):
71     return t + 1 / (t - 1)
72
73
74 # Problema c:  $y'' = 1 + y/t$       Sistema:  $y' = z, z' = 1 + y/t$ 
75 def f1_c(t, y, z):
76     return z
77
78
79 def f2_c(t, y, z):
80     return 1 + y / t if t != 0 else 0
81
82
83 def y_real_c(t):
84     return t * np.log(t) + 2 * t
85
86
87 # Problema d:  $y'' = \cos(2t) + \sin(3t)$       Sistema:  $y' = z, z' = \cos(2t) + \sin(3t)$ 
88 def f1_d(t, y, z):
89     return z
90
91
92 def f2_d(t, y, z):
93     return np.cos(2 * t) + np.sin(3 * t)
94
95
96 def y_real_d(t):
97     return 0.5 * np.sin(2 * t) - (1 / 3) * np.cos(3 * t) + 4 / 3
98
99
100 # Configuraci3n de los problemas
101 problemas = [
102     {"f1": f1_a, "f2": f2_a, "t0": 0, "y0": 0, "z0": 1 / 3, "t_final": 1,
103      "h": 0.2, "y_real": y_real_a,
104      "nombre": "a)  $y'' = t^2 - 2y$ "},
105     {"f1": f1_b, "f2": f2_b, "t0": 2, "y0": 1, "z0": 1, "t_final": 3, "h":
106      0.2, "y_real": y_real_b,
107      "nombre": "b)  $y'' = 1 + (t - y)^2$ "},
108     {"f1": f1_c, "f2": f2_c, "t0": 1, "y0": 2, "z0": 1, "t_final": 2, "h":
109      0.2, "y_real": y_real_c,
110      "nombre": "c)  $y'' = 1 + y/t$ "},
111     {"f1": f1_d, "f2": f2_d, "t0": 0, "y0": 1, "z0": 0, "t_final": 1, "h":
112      0.2, "y_real": y_real_d,
113      "nombre": "d)  $y'' = \cos(2t) + \sin(3t)$ "}

```

```

110 ]
111
112 # Resolver cada problema
113 for problema in problemas:
114     # Generar valores iniciales con RK4
115     steps_rk4 = 3 # AB4 necesita 4 puntos iniciales
116     t_rk4, y_rk4, z_rk4 = runge_kutta_4_system(
117         problema["f1"], problema["f2"],
118         problema["t0"], problema["y0"], problema["z0"],
119         problema["h"], steps_rk4
120     )
121
122     # Extender arrays para AB4
123     total_steps = int((problema["t_final"] - problema["t0"]) / problema["h"]
124     "])
125     t = np.zeros(total_steps + 1)
126     y = np.zeros(total_steps + 1)
127     z = np.zeros(total_steps + 1)
128     t[:4] = t_rk4
129     y[:4] = y_rk4
130     z[:4] = z_rk4
131
132     # Aplicar AB4
133     t, y, z = adams_bashforth_4_system(problema["f1"], problema["f2"], t,
134     y, z, problema["h"])
135
136     # Calcular soluci n real y error
137     y_real = problema["y_real"](t)
138
139     # Resultados
140     print(f"\n--- {problema['nombre']} ---")
141     print("t\t y_aprox\t y_real\t\t Error")
142     for ti, yi, yri in zip(t, y, y_real):
143         print(f"{ti:.2f}\t {yi:.6f}\t {yri:.6f}\t {np.abs(yi - yri):.6f}")
144
145     # Gr fica
146     plt.figure(figsize=(10, 6))
147     plt.plot(t, y, 'bo-', label='AB4 Aproximaci n')
148     plt.plot(t, y_real, 'r-', label='Soluci n Real')
149     plt.title(f"Comparaci n AB4 vs Real: {problema['nombre']}")
150     plt.xlabel("t")
151     plt.ylabel("y(t)")
152     plt.legend()
153     plt.grid()
154     plt.show()

```

Listing 11: c

## Implementaci3n en Python

A continuaci3n se muestra el c3digo implementado:

```

1 import numpy as np

```



```
2 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.

```
1 def runge_kutta_4_system(f1, f2, t0, y0, z0, h, steps):
2     t = np.zeros(steps + 1)
3     y = np.zeros(steps + 1)
4     z = np.zeros(steps + 1)
5     t[0], y[0], z[0] = t0, y0, z0
6
7     for i in range(steps):
8         k1_y = h * f1(t[i], y[i], z[i])
9         k1_z = h * f2(t[i], y[i], z[i])
10
11         k2_y = h * f1(t[i] + h/2, y[i] + k1_y/2, z[i] + k1_z/2)
12         k2_z = h * f2(t[i] + h/2, y[i] + k1_y/2, z[i] + k1_z/2)
13
14         k3_y = h * f1(t[i] + h/2, y[i] + k2_y/2, z[i] + k2_z/2)
15         k3_z = h * f2(t[i] + h/2, y[i] + k2_y/2, z[i] + k2_z/2)
16
17         k4_y = h * f1(t[i] + h, y[i] + k3_y, z[i] + k3_z)
18         k4_z = h * f2(t[i] + h, y[i] + k3_y, z[i] + k3_z)
19
20         y[i+1] = y[i] + (k1_y + 2*k2_y + 2*k3_y + k4_y) / 6
21         z[i+1] = z[i] + (k1_z + 2*k2_z + 2*k3_z + k4_z) / 6
22         t[i+1] = t[i] + h
23
24     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.

```
1 def adams_bashforth_4_system(f1, f2, t, y, z, h):
2     for i in range(3, len(t) - 1):
3         f_y = [f1(t[i-j], y[i-j], z[i-j]) for j in range(4)]
4         y[i+1] = y[i] + h / 24 * (55*f_y[0] - 59*f_y[1] + 37*f_y[2] - 9*
5         f_y[3])
6
7         f_z = [f2(t[i-j], y[i-j], z[i-j]) for j in range(4)]
8         z[i+1] = z[i] + h / 24 * (55*f_z[0] - 59*f_z[1] + 37*f_z[2] - 9*
9         f_z[3])
10
11         t[i+1] = t[i] + h
12
13     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.

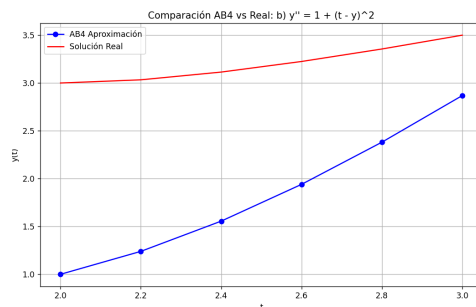
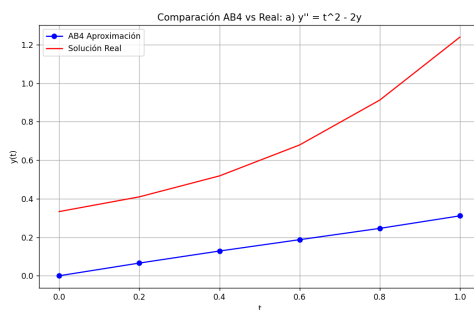
```
1 # Ejemplo problema a:  $y'' = t^2 - 2y$ 
2 def f1_a(t, y, z): return z
3 def f2_a(t, y, z): return  $t^2 - 2y$ 
4 def y_real_a(t): return  $(1/3)t^3 + (1/3)\exp(t)$ 
```

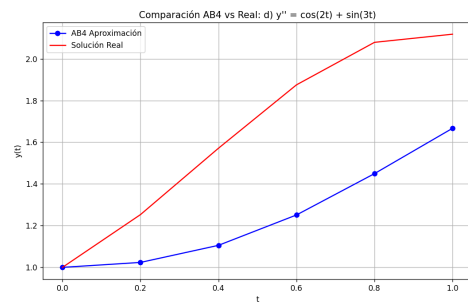
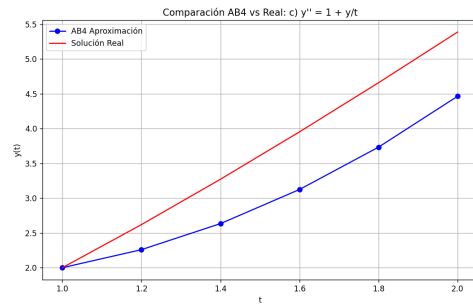
*Nota: Se definen funciones similares para los problemas b, c y d.*

## Ejecución y gráficos

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

```
1 for problema in problemas:
2     t_rk4, y_rk4, z_rk4 = runge_kutta_4_system(...)
3     t[:4], y[:4], z[:4] = t_rk4, y_rk4, z_rk4
4     t, y, z = adams_bashforth_4_system(...)
5     y_real = problema["y_real"](t)
6
7     # Imprimir resultados y graficar
8     for ti, yi, yri in zip(t, y, y_real):
9         print(ti, yi, yri, abs(yi - yri))
10
11     plt.plot(t, y, 'bo-', label='AB4')
12     plt.plot(t, y_real, 'r-', label='Exacta')
13     plt.legend()
14     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)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def runge_kutta_4_system(functions, t0, initial_conditions, h, steps):
6     """
7     M todo de Runge-Kutta de 4to orden para sistemas de EDOs.
8
9     Par metros:
10    - functions: Lista de funciones [f1, f2, ..., fn] que definen las EDOs
11    - t0: Tiempo inicial.
12    - initial_conditions: Lista de valores iniciales [u1_0, u2_0, ...,
13      un_0].
14    - h: Tama o del paso.
15    - steps: N mero de pasos.
16
17    Retorna:
18    - t: Array de tiempos.
19    - solutions: Matriz de soluciones (cada fila corresponde a una
20      variable).
21    """
22    n = len(functions)
23    t = np.zeros(steps + 1)
24    solutions = np.zeros((n, steps + 1))
25    t[0] = t0
26    solutions[:, 0] = initial_conditions
27
28    for i in range(steps):
29        k1 = np.zeros(n)
30        k2 = np.zeros(n)
31        k3 = np.zeros(n)
32        k4 = np.zeros(n)
33
34        # C lculo de k1
35        args = [t[i]] + list(solutions[:, i])

```

```

34     for j in range(n):
35         k1[j] = h * functions[j](args)
36
37     # C lculo de k2
38     args_k2 = [t[i] + h / 2] + list(solutions[:, i] + k1 / 2)
39     for j in range(n):
40         k2[j] = h * functions[j](args_k2)
41
42     # C lculo de k3
43     args_k3 = [t[i] + h / 2] + list(solutions[:, i] + k2 / 2)
44     for j in range(n):
45         k3[j] = h * functions[j](args_k3)
46
47     # C lculo de k4
48     args_k4 = [t[i] + h] + list(solutions[:, i] + k3)
49     for j in range(n):
50         k4[j] = h * functions[j](args_k4)
51
52     # Actualizaci n
53     solutions[:, i + 1] = solutions[:, i] + (k1 + 2 * k2 + 2 * k3 + k4
54 ) / 6
55     t[i + 1] = t[i] + h
56
57     return t, solutions
58
59 # =====
60 # Problema (a)
61 # =====
62 def f1_a(t, u1, u2):
63     return 3 * u1 + 2 * u2 - (2 * t ** 2 + 1) * np.exp(2 * t)
64
65
66 def f2_a(t, u1, u2):
67     return 4 * u1 + u2 + (t ** 2 + 2 * t - 4) * np.exp(2 * t)
68
69
70 def u1_real_a(t):
71     return (1 / 2) * np.exp(t ** 2) - (1 / 2) * np.exp(-t) + np.exp(2 * t)
72
73
74 def u2_real_a(t):
75     return (1 / 2) * np.exp(t ** 2) + (3 / 2) * np.exp(-t) + t ** 2 * np.
76         exp(2 * t)
77
78 # Configuraci n
79 t0_a = 0
80 u0_a = [1, 1] # u1(0) = 1, u2(0) = 1
81 t_final_a = 1
82 h_a = 0.2
83 steps_a = int((t_final_a - t0_a) / h_a)
84
85 # Soluci n

```

```

86 t_a, sol_a = runge_kutta_4_system([f1_a, f2_a], t0_a, u0_a, h_a, steps_a)
87 u1_real_a_vals = u1_real_a(t_a)
88 u2_real_a_vals = u2_real_a(t_a)
89
90 # Gr ficas
91 plt.figure(figsize=(12, 5))
92 plt.subplot(1, 2, 1)
93 plt.plot(t_a, sol_a[0], 'bo-', label='RK4 $u_1(t)$')
94 plt.plot(t_a, u1_real_a_vals, 'r-', label='Real $u_1(t)$')
95 plt.title("Problema (a): $u_1(t)$")
96 plt.xlabel("t")
97 plt.legend()
98 plt.grid()
99
100 plt.subplot(1, 2, 2)
101 plt.plot(t_a, sol_a[1], 'bo-', label='RK4 $u_2(t)$')
102 plt.plot(t_a, u2_real_a_vals, 'r-', label='Real $u_2(t)$')
103 plt.title("Problema (a): $u_2(t)$")
104 plt.xlabel("t")
105 plt.legend()
106 plt.grid()
107 plt.tight_layout()
108 plt.show()
109
110
111 # =====
112 # Problema (b)
113 # =====
114 def f1_b(t, u1, u2):
115     return -4 * u1 - 2 * u2 + np.cos(t) + 4 * np.sin(t)
116
117
118 def f2_b(t, u1, u2):
119     return 3 * u1 + u2 - 3 * np.sin(t)
120
121
122 def u1_real_b(t):
123     return 2 * np.exp(-t) - 2 * np.exp(-2 * t) + np.sin(t)
124
125
126 def u2_real_b(t):
127     return -3 * np.exp(-t) + 2 * np.exp(-2 * t)
128
129
130 # Configuraci n
131 t0_b = 0
132 u0_b = [0, -1] # u1(0) = 0, u2(0) = -1
133 t_final_b = 2
134 h_b = 0.1
135 steps_b = int((t_final_b - t0_b) / h_b)
136
137 # Soluci n
138 t_b, sol_b = runge_kutta_4_system([f1_b, f2_b], t0_b, u0_b, h_b, steps_b)
139 u1_real_b_vals = u1_real_b(t_b)

```

```

140 u2_real_b_vals = u2_real_b(t_b)
141
142 # Gráficas
143 plt.figure(figsize=(12, 5))
144 plt.subplot(1, 2, 1)
145 plt.plot(t_b, sol_b[0], 'bo-', label='RK4 $u_1(t)$')
146 plt.plot(t_b, u1_real_b_vals, 'r-', label='Real $u_1(t)$')
147 plt.title("Problema (b): $u_1(t)$")
148 plt.xlabel("t")
149 plt.legend()
150 plt.grid()
151
152 plt.subplot(1, 2, 2)
153 plt.plot(t_b, sol_b[1], 'bo-', label='RK4 $u_2(t)$')
154 plt.plot(t_b, u2_real_b_vals, 'r-', label='Real $u_2(t)$')
155 plt.title("Problema (b): $u_2(t)$")
156 plt.xlabel("t")
157 plt.legend()
158 plt.grid()
159 plt.tight_layout()
160 plt.show()
161
162
163 # =====
164 # Problema (c) - Sistema de 3 EDOs
165 # =====
166 def f1_c(t, u1, u2, u3):
167     return u2
168
169
170 def f2_c(t, u1, u2, u3):
171     return -u1 - 2 * np.exp(t + 1)
172
173
174 def f3_c(t, u1, u2, u3):
175     return -u1 - np.exp(t + 1)
176
177
178 def u1_real_c(t):
179     return np.cos(t) + np.sin(t) - np.exp(t + 1)
180
181
182 def u2_real_c(t):
183     return -np.sin(t) + np.cos(t) - np.exp(t)
184
185
186 def u3_real_c(t):
187     return -np.sin(t) + np.cos(t)
188
189
190 # Configuración
191 t0_c = 0
192 u0_c = [1, 0, 1] # u1(0)=1, u2(0)=0, u3(0)=1
193 t_final_c = 2

```

```

194 h_c = 0.5
195 steps_c = int((t_final_c - t0_c) / h_c)
196
197 # Soluci n
198 t_c, sol_c = runge_kutta_4_system([f1_c, f2_c, f3_c], t0_c, u0_c, h_c,
    steps_c)
199 u1_real_c_vals = u1_real_c(t_c)
200 u2_real_c_vals = u2_real_c(t_c)
201 u3_real_c_vals = u3_real_c(t_c)
202
203 # Gr ficas
204 plt.figure(figsize=(15, 5))
205 for i in range(3):
206     plt.subplot(1, 3, i + 1)
207     plt.plot(t_c, sol_c[i], 'bo-', label=f'RK4 $u_{i + 1}(t)$')
208     plt.plot(t_c, [u1_real_c_vals, u2_real_c_vals, u3_real_c_vals][i], 'r-
    ', label=f'Real $u_{i + 1}(t)$')
209     plt.title(f"Problema (c): $u_{i + 1}(t)$")
210     plt.xlabel("t")
211     plt.legend()
212     plt.grid()
213 plt.tight_layout()
214 plt.show()
215
216
217 # =====
218 # Problema (d) - Sistema de 3 ED0s
219 # =====
220 def f1_d(t, u1, u2, u3):
221     return u2 - u3 + t
222
223
224 def f2_d(t, u1, u2, u3):
225     return 3 * t ** 2
226
227
228 def f3_d(t, u1, u2, u3):
229     return u2 + np.exp(-t)
230
231
232 def u1_real_d(t):
233     return -0.05 * np.exp(t) + 0.25 * t ** 3 + t + 2 - np.exp(-t)
234
235
236 def u2_real_d(t):
237     return t ** 3 + 1
238
239
240 def u3_real_d(t):
241     return 0.25 * t ** 3 + t - np.exp(-t)
242
243
244 # Configuraci n
245 t0_d = 0

```



```

246 u0_d = [1, 1, -1] # u1(0)=1, u2(0)=1, u3(0)=-1
247 t_final_d = 1
248 h_d = 0.1
249 steps_d = int((t_final_d - t0_d) / h_d)
250
251 # Soluci n
252 t_d, sol_d = runge_kutta_4_system([f1_d, f2_d, f3_d], t0_d, u0_d, h_d,
    steps_d)
253 u1_real_d_vals = u1_real_d(t_d)
254 u2_real_d_vals = u2_real_d(t_d)
255 u3_real_d_vals = u3_real_d(t_d)
256
257 # Gr ficas
258 plt.figure(figsize=(15, 5))
259 for i in range(3):
260     plt.subplot(1, 3, i + 1)
261     plt.plot(t_d, sol_d[i], 'bo-', label=f'RK4 $u_{i + 1}(t)$')
262     plt.plot(t_d, [u1_real_d_vals, u2_real_d_vals, u3_real_d_vals][i], 'r-
    ', label=f'Real $u_{i + 1}(t)$')
263     plt.title(f"Problema (d): $u_{i + 1}(t)$")
264     plt.xlabel("t")
265     plt.legend()
266     plt.grid()
267 plt.tight_layout()
268 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:

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

## 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:

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

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

### 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:**

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

#### 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:**

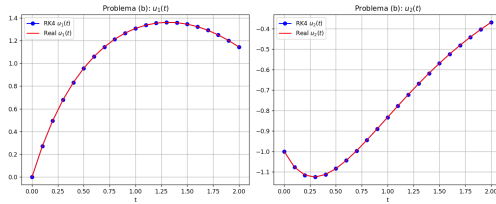
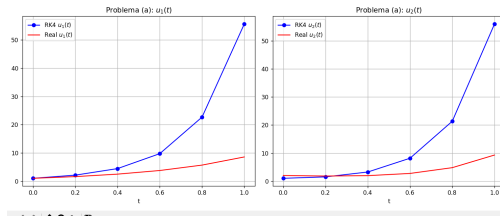
$$u_1(t) = -0.05e^t + 0.25t^3 + t + 2 - e^{-t}$$

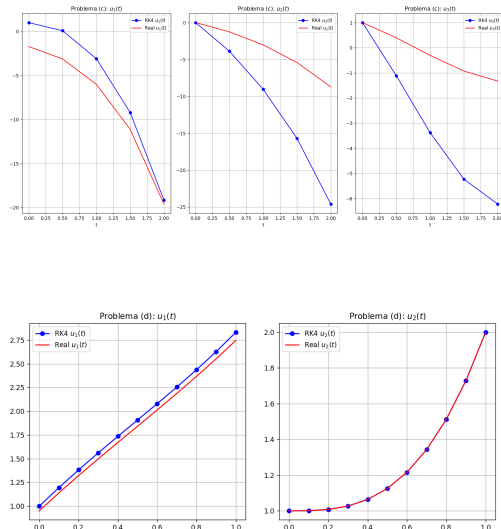
$$u_2(t) = t^3 + 1$$

$$u_3(t) = 0.25t^3 + t - e^{-t}$$

## 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)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def runge_kutta_4_system(f1, f2, t0, x10, x20, h, steps, k1, k2, k3, k4):
6     """
7     M todo de Runge-Kutta de 4to orden para el sistema Lotka-Volterra.
8
9     Par metros:
10    - f1, f2: Funciones que definen las EDOs (dx1/dt y dx2/dt).
11    - t0: Tiempo inicial.
12    - x10, x20: Poblaciones iniciales de presas y depredadores.
13    - h: Tama o del paso.
14    - steps: N mero de pasos.
15    - k1, k2, k3, k4: Constantes del modelo.
16
17    Retorna:
18    - t: Array de tiempos.
19    - x1, x2: Arrays de poblaciones de presas y depredadores.
20    """
21    t = np.zeros(steps + 1)
22    x1 = np.zeros(steps + 1)
23    x2 = np.zeros(steps + 1)
24    t[0], x1[0], x2[0] = t0, x10, x20
25
26    for i in range(steps):
27        # Coeficientes k para x1 (presas)
28        k1_x1 = h * f1(t[i], x1[i], x2[i], k1, k2)
29        k1_x2 = h * f2(t[i], x1[i], x2[i], k3, k4)

```

```

30
31     k2_x1 = h * f1(t[i] + h / 2, x1[i] + k1_x1 / 2, x2[i] + k1_x2 / 2,
k1, k2)
32     k2_x2 = h * f2(t[i] + h / 2, x1[i] + k1_x1 / 2, x2[i] + k1_x2 / 2,
k3, k4)
33
34     k3_x1 = h * f1(t[i] + h / 2, x1[i] + k2_x1 / 2, x2[i] + k2_x2 / 2,
k1, k2)
35     k3_x2 = h * f2(t[i] + h / 2, x1[i] + k2_x1 / 2, x2[i] + k2_x2 / 2,
k3, k4)
36
37     k4_x1 = h * f1(t[i] + h, x1[i] + k3_x1, x2[i] + k3_x2, k1, k2)
38     k4_x2 = h * f2(t[i] + h, x1[i] + k3_x1, x2[i] + k3_x2, k3, k4)
39
40     # Actualizaci n
41     x1[i + 1] = x1[i] + (k1_x1 + 2 * k2_x1 + 2 * k3_x1 + k4_x1) / 6
42     x2[i + 1] = x2[i] + (k1_x2 + 2 * k2_x2 + 2 * k3_x2 + k4_x2) / 6
43     t[i + 1] = t[i] + h
44
45     return t, x1, x2
46
47
48 # Definici n de las EDOs del modelo Lotka-Volterra
49 def f1_presas(t, x1, x2, k1, k2):
50     return k1 * x1 - k2 * x1 * x2 # dx1/dt = k1*x1 - k2*x1*x2
51
52
53 def f2_depredadores(t, x1, x2, k3, k4):
54     return k3 * x1 * x2 - k4 * x2 # dx2/dt = k3*x1*x2 - k4*x2
55
56
57 # Par metros del modelo (ejemplo)
58 k1 = 0.4 # Tasa de natalidad de presas
59 k2 = 0.01 # Tasa de mortalidad de presas por depredadores
60 k3 = 0.001 # Tasa de natalidad de depredadores por presas
61 k4 = 0.3 # Tasa de mortalidad de depredadores
62
63 # Configuraci n de la simulaci n
64 t0 = 0
65 x10 = 50 # Poblaci n inicial de presas
66 x20 = 20 # Poblaci n inicial de depredadores
67 t_final = 100
68 h = 0.1
69 steps = int((t_final - t0) / h)
70
71 # Resolver el sistema
72 t, x1, x2 = runge_kutta_4_system(f1_presas, f2_depredadores, t0, x10, x20,
h, steps, k1, k2, k3, k4)
73
74 # Gr ficas
75 plt.figure(figsize=(12, 6))
76
77 # Poblaciones vs tiempo
78 plt.subplot(1, 2, 1)

```

```

79 plt.plot(t, x1, 'g-', label='Presas ($x_1$)')
80 plt.plot(t, x2, 'r-', label='Depredadores ($x_2$)')
81 plt.title('Dinámica de Poblaciones (Lotka-Volterra)')
82 plt.xlabel('Tiempo ($t$)')
83 plt.ylabel('Población')
84 plt.legend()
85 plt.grid()
86
87 # Diagrama de fase
88 plt.subplot(1, 2, 2)
89 plt.plot(x1, x2, 'b-')
90 plt.title('Diagrama de Fase ($x_1$ vs $x_2$)')
91 plt.xlabel('Presas ($x_1$)')
92 plt.ylabel('Depredadores ($x_2$)')
93 plt.grid()
94
95 plt.tight_layout()
96 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:

$$\begin{aligned}\frac{dx_1}{dt} &= k_1x_1 - k_2x_1x_2 && \text{(presas)} \\ \frac{dx_2}{dt} &= k_3x_1x_2 - k_4x_2 && \text{(depredadores)}\end{aligned}$$

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:

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

### 3. Implementación en Python

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

```

1 def f1_presas(t, x1, x2, k1, k2):
2     return k1 * x1 - k2 * x1 * x2
3
4 def f2_depredadores(t, x1, x2, k3, k4):
5     return k3 * x1 * x2 - k4 * x2
6
7 def runge_kutta_4_system(f1, f2, t0, x10, x20, h, steps, k1, k2, k3, k4):
8     ...

```

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.

