Task3 - Método de Euler

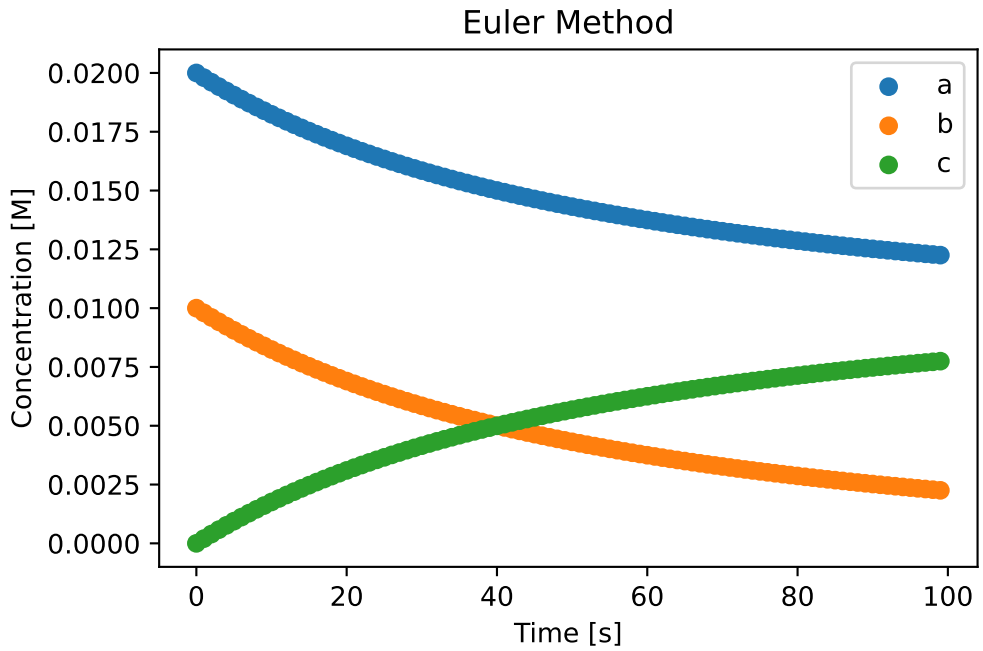
Redes Biológicas y Biología de Sistemas

Sandra Mingo Ramírez

Parte 1: Método de Euler para $a + b \xrightarrow{k} c$

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

def euler(initial_conditions, constant, timespan):

    U = np.zeros((len(initial_conditions), timespan))
    U[:, 0] = initial_conditions

    for i in range(1, timespan):
        U[0, i] = U[0, i-1] - constant * U[0, i-1] * U[1, i-1]
        U[1, i] = U[1, i-1] - constant * U[0, i-1] * U[1, i-1]
        U[2, i] = U[2, i-1] + constant * U[0, i-1] * U[1, i-1]

    return U

u_0 = [0.02, 0.01, 0.0]
k = 1
timespan = 100 #seconds

time_evolution = euler(u_0, k, timespan)


fig = plt.figure()
plt.scatter(range(timespan), time_evolution[0, :], label='a')
plt.scatter(range(timespan), time_evolution[1, :], label='b')
plt.scatter(range(timespan), time_evolution[2, :], label='c')
plt.xlabel("Time [s]")
```

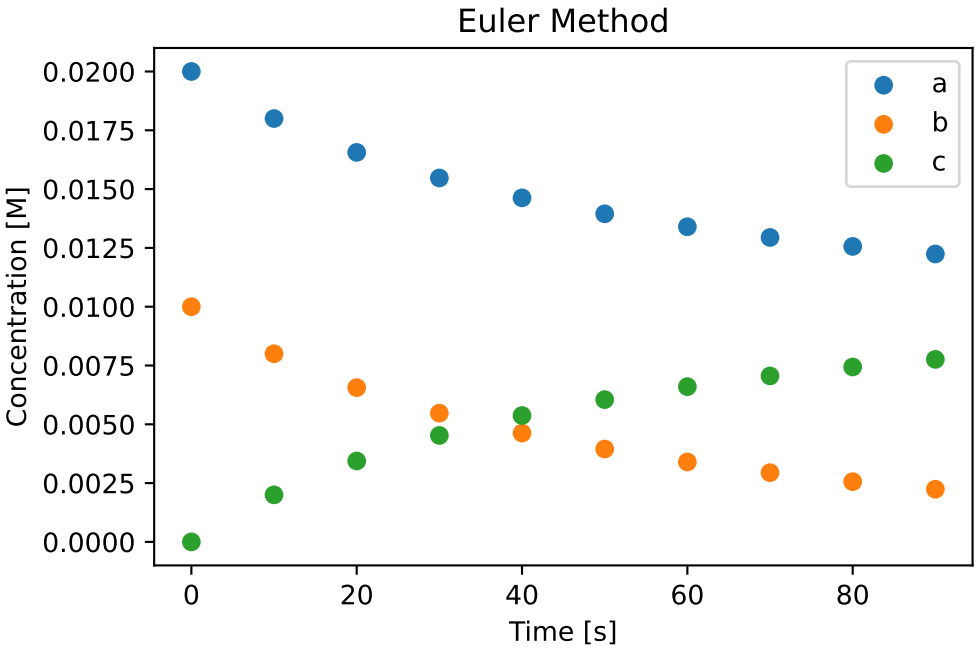
```
plt.ylabel("Concentration [M]")
plt.title("Euler Method")
plt.legend()
plt.show()
```



Incluir tiempo de integración. Medimos durante la misma cantidad de tiempo total, pero queremos medir las concentraciones con distinta frecuencia (cada segundo, cada dos segundos, etc.).

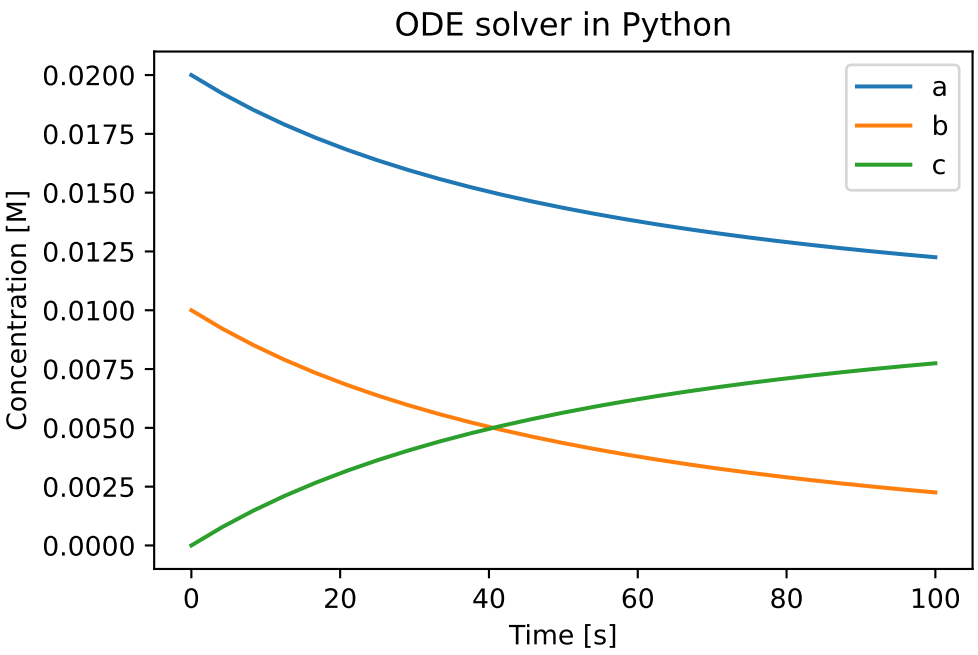
```
import numpy as np
import matplotlib.pyplot as plt

def euler(initial_conditions, constant, timespan):

    U = np.zeros((len(initial_conditions), timespan))
    U[:, 0] = initial_conditions

    for i in range(1, timespan):
        U[0, i] = U[0, i-1] - timespan * constant * U[0, i-1] * U[1, i-1]
        U[1, i] = U[1, i-1] - timespan * constant * U[0, i-1] * U[1, i-1]
        U[2, i] = U[2, i-1] + timespan * constant * U[0, i-1] * U[1, i-1]
```

```
    return U

u_0 = [0.02, 0.01, 0.0]
k = 1
timestep = 10
timespan = int(100/timestep)

time_evolution = euler(u_0, k, timespan)


fig = plt.figure()
plt.scatter(np.arange(timespan) * timestep, time_evolution[0, :], label='a')
plt.scatter(np.arange(timespan) * timestep, time_evolution[1, :], label='b')
plt.scatter(np.arange(timespan) * timestep, time_evolution[2, :], label='c')
plt.xlabel("Time [s]")
plt.ylabel("Concentration [M]")
plt.title("Euler Method")
plt.legend()
plt.show()
```



Ahora, en lugar de utilizar la función de Euler, utilizaremos ODEINT:

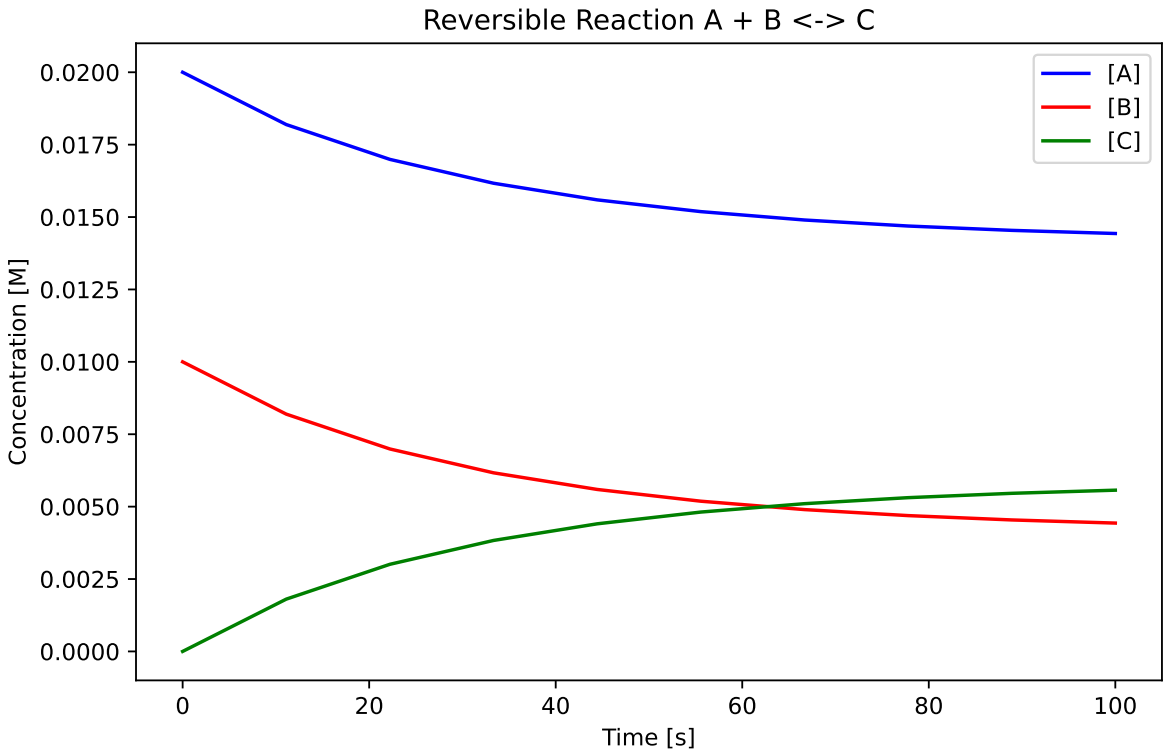
```
from scipy.integrate import odeint
import numpy as np

# Definir el sistema de ecuaciones diferenciales
def differential_equations(y, t, k):
    A, B, C = y  # Concentraciones de A, B y C
    dA_dt = -k * A * B  # Ecuación para A
    dB_dt = -k * A * B  # Ecuación para B
    dC_dt = k * A * B   # Ecuación para C
    return [dA_dt, dB_dt, dC_dt]

# Parámetros
k = 1  # Constante de reacción
A0 = 0.02  # Concentración inicial de A
B0 = 0.01 # Concentración inicial de B
C0 = 0.0  # Concentración inicial de C

# Condiciones iniciales (A0, B0, C0)
y0 = [A0, B0, C0]

# Tiempo de integración
t = np.linspace(0, 100, 25)

# Resolver las ecuaciones diferenciales
ode_solved = odeint(differential_equations, y0, t, args=(k,))

# Graficar los resultados
plt.plot(t, ode_solved[:, 0], label='a')
plt.plot(t, ode_solved[:, 1], label='b')
plt.plot(t, ode_solved[:, 2], label='c')

plt.xlabel('Time [s]')
plt.ylabel('Concentration [M]')
plt.title('ODE solver in Python')
plt.legend()
plt.show()
```



Parte 2: Método de Euler para ecuación general $a + b \underset{k_2}{\overset{k_1}{\rightleftharpoons}} c$

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

# Definir el sistema de ecuaciones diferenciales
def differential_equations(y, t, k1, k2):
    A, B, C = y  # Concentraciones de A, B y C
    dA_dt = -k1 * A * B + k2 * C  # Ecuación para A
    dB_dt = -k1 * A * B + k2 * C  # Ecuación para B
    dC_dt = k1 * A * B - k2 * C   # Ecuación para C
    return [dA_dt, dB_dt, dC_dt]

# Parámetros
k1 = 1   # Constante de reacción directa
k2 = 0.01 # Constante de reacción inversa

# Condiciones iniciales
```

```
A0 = 0.02
B0 = 0.01
C0 = 0.0
y0 = [A0, B0, C0]

# Tiempo de integración
t = np.linspace(0, 100, 10)

# Resolver las ecuaciones diferenciales
ode_solved = odeint(differential_equations, y0, t, args=(k1, k2))

# Graficar los resultados
plt.figure(figsize=(8, 5))
plt.plot(t, ode_solved[:, 0], label='[A]', color='blue')
plt.plot(t, ode_solved[:, 1], label='[B]', color='red')
plt.plot(t, ode_solved[:, 2], label='[C]', color='green')

plt.xlabel('Time [s]')
plt.ylabel('Concentration [M]')
plt.title('Reversible Reaction A + B <-> C')
plt.legend()
plt.show()
```

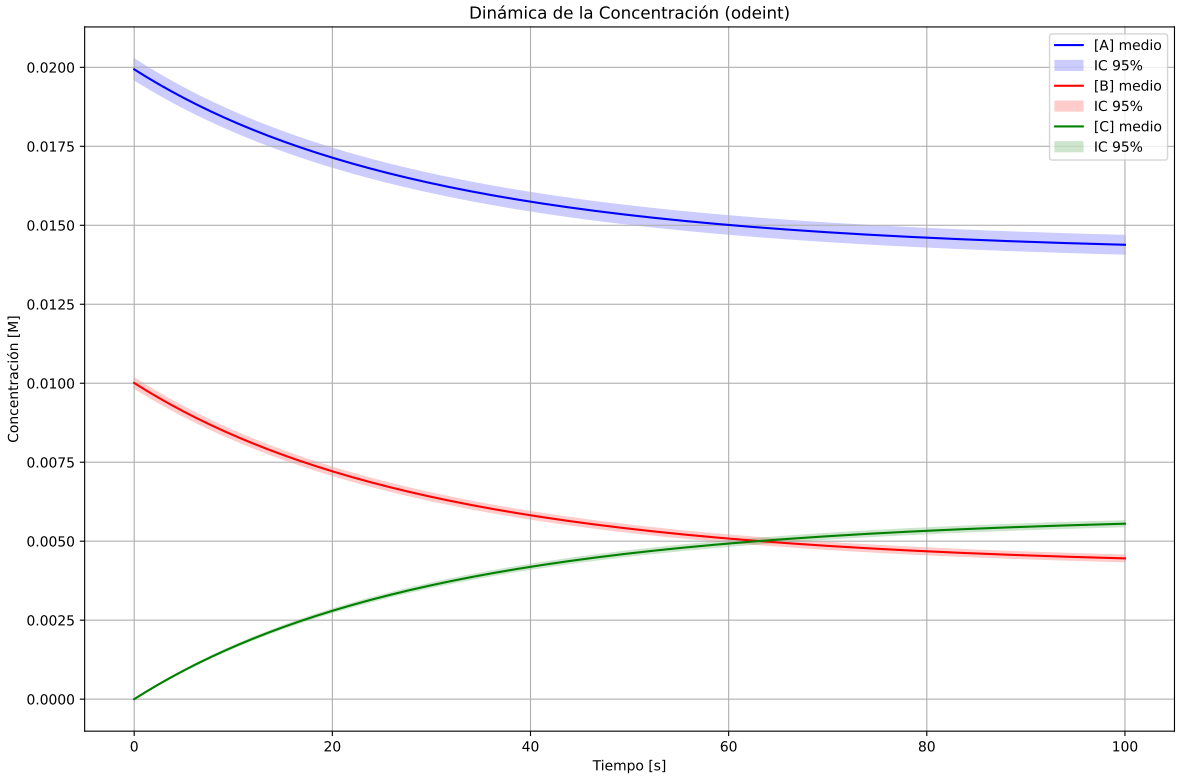


Para la segunda parte hay que introducir un 10% de variabilidad en las condiciones iniciales y realizar 50 simulaciones para calcular las dinámicas medias y calcular el intervalo de confianza del 95%.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.stats import sem, t

# Parámetros de la reacción
k1 = 1.0    # Constante de velocidad directa
k2 = 0.01   # Constante de velocidad inversa

# Condiciones iniciales base
A0_base = 0.02
B0_base = 0.01
C0_base = 0.0

# Configuración de la simulación
t_max = 100          # Tiempo máximo
n_points = 1000      # Número de puntos de tiempo
```

```
n_simulations = 50   # Número de simulaciones
variability = 0.1    # 10% de variabilidad

# Definir el sistema de ecuaciones diferenciales
def reaction_system(y, t, k1, k2):
    A, B, C = y
    dA_dt = -k1 * A * B + k2 * C
    dB_dt = -k1 * A * B + k2 * C
    dC_dt = k1 * A * B - k2 * C
    return [dA_dt, dB_dt, dC_dt]

# Tiempo de integración
t_points = np.linspace(0, t_max, n_points)

# Almacenamiento de resultados
all_A = np.zeros((n_simulations, n_points))
all_B = np.zeros((n_simulations, n_points))
all_C = np.zeros((n_simulations, n_points))

# Realizar las 50 simulaciones
for sim in range(n_simulations):
    # Añadir variabilidad del 10% a las condiciones iniciales
    A0 = A0_base * np.random.uniform(1 - variability, 1 + variability)
    B0 = B0_base * np.random.uniform(1 - variability, 1 + variability)
    C0 = C0_base * np.random.uniform(1 - variability, 1 + variability)

    # Resolver con odeint
    solution = odeint(reaction_system, [A0, B0, C0], t_points, args=(k1, k2))

    all_A[sim] = solution[:, 0]
    all_B[sim] = solution[:, 1]
    all_C[sim] = solution[:, 2]

# Calcular estadísticas
mean_A = np.mean(all_A, axis=0)
mean_B = np.mean(all_B, axis=0)
mean_C = np.mean(all_C, axis=0)

# Calcular intervalos de confianza del 95%
def confidence_interval(data, confidence=0.95):
    n = len(data)
    m = np.mean(data, axis=0)
```

```
    std_err = sem(data, axis=0)
    h = std_err * t.ppf((1 + confidence) / 2, n - 1)
    return m - h, m + h

ci_A_low, ci_A_high = confidence_interval(all_A)
ci_B_low, ci_B_high = confidence_interval(all_B)
ci_C_low, ci_C_high = confidence_interval(all_C)

# Graficar resultados
plt.figure(figsize=(12, 8))

# Concentración de A
plt.plot(t_points, mean_A, label='[A] medio', color='blue')
plt.fill_between(t_points, ci_A_low, ci_A_high, color='blue', alpha=0.2, label='IC 95%')

# Concentración de B
plt.plot(t_points, mean_B, label='[B] medio', color='red')
plt.fill_between(t_points, ci_B_low, ci_B_high, color='red', alpha=0.2, label='IC 95%')

# Concentración de C
plt.plot(t_points, mean_C, label='[C] medio', color='green')
plt.fill_between(t_points, ci_C_low, ci_C_high, color='green', alpha=0.2, label='IC 95%')

plt.xlabel('Tiempo [s]')
plt.ylabel('Concentración [M]')
plt.title('Dinámica de la Concentración (odeint)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



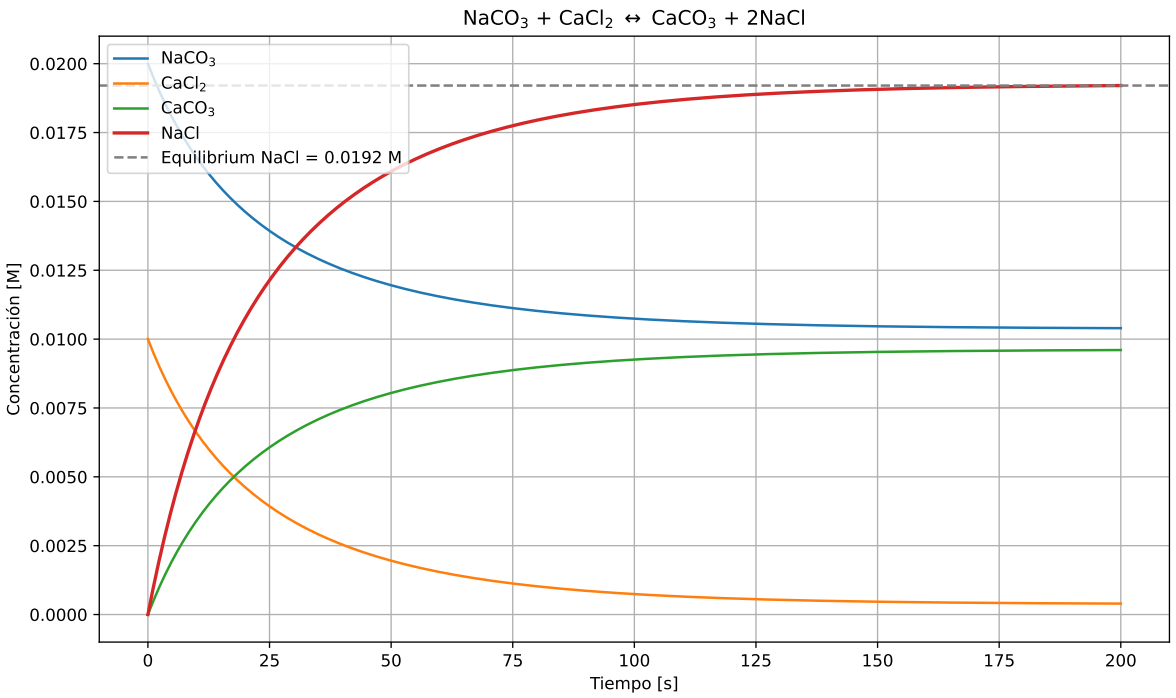
Parte 3: Método de Euler para ecuación de la sal

$$NaCO_3 + CaCl_2 \underset{k_2}{\overset{k_1}{\rightleftharpoons}} CaCO_3 + 2NaCl$$

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

# Definir el sistema de ecuaciones diferenciales
def differential_equations(y, t, k1, k2):
    A, B, C, D = y  # A: NaCO3, B: CaCl2, C: CaCO3, D: NaCl
    dA_dt = -k1 * A * B + k2 * C * D**2
    dB_dt = -k1 * A * B + k2 * C * D**2
    dC_dt = k1 * A * B - k2 * C * D**2
    dD_dt = 2 * (k1 * A * B - k2 * C * D**2)
    return [dA_dt, dB_dt, dC_dt, dD_dt]
```

```
# Parámetros de reacción
k_1 = 2.3  # Constante de velocidad directa
k_2 = 2.5  # Constante de velocidad indirecta

# Concentraciones iniciales
a_0 = 0.02  # NaCO3
b_0 = 0.01  # CaCl2
c_0 = 0.0   # CaCO3
d_0 = 0.0   # NaCl
y0 = [a_0, b_0, c_0, d_0]

# Tiempo de integración
simulation_time = 200  # Tiempo de reacción
time_points = 1000      # Número de mediciones, puntos en el tiempo

t = np.linspace(0, simulation_time, time_points)

# Resolver ecuaciones diferenciales
solution = odeint(differential_equations, y0, t, args=(k_1, k_2))

# Concentración en equilibrio (último valor en el array)
equilibrium_NaCl = solution[-1, 3]

# Graficar resultados
plt.figure(figsize=(10, 6))
plt.plot(t, solution[:, 0], label='NaCO$_3$')
plt.plot(t, solution[:, 1], label='CaCl$_2$')
plt.plot(t, solution[:, 2], label='CaCO$_3$')
plt.plot(t, solution[:, 3], label='NaCl', linewidth=2)

# Añadir línea del equilibrio de NaCl
plt.axhline(y=equilibrium_NaCl, color='gray', linestyle='--',
            label=f'Equilibrium NaCl = {equilibrium_NaCl:.4f} M')

plt.xlabel('Tiempo [s]')
plt.ylabel('Concentración [M]')
plt.title(f'NaCO$_3$ + CaCl$_2$ $\leftrightarrow$ CaCO$_3$ + 2NaCl')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
print(f"Concentración final de NaCl en {simulation_time} s: {equilibrium_NaCl:.4f} M")
```




```
Concentración final de NaCl en 200 s: 0.0192 M
```