

▼ Práctica 2 - Parte 1: Modelo poblacional

En este cuaderno se presenta un **ejemplo de modelo poblacional** basado en el modelo logístico clásico. El objetivo es analizar el comportamiento de una población que crece bajo recursos limitados (capacidad de carga).

Enunciado

1. Considera el modelo logístico de crecimiento poblacional:

$$\dot{N} = rN \left(1 - \frac{N}{K}\right)$$

donde:

- $N(t)$ es la población en el tiempo,
- r es la tasa de crecimiento,
- K es la capacidad de carga del entorno.

2. Añade al modelo una señal de control $u(t)$ que represente una **cosecha constante** (extracción de individuos):

$$\dot{N} = rN \left(1 - \frac{N}{K}\right) - u(t)$$

3. Analiza el comportamiento del sistema para tres casos de $u(t)$:

- Caso $u = 0$ (sin cosecha).
- Caso $0 < u < rK/4$ (cosecha moderada).
- Caso $u > rK/4$ (cosecha excesiva).

4. Representa gráficamente la evolución temporal de la población en cada caso y comenta las diferencias observadas.
-

Ejercicio propuesto

Implementa en Python el modelo logístico con cosecha y representa la evolución temporal $N(t)$ para los tres casos de $u(t)$. Interpreta los resultados y discute qué ocurre con la población en cada situación.

EXPLICACIÓN DEL CÓDIGO

El código adjuntado bajo resuelve numéricamente el modelo logístico con cosecha constante para tres valores de tasa de extracción u y dibuja las tres trayectorias $N(t)$ en una misma gráfica.

Primeramente se definen los parámetros importantes y constantes (tasa r , capacidad K y condición inicial N_0) y usa `solve_ivp` para integrar la ecuación $\dot{N} = rN(1-N/K) - u$ en cada caso de u . La función `modelo_logistico` calcula la derivada y pretende evitar valores negativos forzando la salida con `mp.maximum` antes de imprimir la gráfica. Esto permite ver claramente cuándo la población alcanza un equilibrio, se mantiene reducida o se extingue segunda la intensidad de la cosecha. Tener en cuenta que, por la robustez numérica es preferible tener un evento de extinción que detenga la integración al alcanzar $N = 0$, en vez de depender solo de correcciones posteriores sobre la solución.

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

# Parámetros del modelo
r = 0.5          # tasa de crecimiento
K = 1000         # capacidad de carga
N0 = 300          # población inicial
t_span = (0, 100) # intervalo de simulación
t_eval = np.linspace(*t_span, 1000)

def modelo_logistico(t, N, r, K, u):
    form = r * N * (1 - N / K) - u
    if N <= 0: # Si la población es menor o igual a 0
        return 0
    return form

# Casos de cosecha
u_sin = 0
u_moderada = r * K / 8           # menor que rK/4
u_excesiva = r * K / 2           # mayor que rK/4

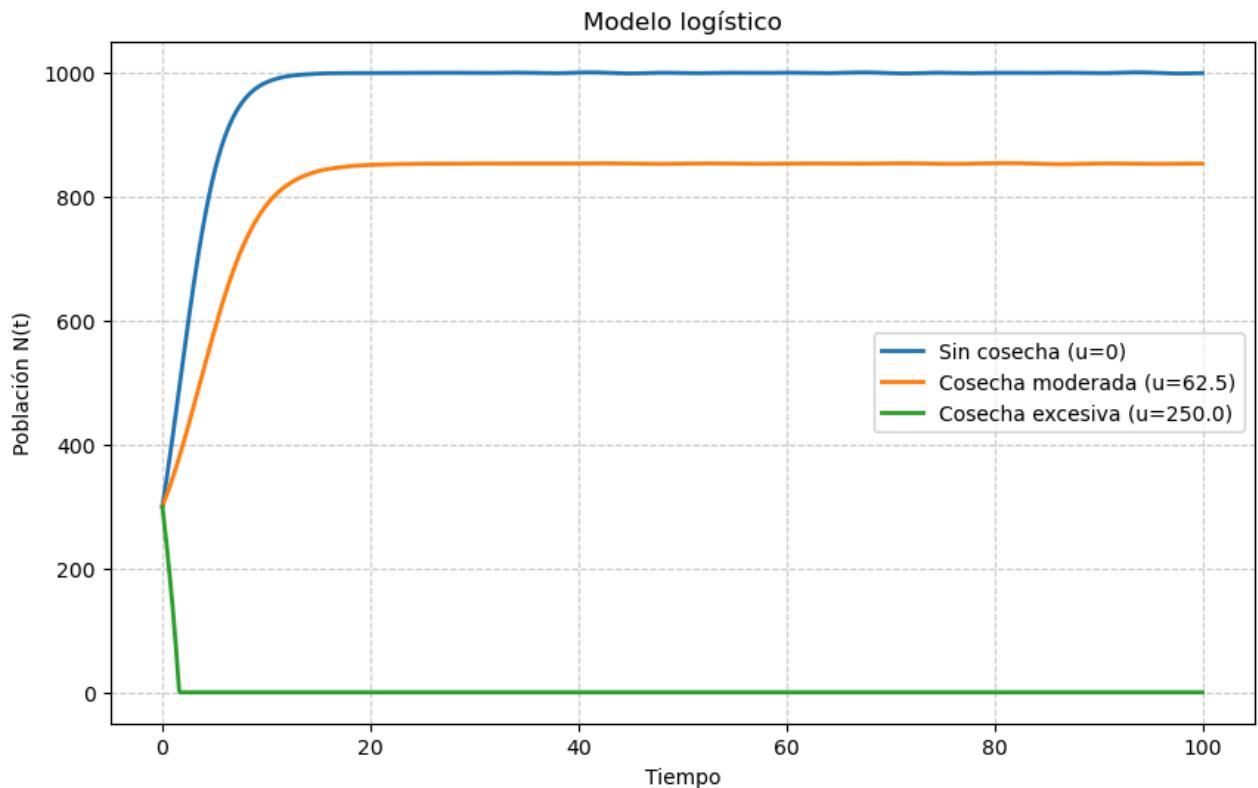
u_valores = [u_sin, u_moderada, u_excesiva]
labels = [
    "Sin cosecha (u=0)",
    f"Cosecha moderada (u={u_moderada:.1f})",
    f"Cosecha excesiva (u={u_excesiva:.1f})"
]

# Gráfico
plt.figure(figsize=(10, 6))

for i in range(3):
    u = u_valores[i]
    label = labels[i]
    sol = solve_ivp(modelo_logistico, t_span, [N0], args=(r, K, u), t_eval=t_eval)

    # Forzamos que N(t) no tenga valores negativos
    N = np.maximum(sol.y[0], 0)
    plt.plot(sol.t, N, label=label, lw=2)

# Personalización del gráfico
plt.title("Modelo logístico")
plt.xlabel("Tiempo")
plt.ylabel("Población N(t)")
plt.grid(True, ls="--", alpha=0.6)
plt.legend()
plt.show()
```



▼ Preguntas de reflexión

1. ¿Qué ocurre con la población en ausencia de cosecha ($u = 0$)? ¿Por qué tiende al valor K ?
2. Para una cosecha moderada ($0 < u < rK/4$), ¿por qué se alcanza un equilibrio distinto de K ? ¿Cuál es su interpretación biológica?
3. En el caso crítico ($u = rK/4$), ¿qué sucede con la dinámica poblacional?
4. Para una cosecha excesiva ($u > rK/4$), ¿por qué la población colapsa? ¿Cuál es la interpretación práctica de este resultado en términos de sostenibilidad?
5. Relaciona estos resultados con el concepto de **estabilidad** estudiado en teoría. ¿Cuál de los equilibrios es estable y cuál inestable?

1. ¿Qué ocurre con la población en ausencia de cosecha ($u = 0$)? ¿Por qué tiende al valor K ?

Con $u = 0$ el modelo vuelve al logístico clásico $N = rN(1 - N/K)$. Si $N > 0$ la tasa de crecimiento es positiva cuando $N < K$ y negativa cuando $N > K$, por lo que la dinámica empuja a la población hacia el punto donde $N = 0$, es decir $N = K$. Esto sucede porque matemáticamente K es un equilibrio global positivo, es decir, desde cualquier condición inicial $N(0) > 0$ la solución se approxima a K . Si pensamos biológicamente esto significa que los recursos limitan el tamaño máximo sostenible y la población se autorregula hasta ocupar la capacidad de carga del entorno.

2. Para una cosecha moderada ($0 < u < rK/4$), ¿por qué se alcanza un equilibrio distinto de K ? ¿Cuál es su interpretación biológica?

Cuando $0 < u < rK/4$ existen soluciones cuadráticas $rN(1-N/K) = u$, es decir hay dos raíces. La rama positiva mayor corresponde al equilibrio estable $N+ < K$, ya que el crecimiento neto $rN(1-N/K)$ compensa exactamente la extracción u . Físicamente, la extracción reduce la capacidad neta de crecimiento, por lo que la población debe asentarse en un valor más bajo que K para que la producción natural iguale la cosecha. Biológicamente, este equilibrio representa el tamaño sostenible bajo esa presión, porque la población no se extingue pero su abundancia se mantiene reducida debido a que parte de la producción se destina a servir a la cosecha en lugar de a la incrementación de la población.

3. En el caso crítico ($u = rK/4$), ¿qué sucede con la dinámica poblacional?

En el valor crítico $u = rK/4$ las dos raíces de la cuadrática dan una raíz doble $N = K/2$.

Matemáticamente esto es un punto de bifurcación en el que la naturaleza de los equilibrios cambia, porque la velocidad de retorno al equilibrio se reduce y el sistema deja ver un amortiguamiento lento. Desde un punto de vista dinámico, pequeñas perturbaciones alrededor de $K/2$ se van corrigiendo más lentamente que fuera del caso crítico. Además, cualquier perturbación que lleve la población por debajo de un cierto umbral puede hacerla descender hacia la extinción. Físicamente, este valor marca la frontera entre sostenibilidad y colapso, ya que es la máxima cosecha que aún permite un equilibrio positivo, y por eso se considera un caso marginal y delicado para la gestión.

4. Para una cosecha excesiva ($u > rK/4$), ¿por qué la población colapsa? ¿Cuál es la interpretación práctica de este resultado en términos de sostenibilidad?

Si $u > rK/4$ el resultado de la cuadrática es negativo y no existen equilibrios positivos, por lo que la ecuación $rN(1-N/K) = u$ no tiene una solución real para $N > 0$. Eso implica que, para cualquier $N \geq 0$, la tasa neta es negativa, por lo que la población disminuye continuamente hasta llegar a la extinción. En términos prácticos de sostenibilidad, esto significa que la tasa de extracción supera la capacidad de renovación del recurso, es decir, no hay ningún tamaño de población que produzca suficientes nacimientos/migraciones netas para compensar la cosecha constante. La interpretación es clara y preocupante, ya que explotar por encima de ese umbral conduce a una pérdida irreversible de la población o de un recurso, y la única medida efectiva para evitar el colapso es reducir u por debajo del umbral crítico o cambiar las proporciones de cosecha dependientes del tamaño.

5. Relaciona estos resultados con el concepto de **estabilidad** estudiado en teoría. ¿Cuál de los equilibrios es estable y cuál inestable?

En el modelo con cosecha existen niveles de población donde el sistema puede quedar quieto (equilibrios). Con cosecha baja aparece un equilibrio alto al que la población tiende con el tiempo. Ese equilibrio alto es estable, tiene pequeñas perturbaciones que se corrigen y la población vuelve donde estaba. También aparece un equilibrio bajo y que no es robusto frente a perturbaciones muy grandes. Este es un equilibrio más inestable, porque si la población cae ligeramente por debajo, sigue descendiendo.

El valor crítico de cosecha es el umbral que separa sostenibilidad de colapso. Justo este es el punto crítico donde el sistema es frágil y responde muy lentamente a cambios. Si la cosecha supera el umbral no existe equilibrio positivo sostenible. En ese caso la población decrece continuamente hasta la extinción. Desde el punto de vista de gestión, hay que mantener la extracción por debajo del

umbral . Ademñas convivne mantener la población por encima del umbral inestable para evitar el colpaso.

En resumente, un equilibrio alto es estable y deseable, por el contrario un equilibrio bajo es inestable y peligroso y si hay un exceso de cosecha nos encontramos con la extinción.

▼ Práctica 2 - Parte 2: Sistema masa-resorte con rozamiento

En esta práctica se estudiará el sistema masa-resorte-amortiguador clásico, que es un ejemplo típico de sistema dinámico de segundo orden.

Enunciado

1. Considera el sistema masa-resorte-amortiguador descrito por la ecuación diferencial:

$$m\ddot{x} + b\dot{x} + kx = 0$$

donde:

- m es la masa,
- b es el coeficiente de rozamiento,
- k es la constante del resorte.

2. Implementa el modelo en Python y simula la evolución temporal para distintos valores de b , con m y k fijos.

3. Ajusta los parámetros para obtener los distintos tipos de comportamiento dinámico:

- Sin amortiguamiento ($\zeta = 0$).
- Subamortiguado ($0 < \zeta < 1$).
- Críticamente amortiguado ($\zeta = 1$).
- Sobreamortiguado ($\zeta > 1$).

4. Representa gráficamente las respuestas y compara los resultados.

▼ Ejercicio propuesto

Implementa en Python el sistema masa resorte con rozamiento y representa la evolución temporal $x(t)$ con distintos valores de los parámetros. Ajusta los parámetros para obtener los comportamientos que se indican en el punto 3.

El código implementa la simulación numérica del sistema masa-resorte con amortiguamiento utilizando el método de integración de ecuaciones diferenciales ordinarias (ODE) proporcionado por la biblioteca SciPy. La ecuación diferencial $m\ddot{x} + b\dot{x} + kx = 0$ se reformula como un sistema de primer orden para facilitar la resolución numérica. Se fijan los valores de $m = 1$ kg y $k = 1$ N/m, y se varía el coeficiente de amortiguamiento b para obtener los diferentes regímenes dinámicos basados en el factor de amortiguamiento $\zeta = \frac{b}{2\sqrt{mk}}$. Las condiciones iniciales son $x(0) = 1$ m y $\dot{x}(0) = 0$ m/s. Finalmente se genera una gráfica que compara las respuestas temporales $x(t)$ para cada caso.

▼ Explicación del Código

1. Importación de Bibliotecas:

- `scipy.integrate.odeint`: Función principal para integrar numéricamente el sistema de ODEs. Resuelve el sistema convirtiendo la ecuación de segundo orden en dos ecuaciones de primer orden.

2. Definición de la Función del Sistema Dinámico (`damped`):

- Esta función define las derivadas del sistema: $\dot{x} = v$ (donde $v = \dot{x}$) y $\dot{v} = -\frac{b}{m}v - \frac{k}{m}x$.
- Los parámetros de entrada son el vector de estado $X = [x, v]$, el tiempo t , y los parámetros físicos b, m, k .
- Es crucial que la función devuelva una lista con las dos derivadas para que `odeint` pueda iterar correctamente. Esto representa la reformulación vectorial de la ecuación original.

3. Parámetros Fijos y Variables:

- $m = 1.0$ y $k = 1.0$: Valores fijos para simplificar el cálculo de ζ .
- Lista de b : [0.0, 1.0, 2.0, 4.0], que corresponden a $\zeta = 0$ (sin amortiguamiento, oscilaciones puras), $\zeta = 0.5$ (subamortiguado, oscilaciones decrecientes), $\zeta = 1$ (críticamente amortiguado, retorno rápido sin oscilar), y $\zeta = 2$ (sobreamortiguado, retorno lento sin oscilar).
- Vector de tiempo t : De 0 a 20 segundos con 1000 puntos, suficiente para capturar la dinámica sin aliasing.
- Condiciones iniciales $X0 = [1.0, 0.0]$: Desplazamiento inicial de 1 metro sin velocidad inicial.

4. Integración y Graficación:

- Para cada valor de b , se llama a `odeint` que resuelve el sistema y devuelve la solución `sol`, donde `sol[:, 0]` es $x(t)$.
- Se grafica $x(t)$ para cada b en la misma figura, con una leyenda que indica el valor de b .
- Etiquetas de ejes: Tiempo en segundos y posición x en metros. La gráfica muestra cómo aumenta el amortiguamiento reduce las oscilaciones hasta eliminarlas por completo.

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

# definicion del sistema de ecuaciones diferenciales
def damped(X, t, b, m, k):
```

```

x, v = X
dxdt = v # Derivada de x: velocidad
dvdt = - (b / m) * v - (k / m) * x # Derivada de v: aceleración según la ecuación
return [dxdt, dvdt] # se retornan las derivadas como lista

# Parámetros fijos
m = 1.0 # Masa en kg
k = 1.0 # Constante del resorte en N/m

# Valores de b para diferentes régimenes (zeta = b / (2 * sqrt(m * k)))
b_values = [0.0, 1.0, 2.0, 4.0] # zeta = 0, 0.5, 1, 2

# Condiciones iniciales: x(0) = 1 m, v(0) = 0 m/s
X0 = [1.0, 0.0]

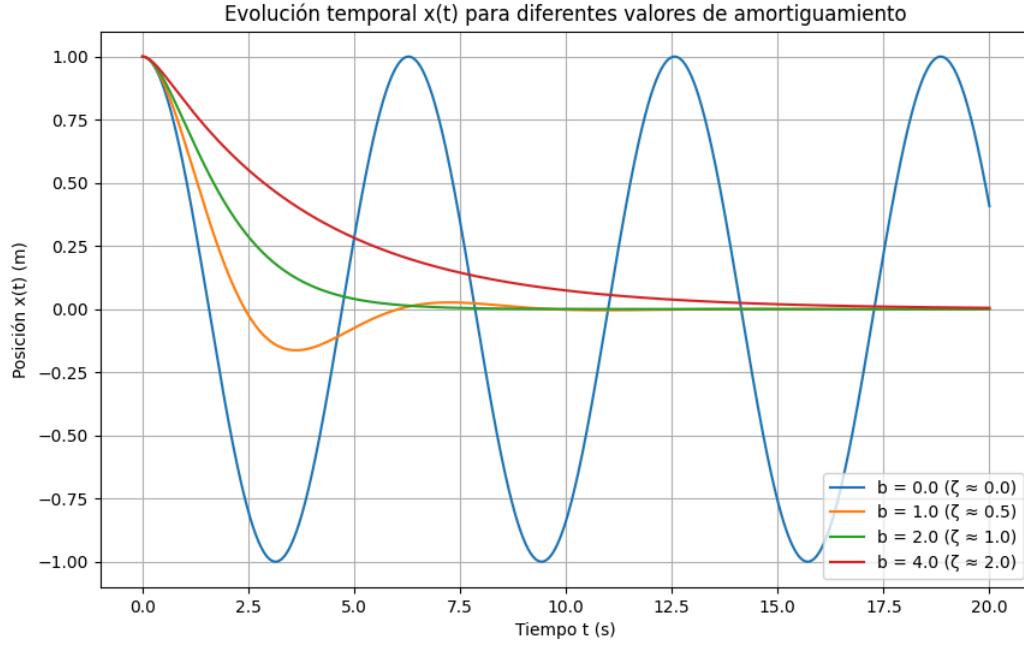
# Vector de tiempo: de 0 a 20 s con 1000 particiones
t = np.linspace(0, 20, 1000)

plt.figure(figsize=(10, 6))
for b in b_values:
    ...
        para cada valor de b se integrara y graficara el sistema,
        se usa odeint que integra numéricamente el sistema y finalmente
        la solución es lo que se grafica
    ...
    sol = odeint(damped, X0, t, args=(b, m, k))

    plt.plot(t, sol[:, 0], label=f'b = {b} (\zeta \approx {b/2:.1f})')

plt.xlabel('Tiempo t (s)')
plt.ylabel('Posición x(t) (m)')
plt.title('Evolución temporal x(t) para diferentes valores de amortiguamiento')
plt.legend()
plt.grid(True)
plt.show()

```



✗ Ejercicio propuesto OPCIONAL

Representa las raíces en el plano complejo para cada uno de los casos.

El siguiente código calcula y visualiza las raíces de la ecuación característica del sistema en el plano complejo para cada régimen de amortiguamiento. La ecuación característica es $mr^2 + br + k = 0$ cuyas raíces determinan el comportamiento: raíces imaginarias puras (oscilatorio), complejas con parte real negativa (amortiguado), o reales negativas (sobreamortiguado). Se usan los mismos parámetros $m = 1$ $k = 1$ y los valores de b del código anterior. Las raíces se calculan analíticamente usando la fórmula cuadrática y se grafican en el plano real-imaginario, donde el eje real representa la parte exponencial de decaimiento y el imaginario la frecuencia osculatoria.

✗ Explicación del Código

1. Fórmula de las Raíces:

- La ecuación característica es $r^2 + \frac{b}{m}r + \frac{k}{m} = 0$
- Raíces: $r = \frac{-\frac{b}{m} \pm \sqrt{\left(\frac{b}{m}\right)^2 - 4\frac{k}{m}}}{2}$.

$$\circ \text{ Con } m = 1 \text{ } k = 1: r = \frac{-b \pm \sqrt{b^2 - 4}}{2}.$$

2. Cálculo de Raíces para Cada b :

- Se itera sobre la lista de b calculando el discriminante $d = b^2 - 4$.
- Las raíces $r1, r2$ se computan con `np.sqrt` para manejar el caso complejo automáticamente.
- Se extraen las partes real e imaginaria de cada raíz usando `.real` e `.imag`.

3. Graficación en el Plano Complejo:

- Eje x: parte real (decaimiento, siempre negativo o cero).
- Eje y: parte imaginaria (frecuencia angular).
- Cada par de raíces se marca con un símbolo diferente y una etiqueta que indica b y ζ .
- Líneas verticales en el eje imaginario ayudan a visualizar los casos oscilatorios. La gráfica muestra cómo las raíces se mueven hacia la izquierda (mayor decaimiento) al aumentar b y pierden la componente imaginaria en el caso sobreamortiguado.

```

import numpy as np
import matplotlib.pyplot as plt

# Parámetros fijos
m = 1.0 # Masa
k = 1.0 # Constante del resorte

colors = ['blue', 'green', 'orange', 'red']

b_values = [0.0, 1.0, 2.0, 4.0]
labels = ['b=0 ( $\zeta=0$ , sin amortiguación)', 'b=1 ( $\zeta=0.5$ , subamortiguado)', 'b=2 ( $\zeta=1$ , crítico)', 'b=4 ( $\zeta=2$ , sobreamortiguado)']

fig, ax = plt.subplots(figsize=(12, 9))

# calculo de las raices segun su formula
for i, b in enumerate(b_values):

    discriminant = (b / m)**2 - 4 * (k / m)
    discriminant_complex = discriminant + 0j # Convertir a tipo complejo

    # Raices (fórmula cuadrática con sqrt compleja)
    r1 = (-b / m + np.sqrt(discriminant_complex)) / 2
    r2 = (-b / m - np.sqrt(discriminant_complex)) / 2

    # Partes real e imaginaria
    real1, imag1 = r1.real, r1.imag
    real2, imag2 = r2.real, r2.imag

    ax.plot(real1, imag1, 'o', markersize=14, label=labels[i], color=colors[i],
            markeredgewidth=2, zorder=10)

    ax.plot(real2, imag2, 'o', color=colors[i], markersize=14,
            markeredgewidth=2, zorder=10)

    # unimos las dos raices con una linea
    ax.plot([real1, real2], [imag1, imag2], color=colors[i],
            linestyle='-', alpha=0.6, linewidth=2.5, zorder=8)

# bordes mas negros para los ejes
ax.grid(True)
ax.axhline(y=0, color='black', linewidth=1, zorder=5)
ax.axvline(x=0, color='black', linewidth=1, zorder=5)

ax.set_xlabel('Parte Real ( $\leftarrow$  Decaimiento exponencial)', fontsize=13, fontweight='bold')

ax.set_ylabel('Parte Imaginaria ( $\uparrow$  Frecuencia de oscilación)', fontsize=13, fontweight='bold')

# unas labels para situar mejor los ejes
ax.text(0.45, 0.05, 'EJE REAL  $\rightarrow$ ', fontsize=11, fontweight='bold',
        color='black', ha='right', va='bottom',
        bbox=dict(boxstyle='round,pad=0.5', facecolor='yellow', alpha=0.7))

ax.text(0.04, 1.35, ' $\uparrow$  EJE IMAGINARIO', fontsize=11, fontweight='bold',
        color='black', ha='left', va='top',
        bbox=dict(boxstyle='round,pad=0.5', facecolor='lightblue', alpha=0.7))

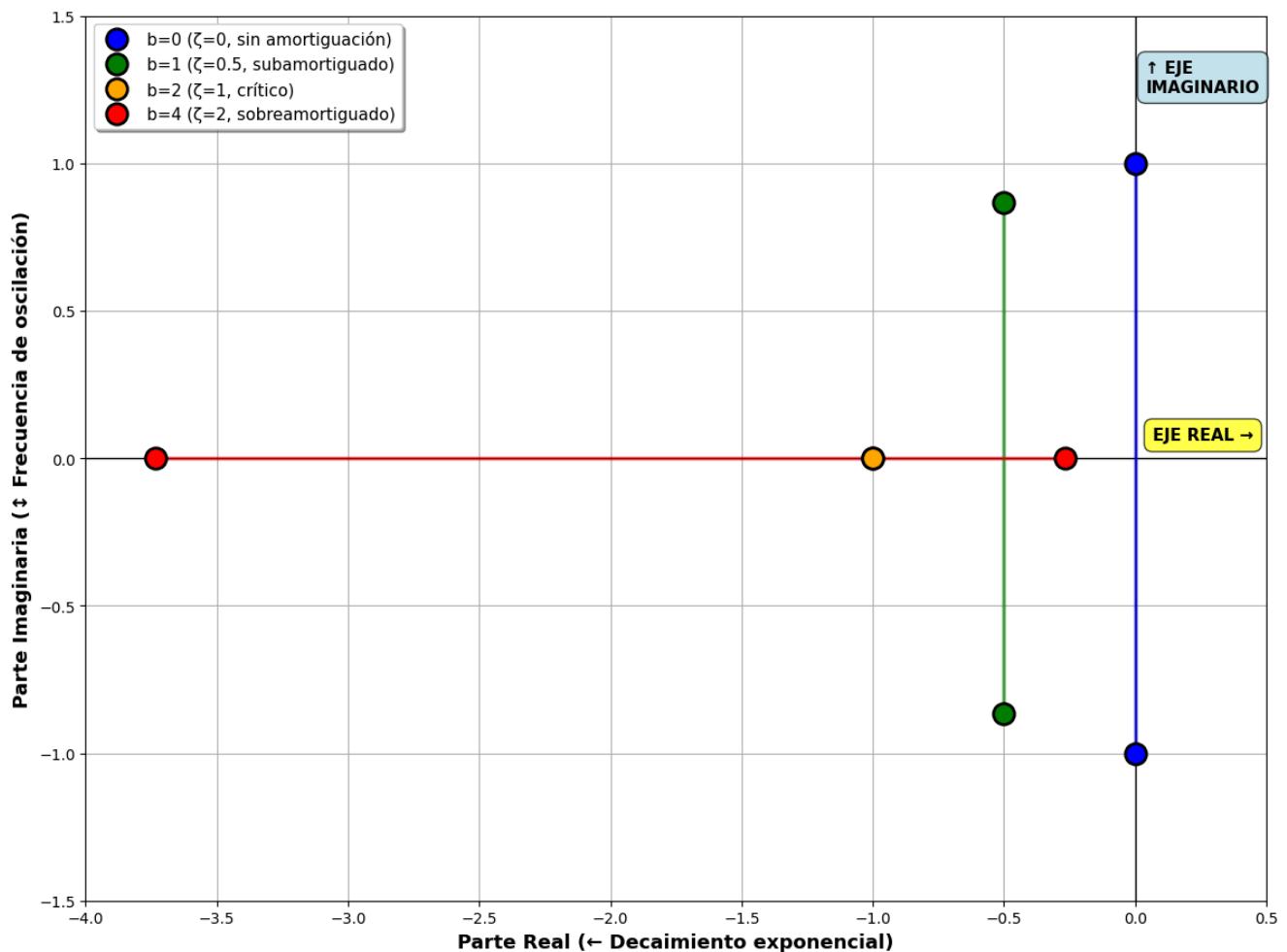
ax.legend(loc='upper left', fontsize=11, fancybox=True, shadow=True)
ax.set_xlim(-4, 0.5)
ax.set_ylim(-1.5, 1.5)

plt.tight_layout()
plt.show()
plt.close(fig)

for i, b in enumerate(b_values):
    discriminant = (b / m)**2 - 4 * (k / m)
    discriminant_complex = discriminant + 0j
    r1 = (-b / m + np.sqrt(discriminant_complex)) / 2
    r2 = (-b / m - np.sqrt(discriminant_complex)) / 2

```

```
print(f"\n{labels[i]}")
print(f" Raíz 1 (r1): {r1.real:.4f} {r1.imag:+.4f}j")
print(f" Raíz 2 (r2): {r2.real:.4f} {r2.imag:+.4f}j")
print("-"*70)
```



b=0 ($\zeta=0$, sin amortiguación)
Raíz 1 (r_1): 0.0000 +1.0000j
Raíz 2 (r_2): -0.0000 -1.0000j

b=1 ($\zeta=0.5$, subamortiguado)
Raíz 1 (r_1): -0.5000 +0.8660j
Raíz 2 (r_2): -0.5000 -0.8660j

b=2 ($\zeta=1$, crítico)
Raíz 1 (r_1): -1.0000 +0.0000j
Raíz 2 (r_2): -1.0000 +0.0000j

b=4 ($\zeta=2$, sobreamortiguado)
Raíz 1 (r_1): -0.2679 +0.0000j
Raíz 2 (r_2): -3.7321 +0.0000j

▼ Preguntas de reflexión

1. ¿Qué diferencias observas en la respuesta entre el caso sin amortiguamiento y el caso subamortiguado?

En el caso sin amortiguamiento ($b=0, \zeta=0$), la respuesta temporal $x(t)$ muestra oscilaciones sinusoidales puras con amplitud constante, sin decaimiento, lo que se refleja en raíces puramente imaginarias en el plano complejo ($0 \pm 1j$), indicando una frecuencia de oscilación constante sin componente de decaimiento.

Por el contrario, en el caso subamortiguado ($b=1, \zeta=0.5$), $x(t)$ exhibe oscilaciones con amplitud decreciente exponencialmente, cruzando el eje de equilibrio múltiples veces pero con picos cada vez más pequeños, correspondiente a raíces complejas conjugadas ($-0.5 \pm 0.866j$) con parte real negativa (decaimiento) y parte imaginaria no cero (oscilación persistente pero amortiguada).

La diferencia clave radica en la energía: sin amortiguamiento, la energía se conserva indefinidamente en oscilaciones; en subamortiguado, la fricción disipa energía gradualmente, haciendo que el sistema regrese al equilibrio de forma oscillatoria pero controlada.

✓ 2. ¿Por qué en el caso crítico la respuesta es la más rápida posible sin oscilaciones?

El caso críticamente amortiguado ($b=2$, $\zeta=1$) ofrece la respuesta más rápida sin oscilaciones porque las raíces son reales y dobles ($-1 + 0j$), lo que genera una solución $x(t) = (A + Bt)e^{-t}$ donde el término lineal Bt optimiza el retorno al equilibrio en el menor tiempo posible sin sobrepasar ni oscilar, como se ve en la curva verde que decae suavemente desde $x=1$ hacia cero sin cruzar en ningún caso el eje x .

Esta configuración representa el límite entre oscillatorio y no oscillatorio: cualquier amortiguamiento menor introduce oscilaciones (subamortiguado, curva naranja con cruces de eje x), mientras que mayor genera decaimiento más lento (sobreamortiguado, curva roja con menos curvatura).

Matemáticamente, el factor $\zeta=1$ minimiza el tiempo de asentamiento al maximizar la componente de decaimiento sin inestabilidad oscillatoria.

✓ 3. ¿Qué efecto tiene un rozamiento excesivo (sobreamortiguado) sobre la rapidez de la respuesta?

En el sobreamortiguado ($b=4$, $\zeta=2$), el rozamiento excesivo hace que la respuesta sea más lenta: $x(t)$ decae sin oscilar, pero con una aproximación gradual y prolongada al equilibrio, como muestra la curva roja que permanece por encima del eje y y se acerca a cero más despacio que el caso crítico.

Esto se debe a raíces reales distintas y ambas muy negativas (aprox. $-0.268 \pm 0j$ y $-3.732 \pm 0j$), donde la raíz menos negativa domina el decaimiento lento, resultando en un "arrastre" excesivo que impide un retorno rápido.

El efecto negativo es la reducción de la velocidad de respuesta: aunque estable, el sistema pierde eficiencia.

✓ 4. ¿Cómo se relaciona el coeficiente de amortiguamiento ζ con la posición de las raíces en el plano complejo?

El coeficiente de amortiguamiento $\zeta = b/(2\sqrt{mk})$ determina la posición de las raíces $r = [-b/m \pm \sqrt{((b/m)^2 - 4k/m)]/2}$ en el plano complejo: para $\zeta < 1$, las raíces tienen parte imaginaria $\neq 0$; para $\zeta = 1$, raíces reales dobles en el eje real negativo (sin imaginaria, decaimiento óptimo); y para $\zeta > 1$, raíces reales separadas en el eje real (sin imaginaria, decaimiento lento).

Especificamente, ζ controla el discriminante $\Delta = (b/m)^2 - 4k/m = 4\zeta^2 - 4$: si $\Delta < 0$ ($\zeta < 1$), imaginaria = $\sqrt{(-\Delta)/2} > 0$; si $\Delta = 0$ ($\zeta = 1$), imaginaria = 0 y real = $-\sqrt{k/m}$; si $\Delta > 0$ ($\zeta > 1$), dos reales negativas separadas.

En general, $\zeta \geq 1$ mantiene las raíces en el semiplano izquierdo (estabilidad, parte real ≤ 0), pero valores altos de ζ desplazan las raíces más a la izquierda (mayor decaimiento, pero más lento en el dominante).

✓ Práctica 2 - Parte 3: Modelo de horno eléctrico

En esta parte trabajaremos con un modelo simplificado de un horno eléctrico, como el que hemos visto en la teoría. El objetivo es comprender cómo evoluciona la temperatura y cómo se puede diseñar una ley de control para alcanzar un valor deseado.

Enunciado general

1. Implementa en Python el modelo simplificado de un horno eléctrico representado como un sistema dinámico de primer orden.
2. Representa gráficamente la evolución de la temperatura para una entrada de potencia constante.
3. Diseña una ley de control sencilla que permita alcanzar una temperatura deseada T_d .
4. Analiza y discute el comportamiento obtenido.

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

✓ Ejercicio 1

Implementa el modelo del horno eléctrico como un sistema dinámico de primer orden:

$$\dot{T}(t) = -a(T(t) - T_{amb}) + bu(t)$$

donde:

- $T(t)$ es la temperatura interna del horno,
- T_{amb} es la temperatura ambiente,
- $u(t)$ es la potencia de entrada (entre 0 y 1),
- $a > 0$ (coeficiente de enfriamiento) y $b > 0$ (ganancia de calentamiento) son parámetros del sistema.

Simula la evolución de $T(t)$ cuando $u(t)$ es constante y representa la gráfica.

```
# Parámetros
a = 0.1 # Constante de enfriamiento
b = 0.5 # Ganancia de calentamiento
T_amb = 25 # Temperatura ambiente
u_const = 10 # Potencia constante aplicada al horno
T0 = 25 # Temperatura inicial del horno

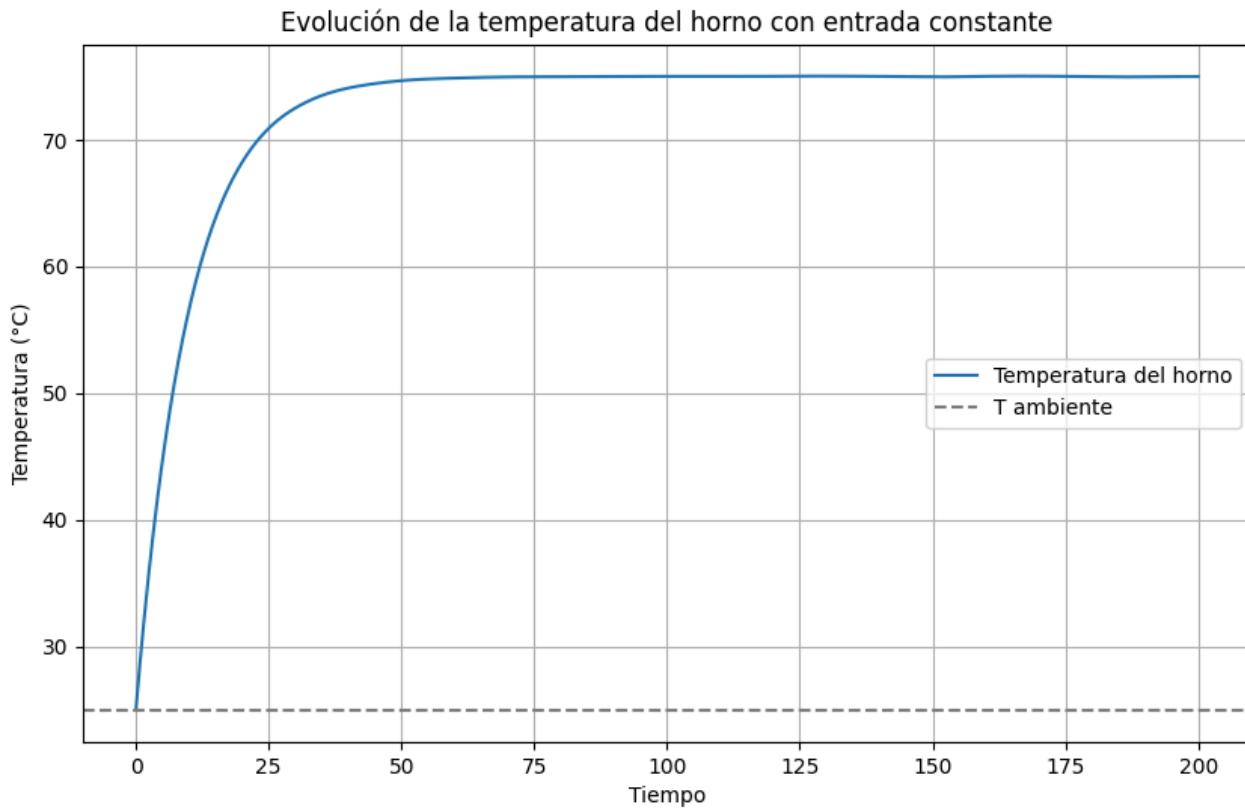
# Definición de la función diferencial
def horno(t, T, a, b, u, T_amb):
    return -a*(T - T_amb) + b*u

# Simulación con entrada constante
t_span = (0, 200) # Intervalo de tiempo
t_eval = np.linspace(t_span[0], t_span[1], 1000) # Puntos de evaluación

# Resolver la ecuación diferencial
sol = solve_ivp(
    horno,                  # Función que define la dinámica del sistema
    t_span,                 # Intervalo de integración
    [T0],                   # Condición inicial (temperatura inicial)
    t_eval=t_eval,          # Tiempos donde se evaluará la solución
    args=(a, b, u_const, T_amb) # Parámetros adicionales
)

# Graficar los resultados
fig = plt.figure(figsize=(10,6))
plt.plot(sol.t, sol.y[0], label="Temperatura del horno")
plt.axhline(T_amb, color="gray", linestyle="--", label="T ambiente")
plt.xlabel("Tiempo")
plt.ylabel("Temperatura (°C)")
```

```
plt.title("Evolución de la temperatura del horno con entrada constante")
plt.legend()
plt.grid(True)
plt.close(fig)
```



Este sistema modela cómo la temperatura interna del horno cambia con el tiempo, de manera que si la potencia permanece constante, la temperatura aumenta de forma exponencial desde la temperatura ambiente y, además, el sistema tiende a estabilizarse en una temperatura de equilibrio. Esto ocurre cuando el calor proporcionado por la potencia de entrada se iguala con las pérdidas de calor hacia el ambiente.

▼ Ejercicio 2

Diseña una **ley de control** que lleve al horno a una temperatura deseada T_d . Una ley de control sencilla para $u(t)$ puede ser del tipo proporcional:

$$u(t) = k_p(T_d - T(t))$$

donde $k_p > 0$ es la ganancia proporcional y $(T_d - T(t))$ es el error.

Simula y representa el comportamiento del horno con este control. Y marca con una línea discontinua la temperatura deseada.

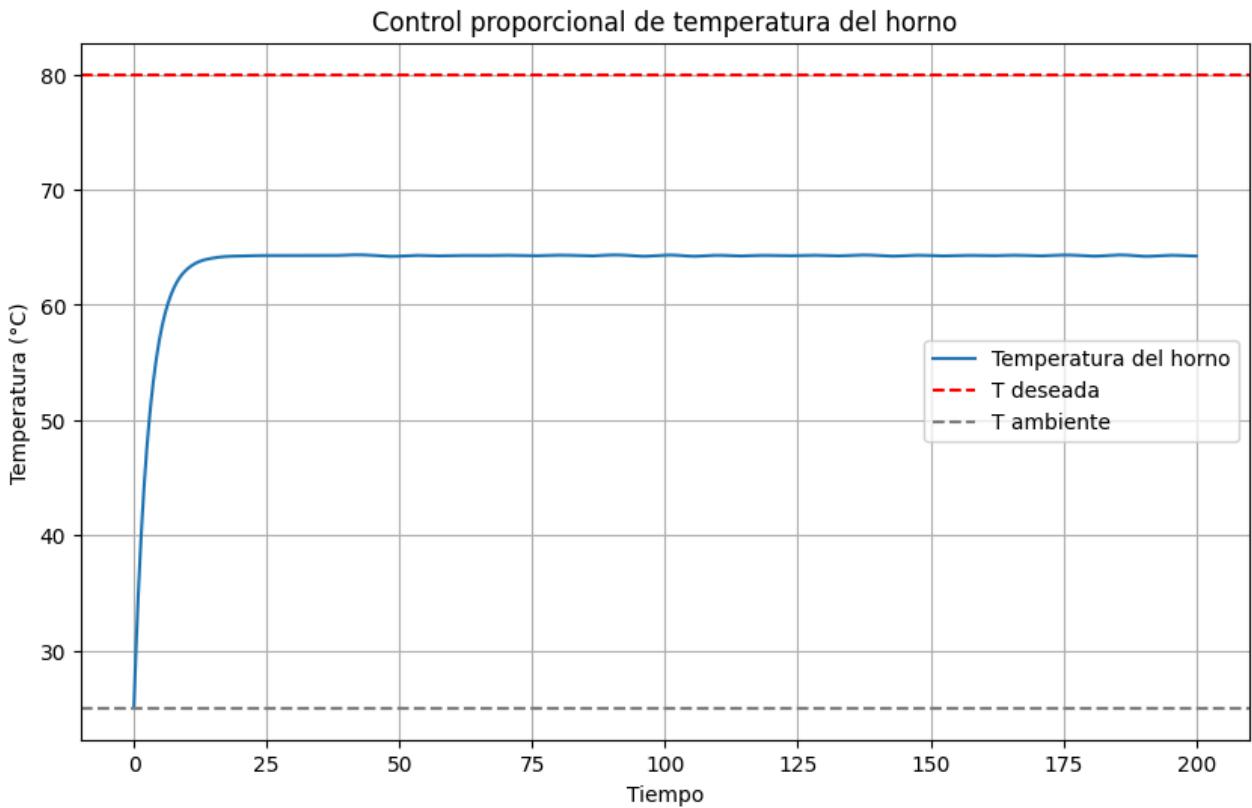
```
# Parámetros del sistema
a = 0.1
b = 0.5
T_amb = 25
T0 = 25
T_d = 80 # Temperatura deseada
k_p = 0.5 # Ganancia proporcional

# Definición de la función diferencial con control proporcional
def horno_control(t, T, a, b, k_p, T_d, T_amb):
    u = k_p * (T_d - T) # Ley de control proporcional
    return -a*(T - T_amb) + b*u

# Simulación con control proporcional
t_span = (0, 200)
t_eval = np.linspace(t_span[0], t_span[1], 1000)
```

```
# Resolver la ecuación diferencial
sol_control = solve_ivp(
    horno_control,
    t_span,
    [T0],
    t_eval=t_eval,
    args=(a, b, k_p, T_d, T_amb)
)

# Graficar los resultados
fig = plt.figure(figsize=(10,6))
plt.plot(sol_control.t, sol_control.y[0], label="Temperatura del horno")
plt.axhline(T_d, color="red", linestyle="--", label="T deseada")
plt.axhline(T_amb, color="gray", linestyle="--", label="T ambiente")
plt.xlabel("Tiempo")
plt.ylabel("Temperatura (°C)")
plt.title("Control proporcional de temperatura del horno")
plt.legend()
plt.grid(True)
plt.show()
```



```
# Parámetros del sistema
a = 0.1
b = 0.5
T_amb = 25
T0 = 25
T_d = 80 # Temperatura deseada
k_p = 1 # Ganancia proporcional

# Definición de la función diferencial con control proporcional
def horno_control(t, T, a, b, k_p, T_d, T_amb):
    u = k_p * (T_d - T) # Ley de control proporcional
    return -a*(T - T_amb) + b*u

# Simulación con control proporcional
t_span = (0, 200)
t_eval = np.linspace(t_span[0], t_span[1], 1000)

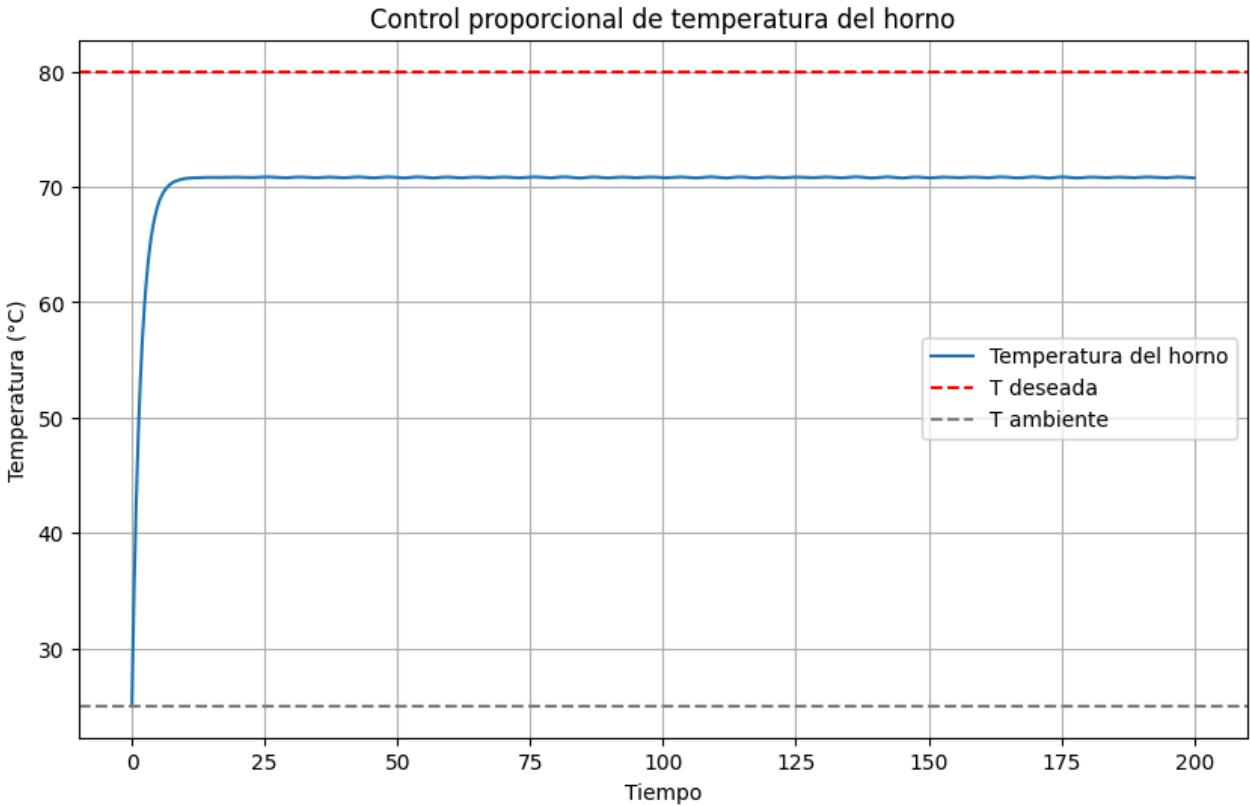
# Resolver la ecuación diferencial
sol_control = solve_ivp(
```

```

        horno_control,
        t_span,
        [T0],
        t_eval=t_eval,
        args=(a, b, k_p, T_d, T_amb)
    )

# Graficar los resultados
plt.figure(figsize=(10,6))
plt.plot(sol_control.t, sol_control.y[0], label="Temperatura del horno")
plt.axhline(T_d, color="red", linestyle="--", label="T deseada")
plt.axhline(T_amb, color="gray", linestyle="--", label="T ambiente")
plt.xlabel("Tiempo")
plt.ylabel("Temperatura (°C)")
plt.title("Control proporcional de temperatura del horno")
plt.legend()
plt.grid(True)
plt.show()

```



```

# Parámetros del sistema
a = 0.1
b = 0.5
T_amb = 25
T0 = 25
T_d = 80 # Temperatura deseada
k_p = 20000 # Ganancia proporcional

# Definición de la función diferencial con control proporcional
def horno_control(t, T, a, b, k_p, T_d, T_amb):
    u = k_p * (T_d - T) # Ley de control proporcional
    return -a*(T - T_amb) + b*u

# Simulación con control proporcional
t_span = (0, 200)
t_eval = np.linspace(t_span[0], t_span[1], 1000)

# Resolver la ecuación diferencial
sol_control = solve_ivp(
    horno_control,
    t_span,
    [T0],
    t_eval=t_eval,
)

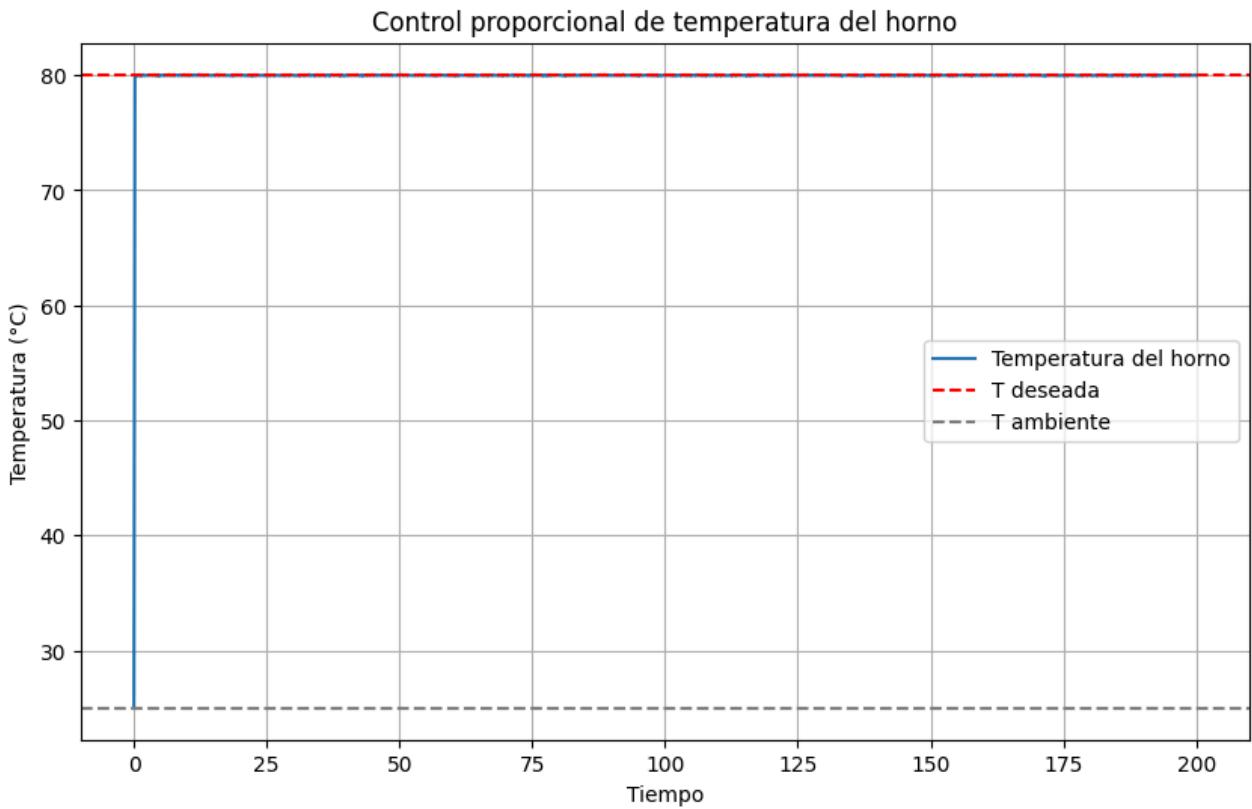
```

```

    args=(a, b, k_p, T_d, T_amb)
)

# Graficar los resultados
plt.figure(figsize=(10,6))
plt.plot(sol_control.t, sol_control.y[0], label="Temperatura del horno")
plt.axhline(T_d, color="red", linestyle="--", label="T deseada")
plt.axhline(T_amb, color="gray", linestyle="--", label="T ambiente")
plt.xlabel("Tiempo")
plt.ylabel("Temperatura (°C)")
plt.title("Control proporcional de temperatura del horno")
plt.legend()
plt.grid(True)
plt.show()

```



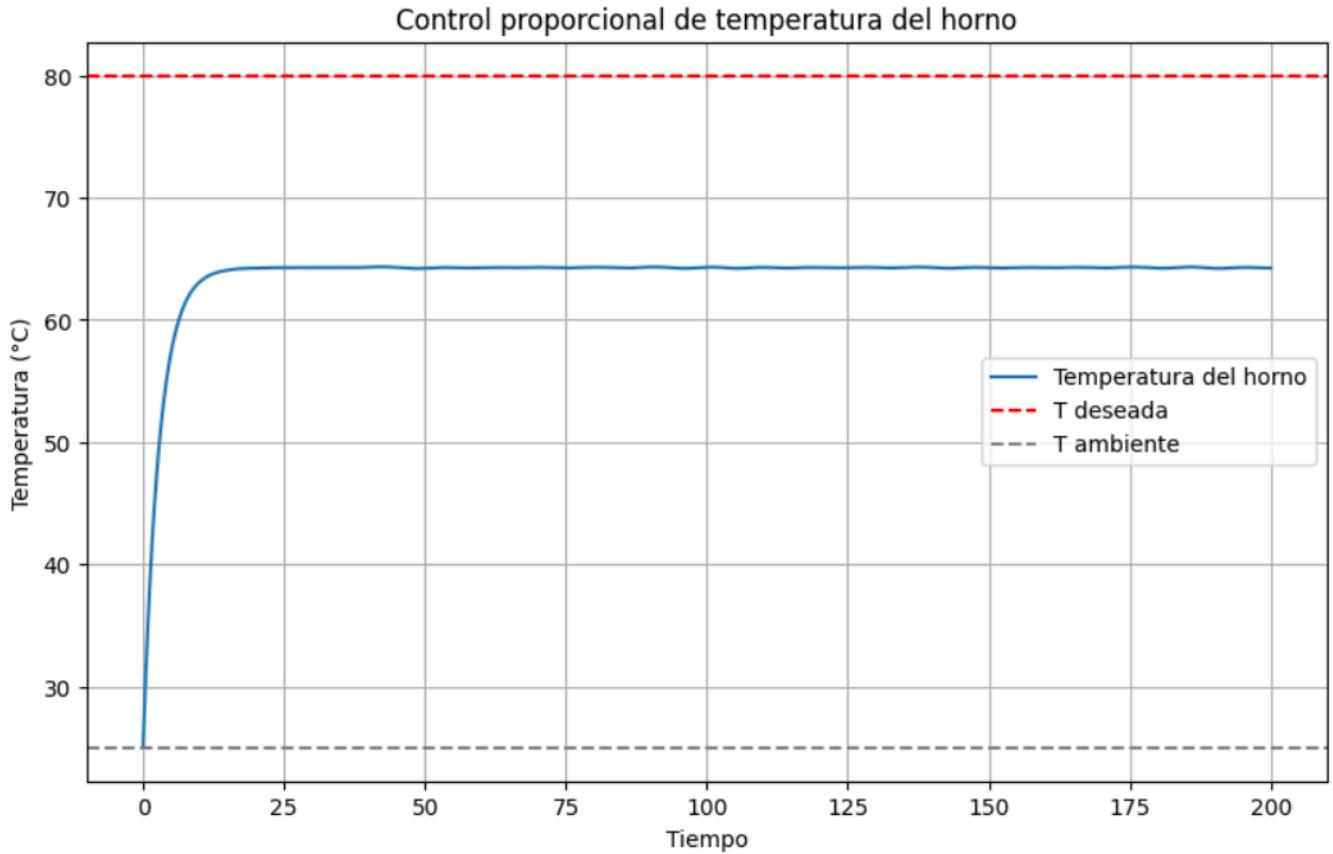
Preguntas de reflexión

1. ¿Cómo cambia el comportamiento del horno cuando se usa una entrada constante frente a una ley de control proporcional?

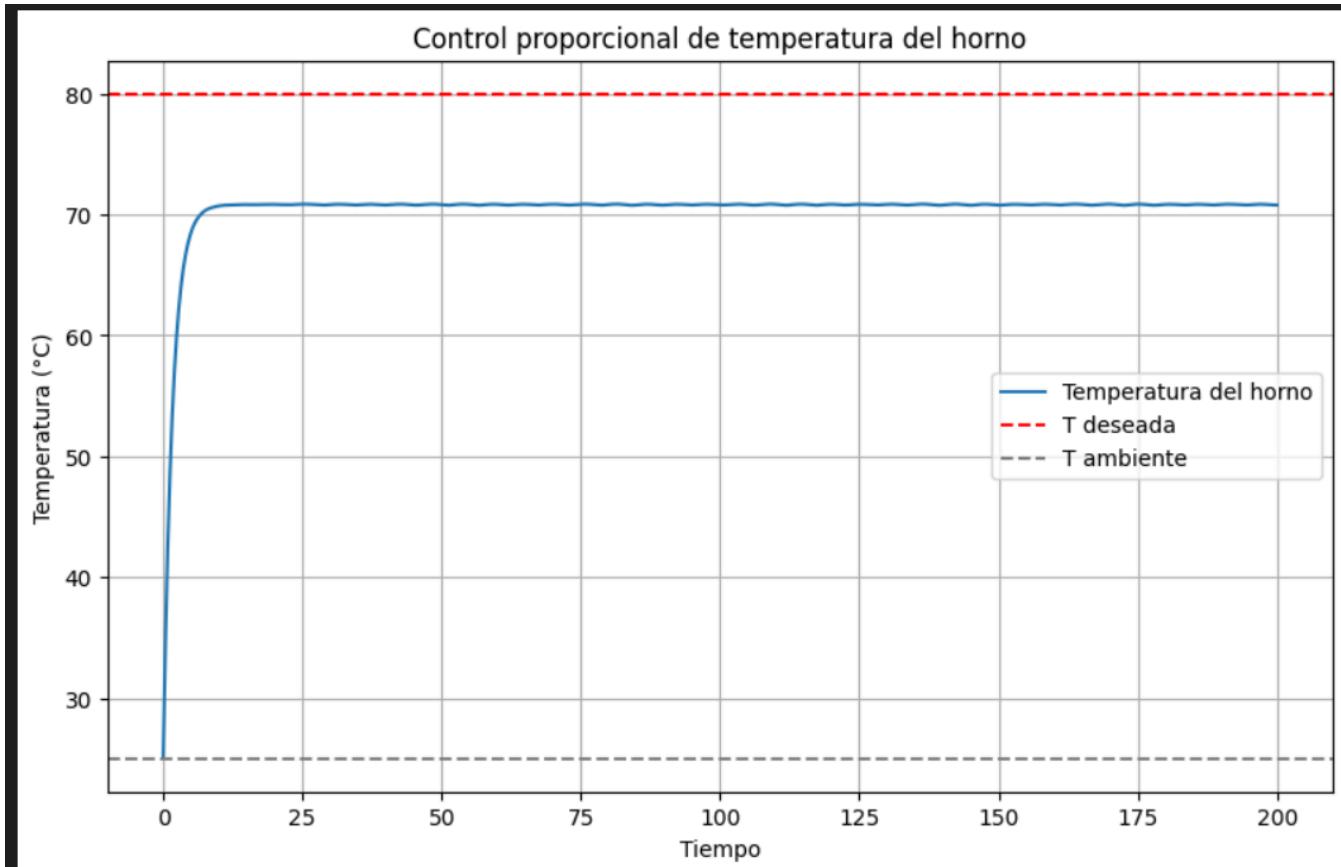
Cuando la entrada es constante (Ejercicio 1), el sistema alcanza un equilibrio donde, como hemos mencionado, la ganancia de calentamiento que entra se iguala con las pérdidas. Esto viene a decir que con una entrada fija (u) solo se puede llegar a la temperatura que esa potencia puede mantener. Sin embargo, con la ley de control proporcional, la entrada se reduce a medida que se reduce el error. Esto hace que la potencia se autorregule en función del error, aunque no fuerza a alcanzar la temperatura deseada a no ser que la ganancia proporcional k_p tienda a infinito.

2. ¿Qué ocurre si la ganancia k_p es muy pequeña? ¿Y si es demasiado grande?

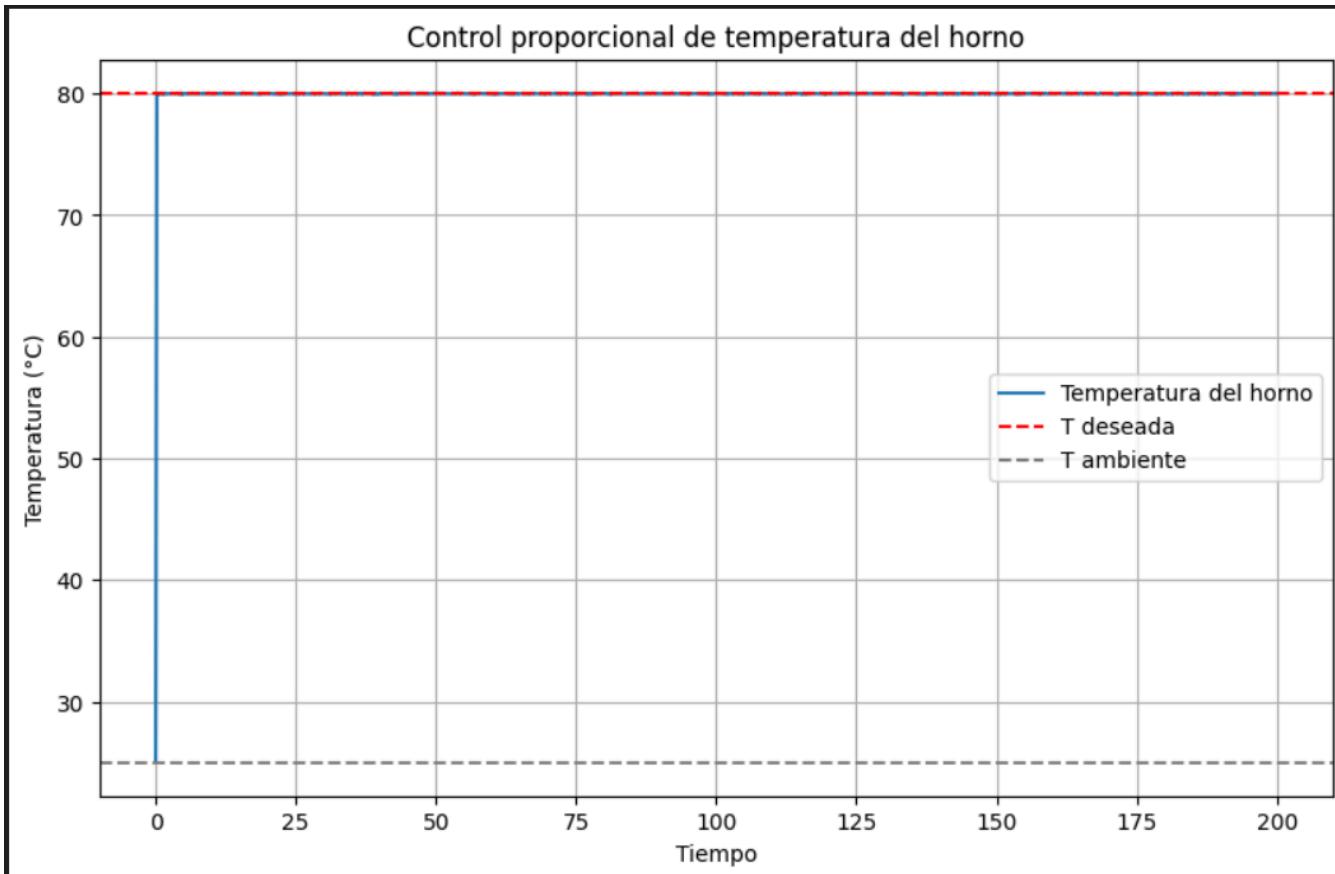
Si la ganancia proporcional es muy pequeña, el sistema se estabiliza lejos del estado deseado con un error mayor:



En cambio, a medida que aumenta k_p , la respuesta es más rápida y hay menos error en el estado estacionario, aunque no cero:



Y ya si este valor es demasiado grande, tendiendo a infinito, la respuesta es todavía más rápida (calentamiento más rápido). Sin embargo, esto puede amplificar el ruido y saturar el sistema, provocando incluso valores físicamente irreales. Además, podemos observar como, por mucho que se aumente k_p , nunca se sobrepasa la temperatura deseada, sino que se aproxima a ella:



3. ¿Qué diferencias observas entre la temperatura ambiente, la deseada y la alcanzada en cada caso?

La temperatura ambiente es la referencia inicial, de manera que si hay un aporte de energía todas las soluciones tienden a un equilibrio por encima de esta temperatura.

La temperatura deseada es el objetivo a alcanzar. En el caso de entrada constante, la temperatura alcanzada es aproximadamente 75°C, que aún está más cerca del ideal establecido posteriormente de 80°C que en el caso de $k_p = 0.5$, donde la temperatura alcanzada es a penas algo más de 60°C. Eso sí, cuando k_p toma un valor mayor (20000, por ejemplo), la temperatura alcanzada es aproximadamente equivalente a la deseada. Es decir, si k_p no es lo suficientemente grande, la acción de control proporcional no tiene la fuerza suficiente para compensar la diferencia entre la temperatura actual y la deseada, por lo que el sistema queda con un error en régimen permanente.

4. ¿Cómo se relaciona este ejemplo con la noción de estabilidad vista en teoría?

Este sistema del horno es un buen ejemplo para analizar los diferentes tipos de estabilidad vistos en teoría:

- **Puntos de equilibrio:** Para una entrada constante u , el punto de equilibrio T_e se obtiene haciendo $\dot{T} = 0$:

$$0 = -a(T_e - T_{amb}) + bu \rightarrow T_e = T_{amb} + \frac{bu}{a}$$

Con control proporcional $u = k_p(T_d - T)$, el punto de equilibrio es:

$$T_e = \frac{aT_{amb} + b k_p T_d}{a + b k_p}$$

- **Estabilidad de Lyapunov:** El sistema muestra estabilidad de Lyapunov porque para cualquier perturbación ϵ alrededor del equilibrio, el sistema permanece cerca de él. Esto se observa en las gráficas donde, independientemente de k_p , la temperatura siempre converge a un valor estable.
- **Estabilidad asintótica:** El sistema es asintóticamente estable porque no solo permanece cerca del equilibrio, sino que converge a él: $\lim_{t \rightarrow \infty} T(t) = T_e$. Esto se observa en todas las simulaciones, donde la temperatura siempre alcanza un valor final estable.
- **Estabilidad exponencial:** La convergencia al equilibrio es exponencial, como se puede ver en la forma de las curvas.

En cuanto a los resultados obtenidos, podemos confirmar que el sistema muestra un comportamiento monótono (no oscilatorio) debido a que es de primer orden. Además, el amortiguamiento viene dado por el término $-a(T - T_{amb})$; y el control proporcional aumenta el amortiguamiento efectivo a $-(a + b k_p)(T - T_e)$.

Este ejemplo ilustra cómo un sistema físico simple puede mostrar varios aspectos de estabilidad, y cómo el control proporcional puede modificar las características de estabilidad manteniendo la estructura fundamental del sistema.

✓ Ejercicio opcional: Control PID

Además del control proporcional, se puede utilizar un **control PID (Proporcional–Integral–Derivativo)** para mejorar la respuesta del horno.

La ley de control PID se expresa como:

$$u(t) = k_p e(t) + k_i \int e(t) dt + k_d \frac{de(t)}{dt}$$

donde $e(t) = T_d - T(t)$ es el error.

1. Implementa un controlador PID sencillo para el horno.
2. Simula y compara los resultados con el control proporcional.

```
# Parámetros del sistema
a = 0.1
b = 0.5
T_amb = 25
T0 = 25
T_d = 80

# Parámetros para los controladores a comparar
kp_p = 1.0          # Proporcional de referencia
kp_pi = 1.0; ki_pi = 0.05 # PI
kp_pid = 1.0; ki_pid = 0.05; kd_pid = 0.5 # PID

u_min, u_max = 0.0, 100.0 # Saturación del actuador (potencia)

def horno_P(t, T, a, b, kp, T_d, T_amb):
    u = kp*(T_d - T)
    u = np.clip(u, u_min, u_max)
    return -a*(T - T_amb) + b*u

# PI (estado [T, I])
def horno_PI(t, y, a, b, kp, ki, T_d, T_amb):
    T, I = y
    e = T_d - T
    u_unsat = kp*e + ki*I
    u = np.clip(u_unsat, u_min, u_max)
    # Integrar sólo si no hay saturación que impida reducir el error
    if (u == u_unsat) or (u==u_min and e>0) or (u==u_max and e<0):
        dI = e
    else:
        dI = 0.0
    dT = -a*(T - T_amb) + b*u
    return [dT, dI]

def horno_PID(t, y):
    T, I = y
    e = T_d - T
    # Cálculo de u no implícito usando la identidad anterior
    num = kp_pid*e + ki_pid*I + kd_pid*a*(T - T_amb)
    denom = 1.0 + kd_pid*b
    u_unsat = num / denom
    u = np.clip(u_unsat, u_min, u_max)
    # Integrar sólo si no hay saturación que impida reducir el error
    if (u == u_unsat) or (u==u_min and e>0) or (u==u_max and e<0):
        dI = e
    else:
        dI = 0.0
    dT = -a*(T - T_amb) + b*u
    return [dT, dI]

# Simulaciones
t_span = (0, 200)
t_eval = np.linspace(t_span[0], t_span[1], 2000)
```

```

sol_p = solve_ivp(lambda t,y: horno_P(t,y,a,b,kp_p,T_d,T_amb), t_span, [T0], t_eval=t_eval)
sol_pi = solve_ivp(lambda t,y: horno_PI(t,y,a,b,kp_pi,ki_pi,T_d,T_amb), t_span, [T0, 0.0], t_eval=t_eval)
sol_pid = solve_ivp(horno_PID, t_span, [T0, 0.0], t_eval=t_eval)

# Graficar comparativa
fig = plt.figure(figsize=(7,7))
plt.plot(sol_p.t, sol_p.y[0], label=f'P kp={kp_p}')
plt.plot(sol_pi.t, sol_pi.y[0], label=f'PI kp={kp_pi}, ki={ki_pi}')
plt.plot(sol_pid.t, sol_pid.y[0], label=f'PID kp={kp_pid}, ki={ki_pid}, kd={kd_pid}')
plt.axhline(T_d, color='red', linestyle='--', label='T deseada')
plt.axhline(T_amb, color='gray', linestyle='--', label='T ambiente')
plt.xlabel('Tiempo')
plt.ylabel('Temperatura (°C)')
plt.title('Comparativa: P vs PI vs PID')
plt.legend()
plt.grid(True)
plt.close(fig)

```

Final (sim) P kp=1.0: 70.7927 °C

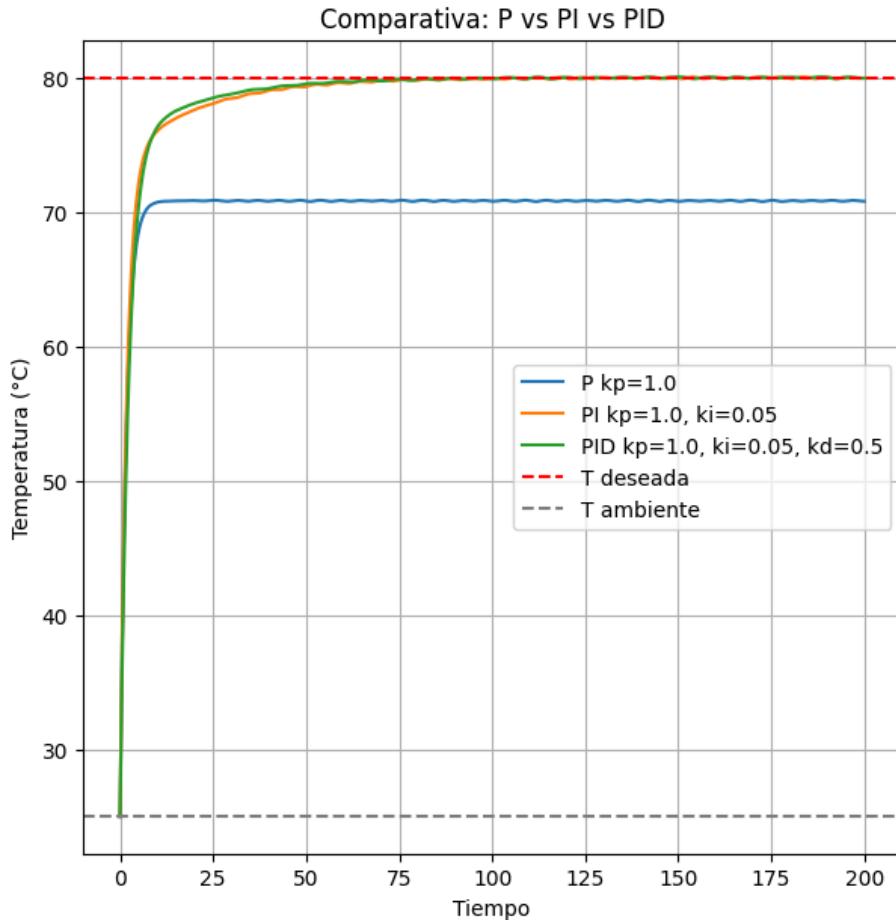
Final (sim) PI kp=1.0, ki=0.05: 79.9741 °C

Final (sim) PID kp=1.0, ki=0.05, kd=0.5: 79.9514 °C

$u_{\text{final}} (\text{P}) \approx 9.2073$

$u_{\text{final}} (\text{PI}) \approx 11.0228$

$u_{\text{final}} (\text{PID}) \approx 11.0321$



Este código implementa y compara tres tipos de controladores para el horno eléctrico: Proporcional (P), Proporcional-Integral (PI) y Proporcional-Integral-Derivativo (PID). El controlador P simplemente usa una ganancia proporcional ($kp=1.0$) que multiplica al error; el PI añade un término integral ($ki=0.05$) que acumula el error a lo largo del tiempo para eliminarlo en estado estacionario; y el PID incluye además un término derivativo ($kd=0.5$) que ayuda a mejorar la respuesta transitoria. El código incluye saturación del actuador (limita la potencia entre 0 y 100) y un mecanismo que evita que el término integral crezca indefinidamente cuando el actuador está saturado. La simulación muestra cómo el controlador P

no logra alcanzar exactamente la temperatura deseada (80°C), mientras que el PI y PID sí lo consiguen gracias al término integral, con el PID ofreciendo una respuesta ligeramente más rápida debido al término derivativo. Al final, el código imprime las temperaturas finales alcanzadas y las potencias de control aplicadas por cada controlador en estado estacionario.

De hecho, en los resultados numéricos, podemos observar claramente las diferencias entre los tres controladores. El control proporcional (P) con $k_p=1.0$ se queda corto, alcanzando solo 70.79°C frente a los 80°C deseados, con un error de aproximadamente 9.2°C y una potencia final de 9.20 unidades. En contraste, tanto el PI como el PID logran prácticamente alcanzar la temperatura objetivo (79.97°C y 79.95°C respectivamente), con errores menores a 0.05°C . Cabe destacar que ambos controladores PI y PID requieren una potencia final algo mayor (11.02 y 11.03 unidades) que el controlador P, lo cual es necesario para vencer las pérdidas térmicas y mantener la temperatura deseada. La similitud entre los resultados del PI y PID sugiere que, para este sistema, el término derivativo no aporta una mejora significativa en el estado estacionario, aunque puede haber influido en la velocidad de respuesta durante el transitorio.

▼ Práctica 2 - Parte 4: Robot móvil en el plano

En esta parte de la práctica se estudiará el modelo cinemático de un robot móvil de tipo diferencial. El objetivo es representar la evolución de su trayectoria en el plano XY, mostrar su orientación con una flecha y diseñar una ley de control para alcanzar un objetivo (goal).

Enunciado general

1. Implementa en Python el modelo cinemático de un robot móvil en el plano: $\dot{x} = v \cos \theta$, $\dot{y} = v \sin \theta$, $\dot{\theta} = \omega$ donde $u(t) = (v, \omega)$ son las entradas de control (velocidad lineal y angular).
2. Simula la evolución del robot para una entrada constante $u(t)$, de modo que la trayectoria sea circular.
3. Representa gráficamente la posición (x, y) del robot en el plano y su orientación con una flecha. Añade también un punto de color distinto que represente el objetivo (goal) en el plano.
4. Implementa una ley de control para llevar al robot a una pose deseada (x_d, y_d, θ_d) y verifica el resultado.
5. Repite el proceso para distintos objetivos y compara los resultados.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import ipywidgets as widgets
from ipywidgets import interact, FloatSlider
```

▼ Ejercicio 1

Simula el modelo del robot móvil con entradas constantes v y ω y representa su trayectoria en el plano XY. Muestra también su orientación con una flecha y dibuja un objetivo (goal) en el plano, que se pueda modificar de manera interactiva.

▼ Explicación del código

1. Definición del modelo:

Se define una función `robot_model` que describe la cinemática de un robot móvil. Cuenta con las entradas de velocidad lineal v y angular w . Además se calculan los errores correspondientes a la posición y orientación (ρ , α , β) respecto al objetivo (x_d , y_d , $\theta_{d\text{ad}}$) y se aplica la ley de control para ajustar la velocidad y orientación del robot según la proximidad y el ángulo respecto al objetivo:

- $v = k_{\rho} * \rho$
- $w = k_{\alpha} * \alpha + k_{\beta} * \beta$

2. Parámetros:

- La simulación comienza desde la posición inicial $(x_0, y_0, \theta_0) = (0, 0, 0)$.
- Las ganancias del controlador serán:
 - $k_{\rho} = 0.8$ (controla velocidad hacia el objetivo)
 - $k_{\alpha} = 2.0$ (ajusta giro hacia el objetivo)
 - $k_{\beta} = -1.0$ (controla la orientación final correcta)

3. Representación gráfica:

Utilizando matplotlib se genera una gráfica 2D (x, y). La línea azul muestra la trayectoria del robot, el punto verde indica el objetivo final y la flecha roja muestra la orientación final del robot al llegar al objetivo.

4. Interactividad:

Usando `interact` y `FloatSlider` de `ipywidgets` se puede modificar interactivamente x_d , y_d y $\theta_{d\text{ad}}$; la gráfica se actualiza cada vez que cambias los sliders.

```
# Modelo del robot móvil
def robot_model(t, state, v, omega):
    x, y, theta = state
    dxdt = v * np.cos(theta)
    dydt = v * np.sin(theta)
    dthetadt = omega
    return [dxdt, dydt, dthetadt]

# Parámetros iniciales
state0 = [0, 0, 0] # (x0, y0, theta0)
t_span = (0, 20)
t_eval = np.linspace(t_span[0], t_span[1], 500)

# Entradas constantes para movimiento circular
v = 1.0
omega = 0.5

# Simulación
sol = solve_ivp(robot_model, t_span, state0, t_eval=t_eval, args=(v, omega))

# Función de representación con goal
def plot_robot_with_goal(x, y, theta, goal=(5,5)):
```

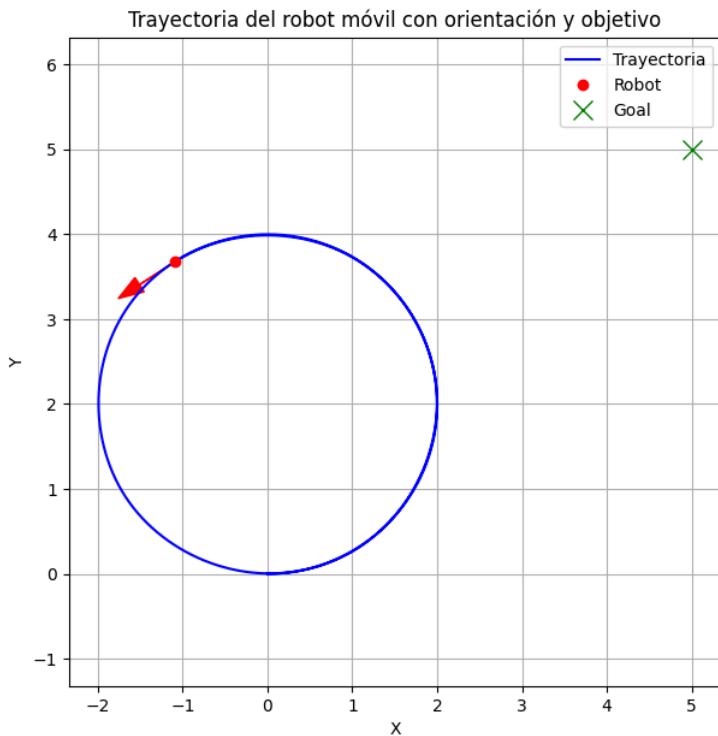
```

plt.figure(figsize=(7,7))
plt.plot(sol.y[0], sol.y[1], 'b-', label="Trayectoria")
plt.plot(x, y, 'ro', label="Robot")
plt.arrow(x, y, 0.5*np.cos(theta), 0.5*np.sin(theta), head_width=0.2, color='r')
plt.plot(goal[0], goal[1], 'gx', markersize=12, label="Goal")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Trayectoria del robot móvil con orientación y objetivo")
plt.legend()
plt.axis("equal")
plt.grid(True)
plt.show()

# Interfaz interactiva para mover el goal
interact(
    lambda goal_x, goal_y: plot_robot_with_goal(
        sol.y[0][-1], sol.y[1][-1], sol.y[2][-1], goal=(goal_x, goal_y)
    ),
    goal_x=FloatSlider(min=-10, max=10, step=0.5, value=5, description='Goal X'),
    goal_y=FloatSlider(min=-10, max=10, step=0.5, value=5, description='Goal Y')
)

# Ejemplo de visualización con goal (5,5)
#plot_robot_with_goal(sol.y[0][-1], sol.y[1][-1], sol.y[2][-1], goal=(5,5))

```



▼ Ejercicio 2

Implementa una **ley de control** que lleve al robot desde su estado inicial $x(0) = (0, 0, 0)$ hasta una pose deseada $goal = (x_d, y_d, \theta_d)$. Simula el comportamiento y representa gráficamente los resultados como en el ejemplo anterior. Incluye interacción con (x_d, y_d, θ_d) .

▼ Explicación del código

1. Definición del modelo:

Se define una función `robot_model` que describe la cinemática de un robot móvil.

Cuenta con las entradas de velocidad lineal **v** y angular **w**.

Además, se calculan los errores correspondientes a la posición y orientación (**rho**, **alpha**, **beta**) respecto al objetivo (**xd**, **yd**, **thetad**) y se aplica la ley de control para ajustar la velocidad y orientación del robot según la proximidad y el ángulo respecto al objetivo:

- $v = k_rho * rho$
- $w = k_alpha * alpha + k_beta * beta$

2. Parámetros:

- La simulación comienza desde la posición inicial $(x_0, y_0, \theta_0) = (0, 0, 0)$.
- Las ganancias del controlador serán:
 - $k_rho = 0.8$ (controla la velocidad hacia el objetivo)

- $k_{\alpha} = 2.0$ (ajusta el giro hacia el objetivo)
- $k_{\beta} = -1.0$ (controla la orientación final correcta)

3. Representación gráfica:

Utilizando la librería *matplotlib* se genera una gráfica en dos dimensiones (x, y).

La línea azul representa la trayectoria que recorre el robot, el punto verde indica el objetivo final (o **goal**) y la flecha roja indica la orientación final que presenta el robot al llegar al punto objetivo.

4. Interactividad:

Utilizando *interact* y *FloatSlider* de la librería *ipywidgets* se permite la modificación interactiva de los parámetros de la posición en x (**xd**), en y (**yd**) y la orientación final deseada (**thetad**).

```
# Modelo
def robot_model(t, state, x_d, y_d, theta_d, k_rho, k_alpha, k_beta):
    x, y, theta = state

    # Errores
    dx = x_d - x
    dy = y_d - y
    rho = np.sqrt(dx**2 + dy**2)
    alpha = np.arctan2(dy, dx) - theta
    alpha = np.arctan2(np.sin(alpha), np.cos(alpha)) # normalización [-pi, pi]
    beta = theta_d - theta - alpha
    beta = np.arctan2(np.sin(beta), np.cos(beta))

    # Ley de control
    v = k_rho * rho
    omega = k_alpha * alpha + k_beta * beta

    # Dinámica
    dxdt = v * np.cos(theta)
    dydt = v * np.sin(theta)
    dthetadt = omega
    return [dxdt, dydt, dthetadt]
```

```
# Simulación y visualización
def simulate_and_plot(x_d, y_d, theta_d):
    state0 = [0, 0, 0]
    t_span = (0, 25)
    t_eval = np.linspace(t_span[0], t_span[1], 1000)

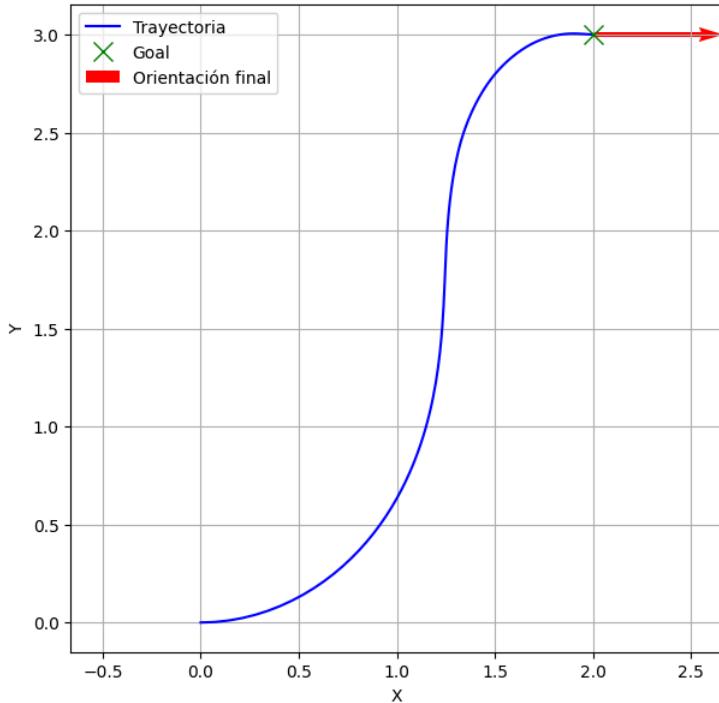
    # Ganancias
    k_rho, k_alpha, k_beta = 0.8, 2.0, -1.0

    sol = solve_ivp(robot_model, t_span, state0, t_eval=t_eval,
                    args=(x_d, y_d, theta_d, k_rho, k_alpha, k_beta))

    fig = plt.figure(figsize=(7,7))
    plt.plot(sol.y[0], sol.y[1], 'b-', label='Trayectoria')
    plt.plot(x_d, y_d, 'gx', markersize=12, label='Goal')
    plt.quiver(sol.y[0][-1], sol.y[1][-1],
               np.cos(sol.y[2][-1]), np.sin(sol.y[2][-1]),
               scale=5, color='r', label='Orientación final')
    plt.xlabel('X'); plt.ylabel('Y')
    plt.title('Control de robot diferencial hacia el objetivo')
    plt.legend()
    plt.axis('equal')
    plt.grid(True)

    # No mostrar ni guardar la figura
    plt.close(fig)
```

Control de robot diferencial hacia el objetivo



▼ Preguntas de reflexión

1. ¿Qué diferencias se observan entre el movimiento circular con $u = (v, \omega)$ constante y el movimiento con ley de control?

En el primer caso, para entradas constantes v y ω se observa un movimiento circular, donde el robot móvil describe esta trayectoria sin importar la ubicación del objetivo, simplemente siguiendo su cinemática. Por el contrario, en el caso de la implementación de una ley de control junto con entradas v y ω que experimentan cambios de forma dinámica, es posible observar una gran diferencia en el comportamiento del robot móvil, pasando de un movimiento predeterminado a un movimiento dirigido y enfocado en llegar al punto objetivo, variando en cada iteración los valores de v y ω según el error respecto al objetivo y ajustándose cada vez más.

2. ¿Cómo influye la elección de las ganancias en la ley de control ($k_\rho, k_\alpha, k_\beta$) en el comportamiento del robot?

En primer lugar, el caso de k_ρ sería aquella encargada de controlar la velocidad lineal hacia el objetivo, siendo que si esta es muy alta sería posible pasarse el objetivo, en contraste con el caso contrario donde adopta valores más bajos, donde si bien el movimiento es más lento también es más estable. Continuando, en el caso de k_α , es posible definirla como la responsable de controlar el giro a la hora de ser necesario orientarse el robot, donde cuanto mayor sea su valor, más buscado será el giro del robot hacia el objetivo al corregir la orientación hacia el mismo. Por último, con el caso de k_β , se apreciará su influencia en el ajuste final de la orientación, caso en el cual será necesario que tome valores negativos para poder garantizarse la estabilidad. Dentro de esta condición, se distinguirá para valores muy altos la posibilidad de realizar giros excesivos, mientras que para valores más reducidos no se llegará a conseguir una alineación correcta en su totalidad.

3. ¿Por qué cambia la trayectoria si solamente modificamos θ_d ?

Variando únicamente el valor de θ_d no afecta en sí al punto objetivo, pero sí influye en la posición final a la que apunta el robot. Esto es, una vez el robot alcanza el punto objetivo, al final este ajusta su trayectoria y gira hasta alcanzar la orientación que se deseé, por lo que incluso si el objetivo es el mismo, el recorrido final realizado por el robot si que cambia en función de la orientación que se deseé conseguir. Este sería el cambio que se percibe al modificar el valor de θ_d , lo que también refleja el control del sistema no únicamente de la posición sino también de la orientación.

4. ¿Qué ocurre si el objetivo (goal) se sitúa muy lejos? ¿Y si está muy cerca?

En el primer caso, donde el objetivo se encuentra muy lejos, ρ presentará un error grande, afectando directamente a que v sea muy grande (dado que $v = k_\rho \rho$). Este hecho se verá reflejado en el comportamiento del robot, que avanzará más rápido y trazaría una trayectoria más curva en el recorrido inicial mientras trata de corregir la orientación. En el caso contrario, donde el objetivo se encuentra muy cerca del robot, se podrá percibir un comportamiento diferenciado. Como se comentó anteriormente, ρ afectará directamente a v , que en este caso será pequeña. Con esto, el robot se moverá más lentamente y de forma más suave, apreciándose un movimiento mínimo para este caso.

5. ¿Cómo se relaciona este ejemplo con los conceptos de controlabilidad y estabilidad vistos en teoría?

Se define como controlable a un sistema si, partiendo desde cualquier estado inicial, es posible encontrar una señal de control que en un tiempo finito lleve al sistema a un estado final. La medida en la que esta definición se relaciona con este ejemplo tratado en el ejercicio 2 sería que el robot cuyo movimiento se estudia es controlable, ya que dadas dos entradas, en este caso v y w , es posible alcanzar cualquier pose. En otras palabras, el robot es capaz de moverse desde un estado inicial a cualquier otro mediante la señal de control adecuada.

Por otro lado, en cuanto a la estabilidad es necesario analizar la ley de control implementada, donde se garantiza que el error en la posición y en la orientación (e_x, e_y, e_θ) de un objeto se reduce a 0 conforme transcurre el tiempo, lo que implica que el sistema alcanza el punto de equilibrio al llegar a la pose deseada, sería asintóticamente estable entorno al objetivo. Este comportamiento sigue la definición