Tarea #4 Física numérica

Oscar Andrés Valencia Magaña

3 de noviembre de 2025

Índice

1. Lanzamiento del martillo **1**

2. Oscilador armónico acoplado **4**

3. Vibración de una cuerda **7**

A. Código para el lanzamiento del martillo **11**

B. Código para el oscilador acoplado **16**

C. Código para la vibración de una cuerda **21**

Índice de figuras

1. Trayectoria del martillo sin fricción 2
2. Trayectoria del martillo en flujo laminar 3
3. Trayectoria del martillo en flujo inestable oscilante 3
4. Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad hacia la derecha 5
5. Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad en sentidos opuestos 6
6. Una masa parte de su posición de equilibrio y la otra de una posición desplazada hacia la derecha 7
7. Diagrama del sistema de dos masas acopladas por resortes 7

Introducción

Buscamos presentar la solución a tres problemas físicos: el movimiento de un proyectil en un medio viscoso, un sistema de osciladores acoplados y una cuerda vibrante. En este trabajo se abordarán tanto las soluciones analíticas (si las hay, sino solo se deducirán las ecuaciones) como las soluciones numéricas y las gráficas obtenidas mediante programas desarrollados en Python.

1. Lanzamiento del martillo

Planteamiento:

El récord mundial para hombres en lanzamiento de martillo es de 86,74 m, establecido por Yuri Sedykh y vigente desde 1986. El martillo tiene una masa de 7,26 kg, es esférico y posee un radio de $R = 6$ cm.

La fricción sobre el martillo puede considerarse proporcional al cuadrado de la velocidad relativa al aire:

$$F_D = \frac{1}{2}\rho A C_D v^2$$

donde ρ es la densidad del aire (1,2 kg/m^3) y $A = \pi R^2$ es la sección transversal del martillo.

El martillo puede experimentar, en principio, un flujo laminar con coeficiente de rozamiento $C_D = 0{,}5$ o un flujo inestable oscilante con $C_D = 0{,}75$.

(a) Resuelva la ecuación de movimiento para el lanzamiento oblicuo del martillo. Deberá transformar las EDO correspondientes a los movimientos en x y y en un sistema de cuatro ecuaciones de primer orden. Considere lanzamientos desde una posición inicial $x_0 = 0$ y $y_0 = 2$ m, para un ángulo ideal $\theta = 45°$, y determine la velocidad que produce la distancia del lanzamiento del récord mundial.
Solución:

Consideremos $\vec{r} = (x, y)$ y $\dot{\vec{r}} = \vec{v} = (v_x, v_y)$. Según la mecánica newtoniana, la ecuación de movimiento está dada por:

$$m\ddot{\vec{r}} = m\vec{g} - \vec{F_D} = m\vec{g} - \frac{1}{2}\rho A C_D v^2 \hat{v} = m\vec{g} - \frac{1}{2}\rho A C_D v\, \vec{v},$$

donde $\vec{g} = (0, -g)$ y $v = |\vec{v}| = \sqrt{v_x^2 + v_y^2}$.
Desarrollando las componentes obtenemos:

$$m\dot{v}_x = -\frac{1}{2}\rho A C_D\, v\, v_x,$$

$$m\dot{v}_y = -mg - \frac{1}{2}\rho A C_D\, v\, v_y.$$

Por lo tanto, el sistema de ecuaciones diferenciales de primer orden queda expresado como:

$$\begin{aligned}
\dot{x} &= v_x, \\
\dot{y} &= v_y, \\
\dot{v}_x &= -\frac{1}{2m}\rho A C_D\, v\, v_x, \\
\dot{v}_y &= -g - \frac{1}{2m}\rho A C_D\, v\, v_y.
\end{aligned}$$

Con las condiciones iniciales:

$$\begin{aligned}
x(0) &= 0, \\
y(0) &= 2 \text{ m}, \\
v_x(0) &= v_0 \cos\theta, \\
v_y(0) &= v_0 \sin\theta,
\end{aligned}$$

donde v_0 es la velocidad inicial que se debe determinar para alcanzar la distancia del récord mundial, y $\theta = 45° = \frac{\pi}{4}$.
Este sistema es no lineal debido a la presencia del término $v = \sqrt{v_x^2 + v_y^2}$ en las ecuaciones para $\dot{v}_x$ y $\dot{v}_y$. Por lo tanto, no existe una solución analítica cerrada, y es necesario recurrir a métodos numéricos para resolverlo.
Proponemos utilizar la librería `scipy` para resolver el sistema de ecuaciones diferenciales, en particular, la función `odeint` para realizar la integración temporal del sistema.
La estrategia para encontrar la velocidad inicial v_0 que produce la distancia del récord mundial es la siguiente:

- Definir una función que resuelva el sistema de ecuaciones diferenciales para un valor dado de v_0 y que retorne la distancia horizontal del lanzamiento.
- Emplear un método de búsqueda de raíces (Newton-Raphson) para hallar el valor de v_0 que hace que la distancia obtenida sea igual a 86,74 m.

Implementando este procedimiento en `Python`, se obtiene que la velocidad inicial necesaria para alcanzar la distancia del récord mundial es aproximadamente:

$$\begin{aligned}
v_0 &\approx 28{,}84 \text{ m/s} && \text{(sin fricción, } C_D = 0), \\
v_0 &\approx 29{,}31 \text{ m/s} && \text{(flujo laminar, } C_D = 0{,}5), \\
v_0 &\approx 29{,}54 \text{ m/s} && \text{(flujo inestable oscilante, } C_D = 1{,}0).
\end{aligned}$$

El código utilizado para estos cálculos se presenta en los apéndices de esta tarea.

(b) Calcule y grafique la dependencia temporal de la altitud del martillo y su trayectoria $y = y(x)$ en los tres regímenes:
   i. Sin fricción

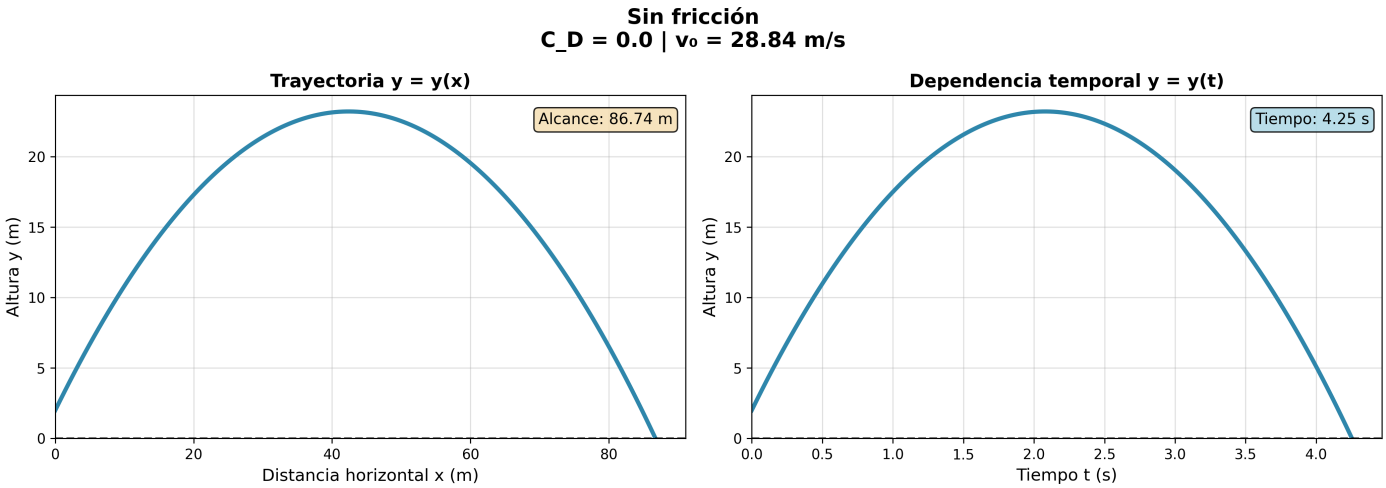

Figura 1: Trayectoria del martillo sin fricción

II. Flujo laminar

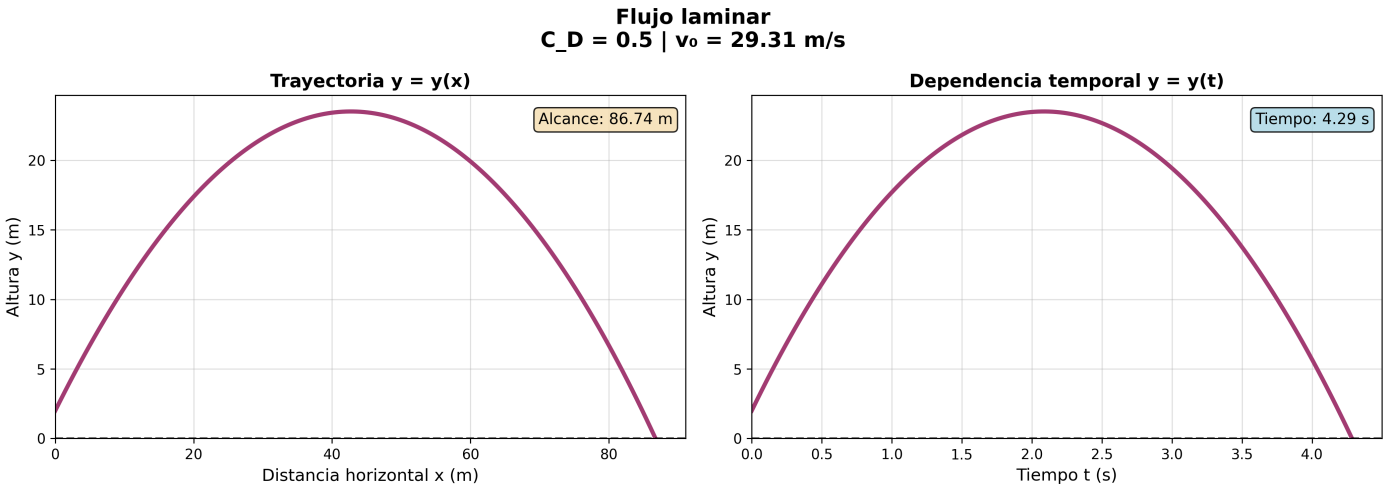

Figura 2: Trayectoria del martillo en flujo laminar

III. Flujo inestable oscilante

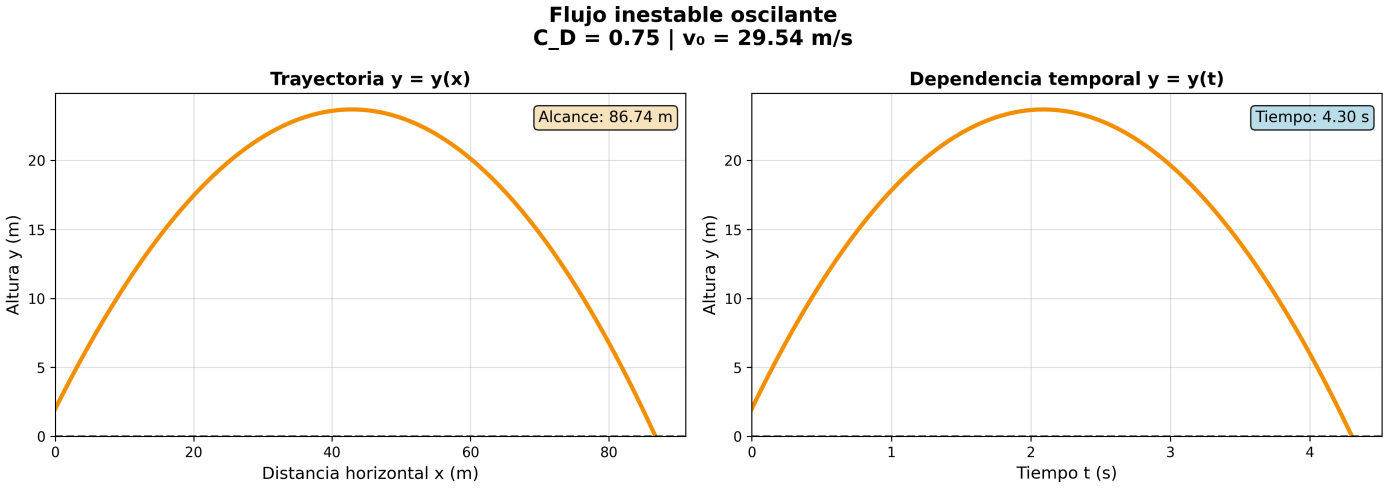

Figura 3: Trayectoria del martillo en flujo inestable oscilante

(c) En el inciso anterior, estime en qué medida la fricción influye en la distancia del lanzamiento.
Solución:
La fricción del aire tiene un impacto significativo en la distancia del lanzamiento del martillo. A continuación, se presenta un análisis detallado de la influencia de la fricción en los diferentes regímenes de flujo:

Régimen	Distancia (m)	Pérdida (m)	Pérdida (%)
Sin fricción	86.74	0.00	0.00 %
Flujo laminar	84.15	2.59	2.99 %
Flujo inestable oscilante	82.92	3.82	4.41 %

Cuadro 1: Influencia de la fricción en la distancia del lanzamiento del martillo

Como se observa en la Tabla 1, la fricción del aire reduce la distancia del lanzamiento en ambos regímenes considerados. En el caso del flujo laminar, la distancia se reduce en aproximadamente 2,59 m, lo que representa una pérdida del 2,99 % respecto al caso sin fricción. En el régimen de flujo inestable oscilante, la reducción es aún mayor, con una pérdida de 3,82 m o un 4,41 %. Esto indica que la fricción del aire tiene un efecto considerable en la trayectoria del martillo, especialmente en condiciones de flujo más turbulento.
Por lo que si resumimos nuestros resultados obtenidos tenemos que:

Régimen	Velocidad inicial (m/s)	Distancia (m)	Perdida (m)
Sin fricción	28.84	86.74	0.00
Flujo laminar	29.31	84.15	2.59
Flujo inestable oscilante	29.54	82.92	3.82

Cuadro 2: Velocidades iniciales y distancias alcanzadas en diferentes regímenes de fricción

De la Tabla 2, podemos concluir que: la fricción del aire reduce el alcance hasta en 4.4 %, lo que representa una pérdida máxima de 3.82 metros.

2. Oscilador armónico acoplado

Considere el sistema de resortes que se muestra en la figura 7.

Sea m la masa de cada bloque (ambas iguales) y supóngase que los resortes lineales tienen constantes elásticas k (resortes exteriores) y k_c (resorte central de acoplamiento), salvo que se indique lo contrario. Denote por $x_1(t)$ y $x_2(t)$ los desplazamientos horizontales de las masas respecto a sus posiciones de equilibrio.

(a) Escriba las ecuaciones de movimiento acopladas para los desplazamientos $x_1(t)$ y $x_2(t)$. Exprese las EDOs en su forma habitual y, a continuación, transforme el sistema a un conjunto equivalente de cuatro ecuaciones de primer orden adecuado para integración numérica.

Solución:

Sea m la masa de cada bloque, k la constante de los resortes exteriores y k_c la constante del resorte central de acoplamiento. Denotemos por $x_1(t)$ y $x_2(t)$ los desplazamientos horizontales de las masas respecto a sus posiciones de equilibrio. Tomando como punto de partida el Lagrangiano

$$\mathscr{L} = \tfrac{1}{2}m\big(\dot{x}_1^2 + \dot{x}_2^2\big) - \tfrac{1}{2}\big(kx_1^2 + k_c(x_2 - x_1)^2 + kx_2^2\big),$$

y aplicando las ecuaciones de Euler–Lagrange, se obtiene lo siguiente.

Ecuaciones de movimiento acopladas.

Calculando $\partial\mathcal{L}/\partial x_i$ y $\partial\mathcal{L}/\partial\dot{x}_i$ y usando $\dfrac{d}{dt}\big(\partial\mathcal{L}/\partial\dot{x}_i\big) - \partial\mathcal{L}/\partial x_i = 0$, llegamos a las ecuaciones:

$$m\ddot{x}_1 + (k + k_c)\,x_1 - k_c\,x_2 = 0,$$
$$m\ddot{x}_2 - k_c\,x_1 + (k + k_c)\,x_2 = 0.$$

Estas son las ecuaciones acopladas de segundo orden para los desplazamientos $x_1(t)$ y $x_2(t)$. Partiendo de dichas ecuaciones de movimiento acopladas es conveniente escribirlas en la forma habitual:

$$\ddot{x}_1 = -\frac{k + k_c}{m}\,x_1 + \frac{k_c}{m}\,x_2, \qquad \ddot{x}_2 = \frac{k_c}{m}\,x_1 - \frac{k + k_c}{m}\,x_2.$$

Para integrarlas numéricamente las transformamos a un sistema equivalente de cuatro ecuaciones de primer orden. Definimos las variables de estado

$$r_1 = x_1, \quad r_2 = x_2, \quad r_3 = \dot{x}_1, \quad r_4 = \dot{x}_2.$$

Luego entonces el sistema de primer orden queda:

$$\dot{r}_1 = r_3,$$
$$\dot{r}_2 = r_4,$$
$$\dot{r}_3 = -\frac{k + k_c}{m}\,r_1 + \frac{k_c}{m}\,r_2,$$
$$\dot{r}_4 = \frac{k_c}{m}\,r_1 - \frac{k + k_c}{m}\,r_2.$$

En forma matricial puede escribirse como $\dot{\mathbf{r}} = \mathbf{A}\mathbf{r}$, con

$$\mathbf{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}, \qquad \mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\dfrac{k+k_c}{m} & \dfrac{k_c}{m} & 0 & 0 \\ \dfrac{k_c}{m} & -\dfrac{k+k_c}{m} & 0 & 0 \end{pmatrix}.$$

Este sistema de cuatro ecuaciones de primer orden es directamente integrable con cualquier método estándar, usando condiciones iniciales.

(b) Calcule las frecuencias de los modos normales de vibración del sistema (modo simétrico y modo antisimétrico), y obtenga las correspondientes relaciones entre amplitudes X_1 y X_2 para cada modo.
Solución:
Dado que la finalidad del curso es aprender a aplicar métodos númericos, unicamente explicaremos lo solicitado, en el apedice agregaremos el codigo para el calculo.
Buscamos soluciones armónicas de la forma

$$x_1(t) = X_1 e^{i\omega t}, \qquad x_2(t) = X_2 e^{i\omega t}.$$

Sustituyendo en las ecuaciones de movimiento se obtiene el sistema algebraico

$$\begin{pmatrix} k + k_c - m\omega^2 & -k_c \\ -k_c & k + k_c - m\omega^2 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \mathbf{0}.$$

Para que existan soluciones no triviales debe cumplirse $\det(\cdot) = 0$, es decir

$$\left(k + k_c - m\omega^2\right)^2 - k_c^2 = 0.$$

De aquí se obtienen las dos ecuaciones para $m\omega^2$:

$$m\omega^2 = k \quad \text{y} \quad m\omega^2 = k + 2k_c.$$

Por tanto las frecuencias angulares de los modos normales son

$$\omega_1 = \sqrt{\frac{k}{m}} \quad \text{(modo simétrico)}, \qquad \omega_2 = \sqrt{\frac{k+2k_c}{m}} \quad \text{(modo antisimétrico)}.$$

Las correspondientes relaciones entre amplitudes (autovectores) son:

$$\omega_1^2 = \frac{k}{m} \;\Rightarrow\; X_1 = X_2 \quad \text{(las masas se mueven en fase)},$$

$$\omega_2^2 = \frac{k+2k_c}{m} \;\Rightarrow\; X_1 = -X_2 \quad \text{(las masas se mueven en contrafase)}.$$

(c) Grafique las posiciones de las masas en función del tiempo para las condiciones iniciales siguientes:

i. Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad hacia la derecha: $x_1(0) = x_2(0) = A, \quad \dot{x}_1(0) = \dot{x}_2(0) = 0.$

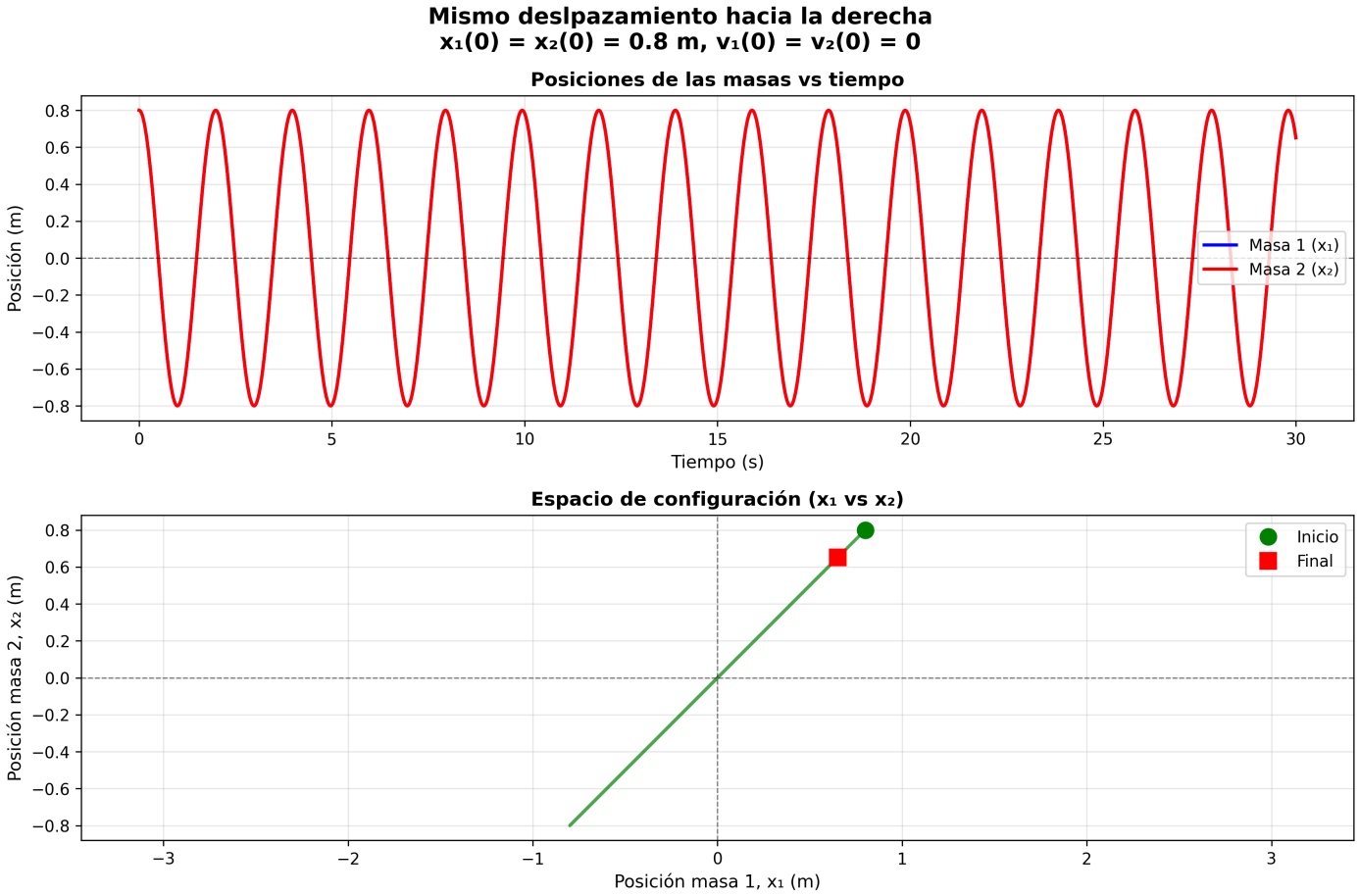

Figura 4: Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad hacia la derecha

II. Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad en sentidos opuestos: $x_1(0) = A,\ x_2(0) = -A, \quad \dot{x}_1(0) = \dot{x}_2(0) = 0.$

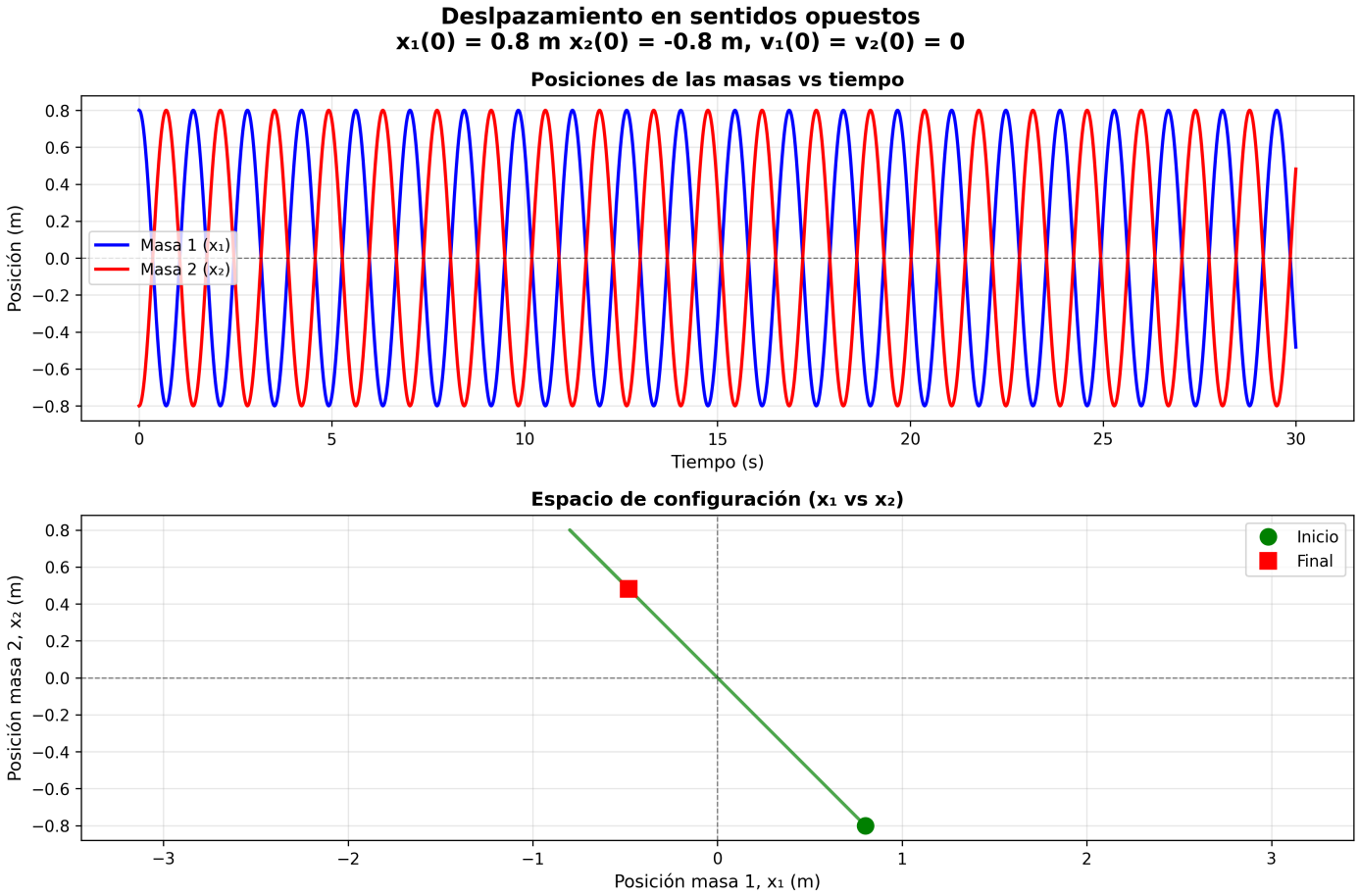

Figura 5: Ambas masas parten del reposo habiendo sido desplazadas la misma cantidad en sentidos opuestos

III. Una masa parte de su posición de equilibrio y la otra de una posición desplazada hacia la derecha: $x_1(0) = 0,\ x_2(0) = A, \quad \dot{x}_1(0) = \dot{x}_2(0) = 0.$

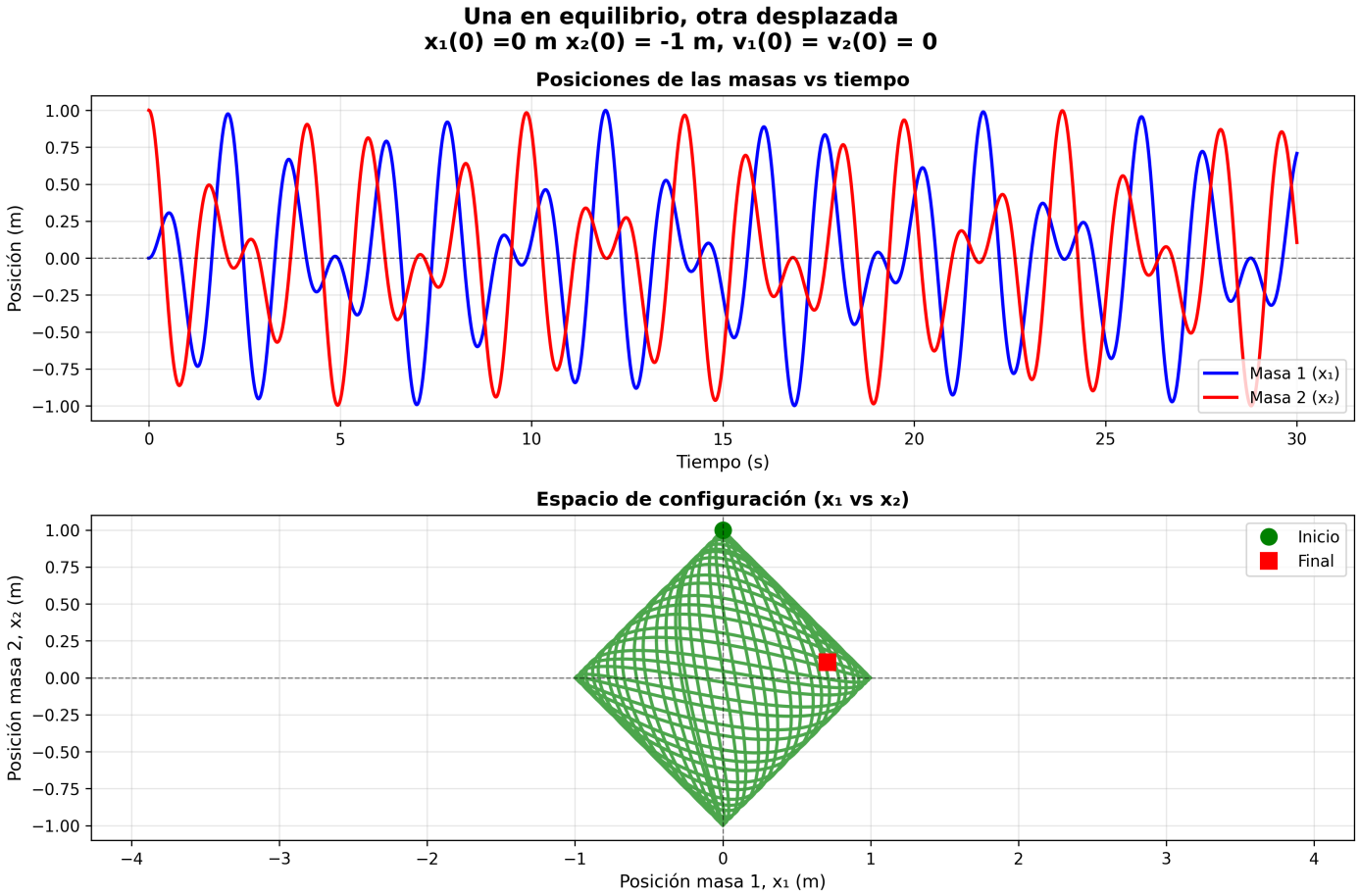

Figura 6: Una masa parte de su posición de equilibrio y la otra de una posición desplazada hacia la derecha

Para cada caso, muestre las curvas $x_1(t)$ y $x_2(t)$ y, cuando sea útil, represente la combinación en coordenadas normales.

(d) Suponga ahora que los resortes no son lineales y que la fuerza restauradora de cada resorte tiene la forma

$$F = -k\big(x + 0{,}1\,x^3\big).$$

Repita el procedimiento del inciso (b): determine (o estime) las frecuencias / comportamientos de oscilación y compare las respuestas del sistema lineal con las del sistema no lineal. Discuta las diferencias cualitativas y cuantitativas entre ambos casos (desplazamiento- dependiente de la frecuencia, aparición de armónicos, etc.).
Solución: Diferencias entre sistema lineal y no lineal:

- Sistema lineal: Oscilaciones armónicas puras (sinusoidales)
- Sistema no lineal: Distorsión de la forma de onda
- No linealidad introduce dependencia de amplitud en frecuencia
- Espacio de fase no lineal muestra trayectorias distorsionadas

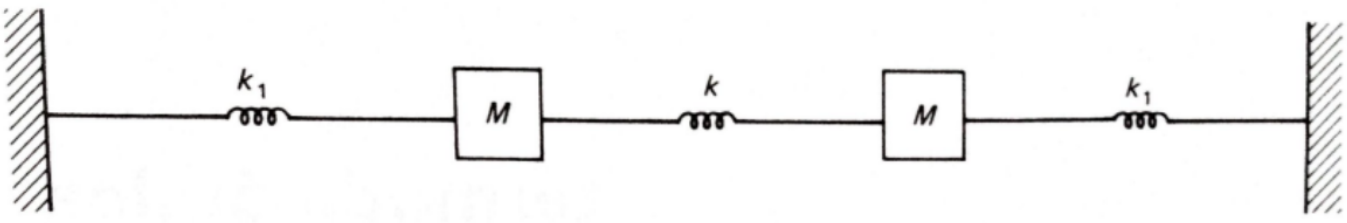

Figura 7: Diagrama del sistema de dos masas acopladas por resortes

3. Vibración de una cuerda

Oscilaciones de una cuerda

Considere una cuerda de longitud L y densidad lineal $\rho(x)$ (masa por unidad de longitud), sujeta en ambos extremos y bajo una tensión $T(x)$. Suponga que el desplazamiento transversal de la cuerda respecto a su posición de equilibrio,

$y(x,t)$, es pequeño y que la pendiente $\partial y/\partial x$ también es pequeña.

(a) Considere una sección infinitesimal de la cuerda entre x y $x+\Delta x$. Notando que la diferencia en las componentes horizontales y verticales de las tensiones produce una fuerza restauradora, demuestre que, aplicando la segunda ley de Newton a esta sección, se obtiene la ecuación

$$\frac{dT(x)}{dx}\,\frac{\partial y(x,t)}{\partial x} + T(x)\,\frac{\partial^2 y(x,t)}{\partial x^2} \;=\; \rho(x)\,\frac{\partial^2 y(x,t)}{\partial t^2}.$$

Demostración:

Consideremos un elemento de cuerda entre x y $x+\Delta x$. Las tensiones en los extremos tienen direcciones que forman ángulos pequeños con el eje x. Denotemos las componentes horizontales y verticales de la tensión en x por $T(x)\cos\theta(x)$ y $T(x)\sin\theta(x)$, y análogamente en $x+\Delta x$.

Para pequeñas pendientes $\theta \ll 1$ se tiene $\sin\theta \approx \theta \approx y_x$ y $\cos\theta \approx 1$. La fuerza neta vertical sobre el elemento proviene de la diferencia de las componentes verticales de la tensión más cualquier fuerza externa (aquí despreciada salvo la masa inercial):

$$F_{\text{vertical}} \simeq T(x+\Delta x)\,\sin\theta(x+\Delta x) - T(x)\,\sin\theta(x).$$

Aproximando para Δx pequeño y usando $\sin\theta \approx y_x$:

$$F_{\text{vertical}} \simeq \frac{d}{dx}\big(T(x)\,y_x(x,t)\big)\,\Delta x.$$

La masa del elemento es $\rho(x)\,\Delta x$. Aplicando la segunda ley de Newton en dirección vertical:

$$\rho(x)\,\Delta x\,y_{tt}(x,t) = \frac{d}{dx}\big(T(x)\,y_x(x,t)\big)\,\Delta x.$$

Cancelando Δx obtenemos la forma general:

$$\boxed{\frac{d}{dx}\big(T(x)\,y_x(x,t)\big) = \rho(x)\,y_{tt}(x,t)}.$$

(b) ¿Qué condiciones sobre $T(x)$ y $\rho(x)$ son necesarias para recuperar la ecuación de onda estándar

$$\frac{\partial^2 y(x,t)}{\partial x^2} \;=\; \frac{1}{c^2}\,\frac{\partial^2 y(x,t)}{\partial t^2}, \qquad c=\sqrt{\frac{T}{\rho}}\,?$$

Explique claramente las hipótesis de homogeneidad y constancia que se requieren.

Solución:

Para obtener la ecuación de onda clásica

$$y_{xx}(x,t) = \frac{1}{c^2}\,y_{tt}(x,t), \qquad c=\sqrt{\frac{T}{\rho}},$$

se requieren las siguientes condiciones:

- *Tensión constante:* $T(x)=T=$ constante.
- *Densidad lineal constante:* $\rho(x)=\rho=$ constante.
- *Pequeñas pendientes:* permite linearizar $\sin\theta\approx y_x$.
- *No fuerzas externas adicionales* (o incluidas explícitamente).

Bajo $T'=0$ y $\rho=$ const., la ecuación del apartado anterior se reduce a

$$T\,y_{xx} = \rho\,y_{tt} \quad\Longrightarrow\quad y_{tt} - \frac{T}{\rho}\,y_{xx} = 0,$$

es decir

$$\boxed{y_{xx} = \frac{1}{c^2}y_{tt}, \qquad c=\sqrt{\frac{T}{\rho}}\,.}$$

(c) ¿Qué condiciones deben imponerse (condiciones iniciales y de frontera) para que la EDP de segundo orden tenga una única solución?

Explicación:

La ecuación de onda es una ecuación en derivadas parciales lineal de segundo orden en tiempo y espacio. Para garantizar unicidad (y existencia) de la solución en un dominio $0\le x\le L,\ t\ge 0$ se necesitan:

a) *Condiciones iniciales* en $t=0$:

$$y(x,0)=f(x), \qquad y_t(x,0)=g(x), \qquad 0\le x\le L,$$

donde f y g son funciones suficientemente regulares (por ejemplo C^2).

b) *Condiciones de frontera* en $x = 0$ y $x = L$ para todo $t > 0$. Ejemplos típicos:

- *Dirichlet (extremos fijos):* $y(0,t) = 0,\ y(L,t) = 0.$
- *Neumann (extremos libres):* $y_x(0,t) = 0,\ y_x(L,t) = 0.$
- Condiciones mixtas o de tipo impedancia según el problema físico.

Con estas condiciones (basta con dos condiciones en t y dos en x apropiadas) la solución del problema de valor inicial y frontera es única por resultados estándar sobre la ecuación de onda (principio de energía, unicidad en problemas hiperbólicos).

(d) Utilice una malla en tiempo y espacio con pasos Δt y Δx. Denote

$$y(x_i, t_j) = y_{i,j}, \qquad x_i = i\,\Delta x, \quad t_j = j\,\Delta t.$$

Escriba la aproximación finita correspondiente para obtener una solución numérica $y_{i,j}$.

Explicación:
Tomemos una malla uniforme:

$$\Delta x = \text{paso espacial}, \qquad \Delta t = \text{paso temporal},$$
$$x_i = i\,\Delta x, \quad i = 0, 1, \ldots, N, \qquad t_j = j\,\Delta t, \quad j = 0, 1, 2, \ldots$$

Aproximamos la solución en la malla por

$$y(x_i, t_j) \approx y_i^j \quad \text{o} \quad y_i^j := y(i\Delta x,\ j\Delta t).$$

(e) Exprese las segundas derivadas temporales y espaciales en términos de diferencias finitas centradas y demuestre que, para el caso homogéneo (constantes T y ρ), esto conduce a la ecuación en diferencias

$$y_{i,j+1} - 2y_{i,j} + y_{i,j-1} \;=\; \frac{c^2(\Delta t)^2}{(\Delta x)^2}\big(y_{i+1,j} - 2y_{i,j} + y_{i-1,j}\big),$$

donde $c = \sqrt{T/\rho}$.
Explicación:
Aproximaciones centrales de segunda orden:

$$\frac{\partial^2 y}{\partial t^2}\Big|_{(x_i,t_j)} \approx \frac{y_i^{j+1} - 2y_i^j + y_i^{j-1}}{(\Delta t)^2},$$

$$\frac{\partial^2 y}{\partial x^2}\Big|_{(x_i,t_j)} \approx \frac{y_{i+1}^j - 2y_i^j + y_{i-1}^j}{(\Delta x)^2}.$$

Sustituyendo en la ecuación de onda continua $y_{tt} = c^2 y_{xx}$:

$$\frac{y_i^{j+1} - 2y_i^j + y_i^{j-1}}{(\Delta t)^2} = c^2 \frac{y_{i+1}^j - 2y_i^j + y_{i-1}^j}{(\Delta x)^2}.$$

Multiplicando por $(\Delta t)^2$ y reordenando obtenemos la forma en diferencias:

$$\boxed{y_i^{j+1} + y_i^{j-1} - 2y_i^j = c^2 \frac{(\Delta t)^2}{(\Delta x)^2}\big(y_{i+1}^j - 2y_i^j + y_{i-1}^j\big)}.$$

(f) De la ecuación anterior, despeche $y_{i,j+1}$ y escriba el algoritmo de obtención del paso temporal siguiente en la forma

$$y_{i,j+1} \;=\; 2y_{i,j} - y_{i,j-1} \;+\; \lambda^2\big(y_{i+1,j} - 2y_{i,j} + y_{i-1,j}\big),$$

donde $\lambda = c\,\dfrac{\Delta t}{\Delta x}$ es el número de Courant reducido (velocidad de la malla $c_0 = \Delta x/\Delta t$ implica $\lambda = c/c_0$).

Explicación:

Definimos el *número de Courant* (sinónimo aquí de razón adimensional):

$$\sigma \;:=\; \frac{c\,\Delta t}{\Delta x}.$$

También se puede definir la *velocidad de malla* c_0 como

$$c_0 := \frac{\Delta x}{\Delta t} \quad \Longrightarrow \quad \left(\frac{c}{c_0}\right)^2 = \sigma^2.$$

Reorganizando la ecuación del apartado (e) para despejar y_i^{j+1}:

$$y_i^{j+1} = 2y_i^j - y_i^{j-1} + \sigma^2\big(y_{i+1}^j - 2y_i^j + y_{i-1}^j\big).$$

Es decir, explicitamente:

$$\boxed{y_i^{j+1} \;=\; 2y_i^j - y_i^{j-1} + \left(\frac{c}{c_0}\right)^2 \left(y_{i+1}^j - 2y_i^j + y_{i-1}^j\right), \qquad c_0 = \frac{\Delta x}{\Delta t}.}$$

Esta es la fórmula de actualización explícita (esquema central en tiempo y espacio) que permite avanzar desde los dos niveles temporales j y $j-1$ al nivel $j+1$.

Inicialización (primer paso) Para iniciar el método se necesita y_i^0 (condición inicial de desplazamiento) y $y_t(x,0) = g(x)$ (velocidad inicial). Un esquema de orden dos para obtener y_i^1 es usar la expansión de Taylor:

$$y_i^1 = y_i^0 + \Delta t\, g_i + \frac{(\Delta t)^2}{2}\, c^2 \frac{y_{i+1}^0 - 2y_i^0 + y_{i-1}^0}{(\Delta x)^2},$$

es decir,

$$y_i^1 = y_i^0 + \Delta t\, g_i + \frac{\sigma^2}{2}\big(y_{i+1}^0 - 2y_i^0 + y_{i-1}^0\big).$$

(g) ¿Cómo se implementan las condiciones iniciales y de frontera en el esquema numérico? Especifique la forma de:

- condiciones de frontera fijas (extremos sujetos: $y_{0,j} = y_{N,j} = 0$),
- condiciones de frontera libres o de radiación (si procede),
- condiciones iniciales $y_{i,0}$ y $\dot{y}_{i,0}$ (desplazamiento y velocidad inicial).

Indique además cómo calcular $y_{i,1}$ (el primer paso en tiempo) a partir de $y_{i,0}$ y $\dot{y}_{i,0}$.
Explicación:

- **Condiciones iniciales:**
$$y_i^0 = f(x_i), \qquad \left.\frac{\partial y}{\partial t}\right|_{t=0}(x_i) = g(x_i)$$
y usar la fórmula de inicialización para obtener y_i^1.
- **Condiciones de frontera:** se imponen en cada paso temporal j.
  - Para extremos fijos (Dirichlet): $y_0^j = 0,\ y_N^j = 0$ para todo j.
  - Para extremos libres (Neumann): aproximar $y_x(0,t) = 0$ por diferencias centradas, por ejemplo $y_1^j - y_{-1}^j = 0$ (o usar frontera fantasma y eliminarla).
  - También son posibles condiciones de absorción o condiciones radiativas discretas.

(h) La condición de Courant para la estabilidad del esquema explícito es

$$\lambda = \frac{c\,\Delta t}{\Delta x} \le 1.$$

Explique qué significa esto en términos de los pasos Δt y Δx (interpretación física y numérica).
Explicación:
El criterio de estabilidad CFL (Courant–Friedrichs–Lewy) para este esquema explícito es:

$$\sigma = \frac{c\,\Delta t}{\Delta x} \le 1.$$

Equivalente:

$$\boxed{\Delta t \le \frac{\Delta x}{c}\,.}$$

Interpretación física: en términos de pasos, la condición dice que durante un paso temporal Δt la onda no debe viajar más de un intervalo espacial Δx. Si $c\,\Delta t > \Delta x$ la información física (propagación de la onda) avanzaría más allá de las celdas vecinas en un solo paso, lo que provoca inestabilidad numérica en el esquema explícito central. Por tanto, Δt debe ser lo suficientemente pequeño respecto a Δx para que la aproximación explícita sea estable.

En términos de c_0: como $c_0 = \Delta x/\Delta t$, la condición es $c/c_0 \leq 1$, o $c_0 \geq c$: la "velocidad de malla" debe ser al menos la velocidad física c.

Observaciones finales:

- El esquema descrito es de orden dos tanto en tiempo como en espacio.
- Para mayor estabilidad o para pasos de tiempo más grandes, se pueden usar esquemas implícitos o esquemas con amortiguamiento numérico; sin embargo, los explícitos son simples y eficientes siempre que se cumpla la condición CFL.
- En implementación numérica prestar atención al tratamiento de condiciones de frontera y al cómputo del primer paso y_i^1.

(i) Escriba un programa que implemente el esquema explicito anterior y produzca una animación del movimiento de la cuerda $y(x,t)$. Indique las decisiones prácticas importantes (elección de Δx, Δt, duración de la simulación, condiciones de frontera, normalización de ejes para la animación).
Explicación:
La codificacion de lo anterior viene en el apendice C

(j) Varíe los pasos Δt y Δx en su programa de modo que en algunos casos se cumpla la condición de Courant y en otros no. Describa y explique qué ocurre en cada caso (estabilidad numérica, propagación correcta de ondas cuando $\lambda \leq 1$; crecimiento no físico e inestabilidad cuando $\lambda > 1$).
Explicación:
Analogamente, en el código compartido en el apendice C vendrá incorporada la resolución.

A. Código para el lanzamiento del martillo

Tarea 4: Resolución del problema del martillo

```
from pylab import *
import numpy as np
from scipy.integrate import odeint
from matplotlib.animation import FuncAnimation
import os

# Parámetros físicos
g = 9.81  # Aceleración debido a la gravedad (m/s^2)
rho = 1.2  # Densidad del aire (kg/m^3)
R = 0.06  # Radio del martillo (m)
A = np.pi * R**2  # Área de sección transversal del martillo (m^2)
m = 7.26  # Masa del martillo (kg)
record_distance = 86.74  # Distancia del récord mundial (m)
theta = np.radians(45)  # Ángulo de lanzamiento (radianes)
x0, y0 = 0,  2  # Posición inicial (m)
it_max = 500  # Número máximo de iteraciones
dt = 0.01  # Paso de tiempo (s)
N= 500  # Número de pasos de tiempo
tol=1e-3  # Tolerancia para la convergencia
drag_coeffs = [0.0, 0.5, 0.75]  # Coeficientes de arrastre para los tres regímenes

# Crear carpeta para resultados
output_dir = 'resultados_martillo' # Carpeta para guardar resultados
if not os.path.exists(output_dir): # Crear carpeta si no existe
    os.makedirs(output_dir) # Crear carpeta si no existe


#definición de las ecuaciones de movimiento (EDOs)
def equations_of_motion(state, t, k):
    """Devuelve las derivadas de las variables de estado."""
    f0= state[1]
    f1= -k/m*state[1]*np.sqrt(state[1]**2 + state[3]**2)
    f2= state[3]
    f3 = -g - k/m*state[3]*np.sqrt(state[1]**2 + state[3]**2)
    return array([f0, f1, f2, f3])

#buscamos la distancia alcanzada para una velocidad inicial dada
def distance_reached(initial, v0, k):
```

```
    """Calcula la distancia alcanzada para una velocidad inicial dada y coeficiente de arrastre
    ↪ k."""
    #como theta = 45 grados, las componentes x e y de la velocidad inicial son iguales
    v= v0 * np.sin(pi/4)
    finaltime = 10.0 # Tiempo final para la simulación
    r0= array([initial[0],v , initial[1], v])  # Estado inicial: [x0, vx0, y0, vy0]
    r=r0 # variable para almacenar el estado actual
    t = linspace(0, it_max * dt, N)  # Vector de tiempo
    groundtime= 0.0

    #inicia lo dificil, jugar a adivinar con algo numerico
    s=0
    while s < it_max:
        # Integración numérica de las EDOs
        sol = odeint(equations_of_motion, r, t, args=(k,))
        n=len(sol)-1
        # Verificar si el martillo no ha tocado el suelo
        if sol[n, 2] > 0:
            finaltime += 1.0
            t= linspace(0., finaltime, N)
        else:
           for j in range(n):
               # verificamos si la fisica nos falla y fuimos capaces de atravesar el suelo
               if sol[j, 2] <= 0:
                   groundtime += t[j-1]-t[0]
                   #checamos la tolerancia
                   if abs(sol[j,2])<= tol/2.:
                       # buscamos que en caso de ser preciso, reacemos todo pero usando el tiempo
                       ↪  para que toque el suelo.
                       t = linspace(0, groundtime+t[j]-t[j-1], N*50*s)
                       sol = odeint(equations_of_motion, r0, t, args=(k,))
                       n=len(sol)-1
                       return (sol[n,0]+sol[n-1,0])/2., groundtime
                   else: #en caso de no ser preciso, resolvemos la ED con los valores inciales,
                   ↪  que seran las condiciones que el martillo llevaba antes de tocar el piso.
                        r = array([sol[j-1][0], sol[j-1][1],
                                   sol[j-1][2], sol[j-1][3]])
                        t=linspace(t[j-1], t[j], 50)
                        break
        s += 1
    print("Distancia recorrida antes de llegar al suelo no encontrada\
          dentro de las iteraciones permitidas. Regresamos 0.")
    return 0.,0.


# nos interesa hallar la velocidad inicial que produce la distancia del récord mundial
def find_initial_velocity(distance, initial, v0, k):
    """Encuentra la velocidad inicial que produce la distancia del récord mundial."""
    f= lambda v: distance_reached(initial, v, k)[0]- distance
    dv= 1.0e-3

    i=0

    #vamos a hacer Newton-Raphson para encontrar la velocidad inicial
    while i < it_max:
        fv = f(v0)
        if abs(fv) <= tol:
            return v0
        # Derivada numérica
        df = (f(v0 + dv/2.) - f(v0 - dv/2.)) / dv
        dv= -fv/df
        v0 += dv

        i += 1
```

```
    print("Velocidad inicial no encontrada dentro de las iteraciones permitidas.\
          Regresamos 0.")
    return 0.
#Tambien nos interesa la trayectoria :
def get_full_trajectory(initial, v0, k, tground):
    """Obtiene la trayectoria completa para graficar."""
    v = v0 * np.sin(np.pi/4)
    r0 = array([initial[0], v, initial[1], v])

    if tground == 0.:
        tground = 5.0

    t = linspace(0., tground, 500)
    sol = odeint(equations_of_motion, r0, t, args=(k,))

    return sol, t
#imprimmos las condiciones iniciales
print("\nCondiciones iniciales:")
print(f"  x0 = {x0} m, y0 = {y0} m, theta = 45")
print("\nEncontrando velocidad inicial para alcanzar récord mundial...")
print("="*70)

v_aux = [] # Lista para almacenar velocidades iniciales
for i, cd in enumerate(drag_coeffs):
    print(f"  * Régimen {i+1} (C_D = {cd}):")
    k = rho * A * cd / 2.0
    v0_record = find_initial_velocity(record_distance, array([x0, y0]), 28.0, k)
    v_aux.append(v0_record) # Agregar velocidad inicial encontrada a la lista --- nos serviran
    ↪ más adelante
    if v0_record == 0.:
        print("ERROR: No se pudo calcular la velocidad inicial.")
        exit(1)
    else:
        print(f"Velocidad inicial necesaria: v0 = {v0_record:.2f} m/s")
        print(f"(Para alcanzar {record_distance} m con C_D = {cd})")
        print("="*70)

print("\n" + "="*70)
print("TRAYECTORIAS EN LOS TRES REGÍMENES")
print("="*70)

solutions = [] # Almacenar soluciones para cada régimen
times_list = [] # Almacenar tiempos para cada régimen
distances_list = [] # Almacenar distancias alcanzadas
labels = ['Sin fricción', 'Flujo laminar', 'Flujo inestable oscilante'] # Etiquetas para los
↪ regímenes
cd_labels = ['C_D = 0.0', 'C_D = 0.5', 'C_D = 0.75'] # Etiquetas para los coeficientes de
↪ arrastre
colors = ['#2E86AB', '#A23B72', '#F18F01'] # Colores para las gráficas

print("\nCalculando trayectorias...")
for i, cd in enumerate(drag_coeffs): # Iterar sobre los coeficientes de arrastre
    k = rho * A * cd / 2.0
    distance, time = distance_reached(array([x0, y0]), v_aux[i], k) # Calcular distancia
    ↪ alcanzada
    distances_list.append(distance)     # Almacenar distancia alcanzada

    print(f"\n{labels[i]} ({cd_labels[i]}):")
    print(f"  Distancia alcanzada: {distance:.2f} m") # Almacenar distancia alcanzada
    print(f"  Tiempo de vuelo: {time:.2f} s") # Almacenar tiempo de vuelo
    print(f"  Velocidad inicial: v0 = {v_aux[i]:.2f} m/s")

    sol, t = get_full_trajectory(array([x0, y0]), v_aux[i], k, time) # Obtener trayectoria
    ↪ completa
```

```
    solutions.append(sol) # Almacenar solución
    times_list.append(t) # Almacenar tiempos
print("="*70)
print("Calculo de las trayectorias con la velocidad inicial dada con C_D = 0.0: v0 =",v_aux[0])
print("\nCalculando trayectorias...")
sols=[] # Almacenar soluciones para cada régimen
tims=[] # Almacenar tiempos para cada régimen
dists=[] # Almacenar distancias alcanzadas
for i, cd in enumerate(drag_coeffs): # Iterar sobre los coeficientes de arrastre
    k = rho * A * cd / 2.0
    distance, time = distance_reached(array([x0, y0]), v_aux[0], k) # Calcular distancia
    ↪ alcanzada
    dists.append(distance)     # Almacenar distancia alcanzada

    print(f"\n{labels[i]} ({cd_labels[i]}):")
    print(f"  Distancia alcanzada: {distance:.2f} m") # Almacenar distancia alcanzada
    print(f"  Tiempo de vuelo: {time:.2f} s") # Almacenar tiempo de vuelo

    sol, t = get_full_trajectory(array([x0, y0]), v_aux[0], k, time) # Obtener trayectoria
    ↪ completa
    sols.append(sol) # Almacenar solución
    tims.append(t) # Almacenar tiempos
print("="*70)
print(" GENERANDO GRÁFICAS POR RÉGIMEN...")
print("="*70)

for i, (sol, t, cd, label, cd_label, color, distance) in enumerate(
    zip(solutions, times_list, drag_coeffs, labels, cd_labels, colors, distances_list)):

    # Crear figura con 2 subplots
    fig, (ax1, ax2) = subplots(1, 2, figsize=(14, 5))
    fig.suptitle(f'{label}\n{cd_label} | v₀ = {v_aux[i]:.2f} m/s',
                 fontsize=15, fontweight='bold')

    # Subplot 1: Trayectoria y = y(x)
    ax1.plot(sol[:, 0], sol[:, 2], color=color, linewidth=3, label='Trayectoria')
    ax1.axhline(y=0, color='black', linewidth=1.5, linestyle='--', alpha=0.7)
    ax1.set_xlabel('Distancia horizontal x (m)', fontsize=12)
    ax1.set_ylabel('Altura y (m)', fontsize=12)
    ax1.set_title('Trayectoria y = y(x)', fontsize=13, fontweight='bold')
    ax1.text(0.98, 0.95, f'Alcance: {distance:.2f} m',
             transform=ax1.transAxes, fontsize=11,
             verticalalignment='top', horizontalalignment='right',
             bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.85))
    ax1.grid(True, alpha=0.4)
    ax1.set_xlim(left=0)
    ax1.set_ylim(bottom=0)

    # Subplot 2: Altura vs tiempo y = y(t)
    ax2.plot(t, sol[:, 2], color=color, linewidth=3, label='Altura')
    ax2.axhline(y=0, color='black', linewidth=1.5, linestyle='--', alpha=0.7)
    ax2.set_xlabel('Tiempo t (s)', fontsize=12)
    ax2.set_ylabel('Altura y (m)', fontsize=12)
    ax2.set_title('Dependencia temporal y = y(t)', fontsize=13, fontweight='bold')
    ax2.text(0.98, 0.95, f'Tiempo: {t[-1]:.2f} s',
             transform=ax2.transAxes, fontsize=11,
             verticalalignment='top', horizontalalignment='right',
             bbox=dict(boxstyle='round', facecolor='lightblue', alpha=0.85))
    ax2.grid(True, alpha=0.4)
    ax2.set_xlim(left=0)
    ax2.set_ylim(bottom=0)

    tight_layout()
    filename = f'{output_dir}/regimen_{i+1}_CD_{cd:.2f}.png'
```

```
    savefig(filename, dpi=300, bbox_inches='tight')
    print(f"Guardado: {filename}")
    show()
    close(fig)

print("\n" + "="*70)
print("INFLUENCIA DE LA FRICCIÓN")
print("="*70)

print("\n" + "-"*70)
print(f"{'Régimen':<35} {'Distancia (m)':<15} {'Pérdida (m)':<15} {'Pérdida (%)'}")
print("-"*70)

losses_m = []
losses_pct = []

for cd, dist, label in zip(drag_coeffs, dists, labels):
    if dist > 0:
        loss_m = record_distance - dist
        loss_pct = (loss_m / record_distance) * 100
        losses_m.append(loss_m)
        losses_pct.append(loss_pct)

        print(f"{label:<35} {dist:<15.2f} {loss_m:<15.2f} {loss_pct:>6.2f}%")
    else:
        losses_m.append(0)
        losses_pct.append(0)
        print(f"{label:<35} {'N/A':<15} {'N/A':<15} {'N/A'}")

print("-"*70)

# Resumen final
print("\n" + "="*70)
print(" RESUMEN DE RESULTADOS")
print("="*70)
print(f"\nVelocidad inicial (sin fricción): v0 = {v0_record:.2f} m/s")
print(f"Distancia récord objetivo: {record_distance:.2f} m\n")

if losses_m[1] > 0:
    print(f"Flujo laminar (C_D = 0.5):")
    print(f"  ->Reduce {losses_m[1]:.2f} m ({losses_pct[1]:.1f}%)")
    print(f"  -> Alcance: {distances_list[1]:.2f} m\n")

if losses_m[2] > 0:
    print(f"Flujo inestable oscilante (C_D = 0.75):")
    print(f"  -> Reduce {losses_m[2]:.2f} m ({losses_pct[2]:.1f}%)")
    print(f"  -> Alcance: {distances_list[2]:.2f} m\n")

print("CONCLUSIÓN:")
print(f"La fricción del aire reduce el alcance hasta en {max(losses_pct):.1f}%,")
print(f"lo que representa una pérdida máxima de {max(losses_m):.2f} metros.")

print("\n" + "="*70)
print(f" PROCESO COMPLETADO")
print("="*70)
print(f"\nArchivos generados en '{output_dir}/':")
print("  * regimen_1_CD_0.00.png - Sin fricción")
print("  * regimen_2_CD_0.50.png - Flujo laminar")
print("  * regimen_3_CD_0.75.png - Flujo inestable oscilante")
print("="*70 + "\n")
```

B. Código para el oscilador acoplado

Tarea 4: Resolución del problema del oscilador armónico acoplado

```
import os
import numpy as np
from pylab import *
from rich.console import Console
from rich.table import Table
from scipy.integrate import odeint

# Cosas importantes pero no tan importantes que la física
output_dir = "resultados_harm_test"  # Para guardar mis cosas
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
console = Console()  # Para que se vea bonito

# definimos lo importante: las constantes físicas del sistema
N = 2000  # numero de pasos
m = [1.0, 2.0, 0.5]  # masa de los bloques la de 2. va con [4] y la de 0.5 con [5]
k = [12.0, 10.0, 8.0, 15.0, 10.0, 20.0]  # constante de los resortes externos
k_c = [4.0, 2.0, 12.0, 5.0, 4.0, 8.0]  # constante del resorte central
tau = 30.0  # tiempo de la simulación
h = tau / float(N - 1)  # el paso del gigante (del tiempo)

ttime = linspace(0, tau, N)  # Generamos tiempo


def calcular_periodo(t, x):
    """
    Calcula el periodo promedio encontrando cruces por cero.
    """
    # Encontrar cruces por cero con pendiente positiva
    cruces = []
    for i in range(len(x) - 1):
        if x[i] <= 0 and x[i + 1] > 0:  # Cruce ascendente
            # Interpolación lineal para mayor precisión
            t_cruce = t[i] - x[i] * (t[i + 1] - t[i]) / (x[i + 1] - x[i])
            cruces.append(t_cruce)

    if len(cruces) < 2:
        return None

    # Calcular diferencias entre cruces consecutivos (medio periodo)
    # Multiplicar por 2 para obtener periodo completo
    periodos = [(cruces[i + 1] - cruces[i]) * 2 for i in range(len(cruces) - 1)]

    return mean(periodos)


def coupled_not_linear(r, t):
    """
    Sistema de EDOs para osciladores acoplados NO LINEALES.

    Fuerza no lineal: F = -k(x + 0.1x³)

    Ecuaciones:
        dx1/dt = v1
        dv1/dt = -k/m·(x1 + 0.1x1³) - k'/m·[(x1-x2) + 0.1(x1-x2)³]
        dx2/dt = v2
        dv2/dt = -k/m·(x2 + 0.1x2³) - k'/m·[(x2-x1) + 0.1(x2-x1)³]
    """
    r_1 = r[0]
    v_1 = r[1]
    r_2 = r[2]
```

```
    v_2 = r[3]
    # Fuerzas no lineales:
    F1_nolin = -k[1] / m[0] * (r_1 + 0.1 * r_1**3)
    F1_coupled = -k_c[3] / m[0] * ((r_1 - r_2) + 0.1 * (r_1 - r_2) ** 3)
    F2_nolin = -k[1] / m[0] * (r_2 + 0.1 * r_1**3)
    F2_coupled = -k_c[3] / m[0] * ((r_2 - r_1) + 0.1 * (r_2 - r_2) ** 3)

    # Ecuaciones diferenciales:
    dx_1_dt = v_1
    dv_1_dt = F1_coupled + F1_nolin
    dx_2_dt = v_2
    dv_2_dt = F2_coupled + F2_nolin
    return array([dx_1_dt, dv_1_dt, dx_2_dt, dv_2_dt])


def coupled(r, t):
    """
    Sistema para un oscilador armonico acoplado

    Variables de estado:
        r[0] = x_1 Posición de la masa (de la izquierda)
        r[1] = v_1 Velocidad de la masa (de la izquierda)
        r[2] = x_2 Posición de la masa (de la derecha)
        r[3] = v_2 Velocidad de la masa (de la derecha)

    Ecuaciones:
        dx_1/dt = v_1
        dv_1/dt = -(k+k_c)/m * x_1 + k_c/m*x_2
        dx_2/dt = v_2
        dv_2/dt = k_c/m * x_1 -(k+k_c)/m * x_2
    """
    r_1 = r[0]
    v_1 = r[1]
    r_2 = r[2]
    v_2 = r[3]
    dx_1_dt = v_1
    dv_1_dt = -(k[1] + k_c[3]) / m[0] * r_1 + k_c[3] / m[0] * r_2
    dx_2_dt = v_2
    dv_2_dt = k_c[3] / m[0] * r_1 - (k[1] + k_c[3]) / m[0] * r_2

    return array([dx_1_dt, dv_1_dt, dx_2_dt, dv_2_dt])


# condiciones iniciales:
conditions = [
    {
        "name": "Mismo deslpazamiento hacia la derecha",
        "state0": [0.8, 0, 0.8, 0],
        "description": "x_1(0) = x_2(0) = 0.8 m, v_1(0) = v_2(0) = 0",
    },
    {
        "name": "Deslpazamiento en sentidos opuestos",
        "state0": [0.8, 0, -0.8, 0],
        "description": "x_1(0) = 0.8 m x_2(0) = -0.8 m, v_1(0) = v_2(0) = 0",
    },
    {
        "name": "Una en equilibrio, otra desplazada",
        "state0": [0, 0, 1.0, 0],
        "description": "x_1(0) =0 m x_2(0) = -1 m, v_1(0) = v_2(0) = 0",
    },
]
console.print("Simulando cada cosa ...", style="blink on blue")

for idx, condition in enumerate(conditions, 1):
```

```
    console.print(f"\n  {condition['name']}")
    console.print(f"    Condición: {condition['description']}")

    # Resolver EDOs
    solution = odeint(coupled, condition["state0"], ttime)
    x1 = solution[:, 0]
    x2 = solution[:, 2]

    # Crear figura
    fig, (ax1, ax2) = subplots(2, 1, figsize=(12, 8))
    fig.suptitle(
        f"{condition['name']}\n{condition['description']}",
        fontsize=14,
        fontweight="bold",
    )

    # Subplot 1: Posiciones vs tiempo
    ax1.plot(ttime, x1, "b-", linewidth=2, label="Masa 1 (x_1)")
    ax1.plot(ttime, x2, "r-", linewidth=2, label="Masa 2 (x_2)")
    ax1.axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    ax1.set_xlabel("Tiempo (s)", fontsize=11)
    ax1.set_ylabel("Posición (m)", fontsize=11)
    ax1.set_title("Posiciones de las masas vs tiempo", fontsize=12, fontweight="bold")
    ax1.grid(True, alpha=0.3)
    ax1.legend(fontsize=10)

    # Subplot 2: Espacio de fase (x1 vs x2)
    ax2.plot(x1, x2, "g-", linewidth=2, alpha=0.7)
    ax2.plot(x1[0], x2[0], "go", markersize=10, label="Inicio")
    ax2.plot(x1[-1], x2[-1], "rs", markersize=10, label="Final")
    ax2.axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    ax2.axvline(x=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    ax2.set_xlabel("Posición masa 1, x_1 (m)", fontsize=11)
    ax2.set_ylabel("Posición masa 2, x_2 (m)", fontsize=11)
    ax2.set_title("Espacio de configuración (x_1 vs x_2)", fontsize=12, fontweight="bold")
    ax2.grid(True, alpha=0.3)
    ax2.legend(fontsize=10)
    ax2.axis("equal")

    tight_layout()
    filename = f"{output_dir}/caso_{idx}_lineal.png"
    savefig(filename, dpi=300, bbox_inches="tight")
    console.print(f"    [green] Guardado: {filename}[/green]")
    show()
    close(fig)

console.print("\n[bold yellow]Consideramos ahora ecuaciones no lineales:[/bold yellow]")
console.print("  F_1 = -k(x_1 + 0.1x_1³) - k'[(x_1-x_2) + 0.1(x_1-x_2)³]")
console.print("  F_2 = -k(x_2 + 0.1x_2³) - k'[(x_2-x_1) + 0.1(x_2-x_1)³]")

console.print("\n[bold]Comparación: Sistema lineal vs no lineal[/bold]")
console.print("Resimulando cada cosa ...", style="blink on blue")

for idx, condition in enumerate(conditions, 1):
    console.print(f"\n  {condition['name']}")

    # Resolver ambos sistemas
    sol_linear = odeint(coupled, condition["state0"], ttime)
    sol_nonlinear = odeint(coupled_not_linear, condition["state0"], ttime)

    # Crear figura comparativa
    fig, axes = subplots(2, 2, figsize=(14, 10))
    fig.suptitle(
        f"Comparación Lineal vs No Lineal\n{condition['name']}",
```

```
        fontsize=15,
        fontweight="bold",
    )

    # Subplot 1: Masa 1 - Lineal vs No lineal
    axes[0, 0].plot(ttime, sol_linear[:, 0], "b-", linewidth=2, label="Lineal")
    axes[0, 0].plot(ttime, sol_nonlinear[:, 0], "r--", linewidth=2, label="No lineal")
    axes[0, 0].axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[0, 0].set_xlabel("Tiempo (s)", fontsize=10)
    axes[0, 0].set_ylabel("Posición x_1 (m)", fontsize=10)
    axes[0, 0].set_title("Masa 1", fontsize=11, fontweight="bold")
    axes[0, 0].grid(True, alpha=0.3)
    axes[0, 0].legend(fontsize=9)

    # Subplot 2: Masa 2 - Lineal vs No lineal
    axes[0, 1].plot(ttime, sol_linear[:, 2], "b-", linewidth=2, label="Lineal")
    axes[0, 1].plot(ttime, sol_nonlinear[:, 2], "r--", linewidth=2, label="No lineal")
    axes[0, 1].axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[0, 1].set_xlabel("Tiempo (s)", fontsize=10)
    axes[0, 1].set_ylabel("Posición x_2 (m)", fontsize=10)
    axes[0, 1].set_title("Masa 2", fontsize=11, fontweight="bold")
    axes[0, 1].grid(True, alpha=0.3)
    axes[0, 1].legend(fontsize=9)

    # Subplot 3: Espacio de fase - Lineal
    axes[1, 0].plot(sol_linear[:, 0], sol_linear[:, 2], "b-", linewidth=2)
    axes[1, 0].plot(sol_linear[0, 0], sol_linear[0, 2], "go", markersize=8)
    axes[1, 0].axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[1, 0].axvline(x=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[1, 0].set_xlabel("x_1 (m)", fontsize=10)
    axes[1, 0].set_ylabel("x_2 (m)", fontsize=10)
    axes[1, 0].set_title("Espacio de fase - LINEAL", fontsize=11, fontweight="bold")
    axes[1, 0].grid(True, alpha=0.3)
    axes[1, 0].axis("equal")

    # Subplot 4: Espacio de fase - No lineal
    axes[1, 1].plot(sol_nonlinear[:, 0], sol_nonlinear[:, 2], "r-", linewidth=2)
    axes[1, 1].plot(sol_nonlinear[0, 0], sol_nonlinear[0, 2], "go", markersize=8)
    axes[1, 1].axhline(y=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[1, 1].axvline(x=0, color="black", linewidth=0.8, linestyle="--", alpha=0.5)
    axes[1, 1].set_xlabel("x_1 (m)", fontsize=10)
    axes[1, 1].set_ylabel("x_2 (m)", fontsize=10)
    axes[1, 1].set_title("Espacio de fase - NO LINEAL", fontsize=11, fontweight="bold")
    axes[1, 1].grid(True, alpha=0.3)
    axes[1, 1].axis("equal")

    tight_layout()
    filename = f"{output_dir}/caso_{idx}_comparacion.png"
    savefig(filename, dpi=300, bbox_inches="tight")
    console.print(f"    [green] Guardado: {filename}[/green]")
    show()
    close(fig)

# Por ultimo calculamos los modos normales de vibración para ambos casos:
console.print("\n" + "=" * 70, style="bold cyan")
console.print(
    " ANÁLISIS SIMPLE: MODOS NORMALES (LINEAL VS NO LINEAL)", style="bold cyan"
)
console.print("=" * 70, style="bold cyan")

# MODOS NORMALES TEÓRICOS (LINEAL)

console.print("\n" + "=" * 70, style="bold yellow")
console.print(" MODOS NORMALES TEÓRICOS (SISTEMA LINEAL)", style="bold yellow")
```

```
console.print("=" * 70, style="bold yellow")

# Matriz del sistema
A = array(
    [[-(k[1] + k_c[3]) / m[0], k_c[3] / m[0]], [k_c[3] / m[0], -(k[1] + k_c[3]) / m[0]]]
)

eigenvalues, eigenvectors = linalg.eig(A)
omega_teorico = sqrt(-eigenvalues)
freq_teorico = omega_teorico / (2 * pi)
periodo_teorico = 1 / freq_teorico

console.print(f"\n[bold]Modo 1 (Simétrico):[/bold]")
console.print(f"  w_1 = {omega_teorico[0]:.4f} rad/s")
console.print(f"  f_1 = {freq_teorico[0]:.4f} Hz")
console.print(f"  T_1 = {periodo_teorico[0]:.4f} s")

console.print(f"\n[bold]Modo 2 (Antisimétrico):[/bold]")
console.print(f"  w_2 = {omega_teorico[1]:.4f} rad/s")
console.print(f"  f_2 = {freq_teorico[1]:.4f} Hz")
console.print(f"  T_2 = {periodo_teorico[1]:.4f} s")

# ANÁLISIS PARA DIFERENTES AMPLITUDES
console.print("\n" + "=" * 70, style="bold green")
console.print(" ANÁLISIS NUMÉRICO: DEPENDENCIA CON AMPLITUD", style="bold green")
console.print("=" * 70, style="bold green")

amplitudes = [0.2, 0.5, 0.8, 1.2]

# Almacenar resultados
resultados = {
    "amplitud": [],
    "T_lineal": [],
    "T_nonlinear": [],
    "f_lineal": [],
    "f_nonlinear": [],
    "diff_pct": [],
}

for amp in amplitudes:
    console.print(f"\n[bold cyan]Amplitud: {amp} m[/bold cyan]")

    # Condición inicial: modo simétrico
    state0 = [amp, 0.0, amp, 0.0]

    # Resolver
    sol_lin = odeint(coupled, state0, ttime)
    sol_nonlin = odeint(coupled_not_linear, state0, ttime)

    # Calcular periodos
    T_lin = calcular_periodo(ttime, sol_lin[:, 0])
    T_nonlin = calcular_periodo(ttime, sol_nonlin[:, 0])

    if T_lin and T_nonlin:
        f_lin = 1 / T_lin
        f_nonlin = 1 / T_nonlin
        diff = abs(f_nonlin - f_lin) / f_lin * 100

        resultados["amplitud"].append(amp)
        resultados["T_lineal"].append(T_lin)
        resultados["T_nonlinear"].append(T_nonlin)
        resultados["f_lineal"].append(f_lin)
        resultados["f_nonlinear"].append(f_nonlin)
        resultados["diff_pct"].append(diff)
```

```
        console.print(f"  Lineal:    T = {T_lin:.4f} s,  f = {f_lin:.4f} Hz")
        console.print(f"  No lineal: T = {T_nonlin:.4f} s,  f = {f_nonlin:.4f} Hz")
        console.print(f"  Diferencia: {diff:.2f}%")

# TABLA COMPARATIVA
console.print("\n" + "=" * 70, style="bold magenta")
console.print(" TABLA COMPARATIVA", style="bold magenta")
console.print("=" * 70, style="bold magenta")

table = Table(title="Frecuencias vs Amplitud", style="red")
table.add_column("Amplitud (m)", justify="center")
table.add_column("f Lineal (Hz)", justify="center")
table.add_column("f No Lineal (Hz)", justify="center")
table.add_column("Diferencia (%)", justify="center")

for i in range(len(resultados["amplitud"])):
    table.add_row(
        f"{resultados['amplitud'][i]:.1f}",
        f"{resultados['f_lineal'][i]:.4f}",
        f"{resultados['f_nonlinear'][i]:.4f}",
        f"{resultados['diff_pct'][i]:.2f}",
    )

console.print(table)

console.print("\n" + "=" * 70, style="bold green")
console.print(" RESUMEN DE RESULTADOS", style="bold green")
console.print("=" * 70, style="bold green")

console.print(f"\n[bold]Diferencias entre sistema lineal y no lineal:[/bold]")
console.print("  • Sistema lineal: Oscilaciones armónicas puras (sinusoidales)")
console.print("  • Sistema no lineal: Distorsión de la forma de onda")
console.print("  • No linealidad introduce dependencia de amplitud en frecuencia")
console.print("  • Espacio de fase no lineal muestra trayectorias distorsionadas")

console.print(f"\n[bold cyan] Archivos generados en '{output_dir}/':[/bold cyan]")
console.print("  • caso_1_lineal.png - Ambas masas a la derecha (lineal)")
console.print("  • caso_2_lineal.png - Masas en sentidos opuestos (lineal)")
console.print("  • caso_3_lineal.png - Una en equilibrio (lineal)")
console.print("  • caso_1_comparacion.png - Comparación lineal vs no lineal")
console.print("  • caso_2_comparacion.png - Comparación lineal vs no lineal")
console.print("  • caso_3_comparacion.png - Comparación lineal vs no lineal")

console.print("\n" + "=" * 70, style="bold green")
console.print(" PROCESO COMPLETADO", style="bold green")
console.print("=" * 70 + "\n", style="bold green")
```

C. Código para la vibración de una cuerda

Tarea 4: Resolución del problema de la cuerda vibrante

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
from typing import Callable, Tuple
import os

# Si bien no hemos visto nada de esto, quiero practicar para ya ponerme a trabajar
# por lo que se me hizo mucho mas facil construir una clase para ya no repetir codigo
class CuerdaVibrante:
    """
```

```
    Simulación de la vibración de una cuerda usando el método de diferencias finitas.

    Resuelve la ecuación de onda:
        d^2y/dx^2 = (1/c^2) d^2y/dt^2

    con condiciones de frontera:
        y(0, t) = y(L, t) = 0

    y condiciones iniciales:
        y(x, 0) = f(x)
        dy/dt(x, 0) = g(x)

    Parametros
    ----------
    L : float
        Longitud de la cuerda (m)
    c : float
        Velocidad de onda (m/s), c = sqrt (T/rho)
    T_max : float
        Tiempo total de simulación (s)
    Nx : int
        Número de puntos espaciales
    Courant : float
        Número de Courant r = c·Delta t/Delta x (debe ser leq 1 para estabilidad)
    """
    #constructor:
    def __init__(self, L: float, c: float, T_max: float, Nx: int, Courant: float):
        self.L = L
        self.c = c
        self.T_max = T_max
        self.Nx = Nx
        self.Courant = Courant

        # Discretización espacial
        self.dx = L / (Nx - 1)
        self.x = np.linspace(0, L, Nx)

        # Discretización temporal (según condición de Courant)
        self.dt = Courant * self.dx / c
        self.Nt = int(T_max / self.dt)
        self.t = np.linspace(0, self.dt * self.Nt, self.Nt)

        # Parámetro de estabilidad
        self.r_squared = (c * self.dt / self.dx) ** 2

        # Matrices de solución
        self.y = np.zeros((Nx, self.Nt))

        # Información
        print(f"{'='*60}")
        print(f"CONFIGURACIÓN DE LA SIMULACIÓN")
        print(f"{'='*60}")
        print(f"Longitud de la cuerda: L = {L} m")
        print(f"Velocidad de onda: c = {c} m/s")
        print(f"Tiempo total: T_max = {T_max} s")
        print(f"Puntos espaciales: Nx = {Nx}")
        print(f"Paso espacial: Delta x = {self.dx:.6f} m")
        print(f"Paso temporal: Delta t = {self.dt:.6f} s")
        print(f"Número de Courant: r = c·Delta t/Delta x = {self.Courant:.4f}")
        print(f"Parámetro r^2 = {self.r_squared:.6f}")
        print(f"Pasos temporales: Nt = {self.Nt}")
        print(f"{'='*60}")

        # Verificar condición de estabilidad
```

```
        if self.Courant > 1.0:
            print(f" !!!!! ADVERTENCIA: Número de Courant > 1.0")
            print(f"   La solución puede ser INESTABLE")
        else:
            print(f"Condición de Courant satisfecha (r leq 1)")
        print(f"{'='*60}\n")
    #funciones a usar más adelante, como desplazamiento, velocidad, etc.
    def condicion_inicial_desplazamiento(self, f: Callable[[np.ndarray], np.ndarray]):
        """
        Establece la condición inicial de desplazamiento y(x, 0) = f(x).

        Parameters
        ----------
        f : callable
            Función que define el desplazamiento inicial
        """
        self.y[:, 0] = f(self.x)
        # Condiciones de frontera
        self.y[0, 0] = 0
        self.y[-1, 0] = 0

    def condicion_inicial_velocidad(self, g: Callable[[np.ndarray], np.ndarray]):
        """
        Establece la condición inicial de velocidad dy/dt(x, 0) = g(x).

        Calcula y(x, Delta t) usando:
            y_{i,1} = y_{i,0} + Delta t·g(x_i) + (1/2)r^2[y_{i+1,0} - 2y_{i,0} + y_{i-1,0}]

        Parameters
        ----------
        g : callable
            Función que define la velocidad inicial
        """
        g_vals = g(self.x)

        for i in range(1, self.Nx - 1):
            self.y[i, 1] = (self.y[i, 0] +
                            self.dt * g_vals[i] +
                            0.5 * self.r_squared * (self.y[i+1, 0] - 2*self.y[i, 0] + self.y[i-1,
                             ↪  0]))

        # Condiciones de frontera
        self.y[0, 1] = 0
        self.y[-1, 1] = 0

    def resolver(self):
        """
        Resuelve la ecuación de onda usando el esquema de diferencias finitas explícito.

        Algoritmo:
            y_{i,j+1} = 2y_{i,j} - y_{i,j-1} + r^2[y_{i+1,j} - 2y_{i,j} + y_{i-1,j}]
        """
        print("Resolviendo ecuación de onda...")

        for j in range(1, self.Nt - 1):
            for i in range(1, self.Nx - 1):
                self.y[i, j+1] = (2 * self.y[i, j] -
                                  self.y[i, j-1] +
                                  self.r_squared * (self.y[i+1, j] - 2*self.y[i, j] + self.y[i-1,
                                   ↪  j]))

            # Condiciones de frontera
            self.y[0, j+1] = 0
            self.y[-1, j+1] = 0
```

```
            # Progreso
            if j % 100 == 0:
                print(f"  Paso temporal {j}/{self.Nt} ({100*j/self.Nt:.1f}%)")

        print("Simulación completada\n")

    def analizar_estabilidad(self):
        """
        Analiza la estabilidad de la solución verificando:
        1. Valores máximos a lo largo del tiempo
        2. Si hay crecimiento exponencial (inestabilidad)
        """
        print("Análisis de estabilidad:")

        max_vals = np.max(np.abs(self.y), axis=0)
        max_inicial = max_vals[0]
        max_final = max_vals[-1]
        max_global = np.max(max_vals)

        print(f"  Amplitud inicial: {max_inicial:.6f}")
        print(f"  Amplitud final: {max_final:.6f}")
        print(f"  Amplitud máxima: {max_global:.6f}")

        if max_final > 10 * max_inicial:
            print(f"  !!!! SOLUCIÓN INESTABLE (crecimiento > 10x)")
            return False
        elif max_final > 2 * max_inicial:
            print(f"  !  Solución marginalmente estable")
            return True
        else:
            print(f"  :D Solución estable")
            return True
    #Wuakala, ya no me haga animas más
    def animar(self, archivo_salida: str = None, intervalo: int = 50, saltar_frames: int = 1):
        """
        Crea una animación de la vibración de la cuerda.

        Parameters
        ----------
        archivo_salida : str, optional
            Nombre del archivo para guardar la animación (formato .gif)
        intervalo : int
            Intervalo entre frames en ms
        saltar_frames : int
            Número de pasos temporales a saltar entre frames
        """
        print(f"Creando animación...")

        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

        # Configurar subplot 1: Animación de la cuerda
        ax1.set_xlim(0, self.L)
        y_max = np.max(np.abs(self.y))
        ax1.set_ylim(-1.5*y_max, 1.5*y_max)
        ax1.set_xlabel('Posición x (m)', fontsize=12)
        ax1.set_ylabel('Desplazamiento y (m)', fontsize=12)
        ax1.grid(True, alpha=0.3)

        line, = ax1.plot([], [], 'b-', lw=2, label='Cuerda')
        time_text = ax1.text(0.02, 0.95, '', transform=ax1.transAxes,
                             fontsize=12, verticalalignment='top',
                             bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))
        ax1.legend(loc='upper right')
```

```
        # Configurar subplot 2: Evolución temporal en puntos específicos
        ax2.set_xlim(0, self.T_max)
        ax2.set_ylim(-1.5*y_max, 1.5*y_max)
        ax2.set_xlabel('Tiempo t (s)', fontsize=12)
        ax2.set_ylabel('Desplazamiento y (m)', fontsize=12)
        ax2.grid(True, alpha=0.3)

        # Puntos de observación
        puntos_obs = [self.Nx//4, self.Nx//2, 3*self.Nx//4]
        colores = ['r', 'g', 'b']
        lineas_temp = []

        for i, (punto, color) in enumerate(zip(puntos_obs, colores)):
            line_temp, = ax2.plot([], [], color=color, lw=1.5,
                                  label=f'x = {self.x[punto]:.2f} m')
            lineas_temp.append(line_temp)

        ax2.legend(loc='upper right')

        # Título principal
        titulo = f'Vibración de Cuerda: L={self.L}m, c={self.c}m/s, Courant={self.Courant:.3f}'
        if self.Courant > 1.0:
            titulo += ' [INESTABLE]'
        fig.suptitle(titulo, fontsize=14, fontweight='bold')

        plt.tight_layout()

        # Función de inicialización
        def init():
            line.set_data([], [])
            for line_temp in lineas_temp:
                line_temp.set_data([], [])
            time_text.set_text('')
            return [line] + lineas_temp + [time_text]

        # Función de animación
        def animate(frame):
            j = frame * saltar_frames
            if j >= self.Nt:
                j = self.Nt - 1

            # Actualizar cuerda
            line.set_data(self.x, self.y[:, j])
            time_text.set_text(f't = {self.t[j]:.4f} s\nFrame {frame}/{len(frames)}')

            # Actualizar evolución temporal
            for i, (punto, line_temp) in enumerate(zip(puntos_obs, lineas_temp)):
                line_temp.set_data(self.t[:j+1], self.y[punto, :j+1])

            return [line] + lineas_temp + [time_text]

        # Frames a animar
        frames = range(0, self.Nt, saltar_frames)

        anim = animation.FuncAnimation(fig, animate, init_func=init,
                                       frames=len(frames), interval=intervalo,
                                       blit=True, repeat=True)

        # Guardar animación
        if archivo_salida:
            print(f"  Guardando animación en {archivo_salida}...")
            writer = PillowWriter(fps=20)
            anim.save(archivo_salida, writer=writer)
```

```
            print(f"  :D Animación guardada")

        plt.show()
        print(":D Animación completada\n")

        return anim

    def graficar_evolucion_temporal(self, puntos_x: list = None):
        """
        Grafica la evolución temporal del desplazamiento en puntos específicos.

        Parameters
        ----------
        puntos_x : list, optional
            Lista de posiciones x donde observar (por defecto: cuartos de la cuerda)
        """
        if puntos_x is None:
            puntos_x = [self.L/4, self.L/2, 3*self.L/4]

        fig, axes = plt.subplots(len(puntos_x), 1, figsize=(12, 3*len(puntos_x)))

        if len(puntos_x) == 1:
            axes = [axes]

        for ax, x_pos in zip(axes, puntos_x):
            idx = np.argmin(np.abs(self.x - x_pos))
            ax.plot(self.t, self.y[idx, :], 'b-', lw=1.5)
            ax.set_xlabel('Tiempo t (s)')
            ax.set_ylabel('Desplazamiento y (m)')
            ax.set_title(f'Evolución temporal en x = {self.x[idx]:.3f} m')
            ax.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.show()

    def graficar_instantaneas(self, tiempos: list):
        """
        Grafica instantáneas de la cuerda en diferentes tiempos.

        Parameters
        ----------
        tiempos : list
            Lista de tiempos donde graficar
        """
        fig, ax = plt.subplots(figsize=(12, 6))

        for t_val in tiempos:
            idx = np.argmin(np.abs(self.t - t_val))
            ax.plot(self.x, self.y[:, idx], label=f't = {self.t[idx]:.4f} s')

        ax.set_xlabel('Posición x (m)')
        ax.set_ylabel('Desplazamiento y (m)')
        ax.set_title('Instantáneas de la cuerda en diferentes tiempos')
        ax.legend()
        ax.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()



# funcion para condiciones iniciales:


def desplazamiento_gaussiano(x, A=0.1, x0=None, sigma=0.05):
```

```
    """
    Pulso gaussiano:
        f(x) = A * exp(-(x - x0)^2 / (2sigma^2))

    Parameters
    ----------
    x : array
        Posiciones
    A : float
        Amplitud
    x0 : float
        Centro del pulso (por defecto: centro de la cuerda)
    sigma : float
        Ancho del pulso
    """
    if x0 is None:
        x0 = (x[0] + x[-1]) / 2
    return A * np.exp(-(x - x0)**2 / (2 * sigma**2))


def desplazamiento_triangular(x, A=0.1, x0=None):
    """
    Desplazamiento triangular.

    Parameters
    ----------
    x : array
        Posiciones
    A : float
        Amplitud
    x0 : float
        Posición del pico
    """
    if x0 is None:
        x0 = (x[0] + x[-1]) / 2
    L = x[-1] - x[0]
    y = np.zeros_like(x)
    mask1 = x <= x0
    mask2 = x > x0
    y[mask1] = A * x[mask1] / x0
    y[mask2] = A * (L - x[mask2]) / (L - x0)
    return y


def velocidad_cero(x):
    """Velocidad inicial cero."""
    return np.zeros_like(x)



# para esto creamos una clase:
#  experimentos


def experimento_1_estable():
    """
    Experimento 1: Condición de Courant SATISFECHA (estable)
    """
    print("\n" + "="*60)
    print("EXPERIMENTO 1: CONDICIÓN DE COURANT SATISFECHA")
    print("="*60 + "\n")

    # Parámetros
    L = 1.0          # Longitud de la cuerda (m)
```

```
    c = 10.0          # Velocidad de onda (m/s)
    T_max = 0.5       # Tiempo total (s)
    Nx = 100          # Puntos espaciales
    Courant = 0.5     # r = 0.5 < 1 → ESTABLE

    # Crear simulación
    cuerda = CuerdaVibrante(L, c, T_max, Nx, Courant)

    # Condiciones iniciales: pulso gaussiano
    cuerda.condicion_inicial_desplazamiento(
        lambda x: desplazamiento_gaussiano(x, A=0.1, sigma=0.05)
    )
    cuerda.condicion_inicial_velocidad(velocidad_cero)

    # Resolver
    cuerda.resolver()

    # Análisis
    cuerda.analizar_estabilidad()

    # Visualización
    cuerda.animar(archivo_salida='animacion_estable.gif', saltar_frames=2)
    cuerda.graficar_instantaneas([0, T_max/4, T_max/2, 3*T_max/4, T_max])

    return cuerda


def experimento_2_inestable():
    """
    Experimento 2: Condición de Courant VIOLADA (inestable)
    """
    print("\n" + "="*60)
    print("EXPERIMENTO 2: CONDICIÓN DE COURANT VIOLADA")
    print("="*60 + "\n")

    # Parámetros
    L = 1.0           # Longitud de la cuerda (m)
    c = 10.0          # Velocidad de onda (m/s)
    T_max = 0.5       # Tiempo total (s)
    Nx = 100          # Puntos espaciales
    Courant = 1.5     # r = 1.5 > 1 → INESTABLE

    # Crear simulación
    cuerda = CuerdaVibrante(L, c, T_max, Nx, Courant)

    # Condiciones iniciales: pulso gaussiano
    cuerda.condicion_inicial_desplazamiento(
        lambda x: desplazamiento_gaussiano(x, A=0.1, sigma=0.05)
    )
    cuerda.condicion_inicial_velocidad(velocidad_cero)

    # Resolver
    cuerda.resolver()

    # Análisis
    cuerda.analizar_estabilidad()

    # Visualización
    cuerda.animar(archivo_salida='animacion_inestable.gif', saltar_frames=2)
    cuerda.graficar_instantaneas([0, T_max/4, T_max/2])

    return cuerda
```

```python
def experimento_3_critico():
    """
    Experimento 3: Condición de Courant EN EL LÍMITE (marginalmente estable)
    """
    print("\n" + "="*60)
    print("EXPERIMENTO 3: CONDICIÓN DE COURANT EN EL LÍMITE")
    print("="*60 + "\n")

    # Parámetros
    L = 1.0          # Longitud de la cuerda (m)
    c = 10.0         # Velocidad de onda (m/s)
    T_max = 0.5      # Tiempo total (s)
    Nx = 100         # Puntos espaciales
    Courant = 1.0    # r = 1.0 (crítico)

    # Crear simulación
    cuerda = CuerdaVibrante(L, c, T_max, Nx, Courant)

    # Condiciones iniciales: pulso gaussiano
    cuerda.condicion_inicial_desplazamiento(
        lambda x: desplazamiento_gaussiano(x, A=0.1, sigma=0.05)
    )
    cuerda.condicion_inicial_velocidad(velocidad_cero)

    # Resolver
    cuerda.resolver()

    # Análisis
    cuerda.analizar_estabilidad()

    # Visualización
    cuerda.animar(archivo_salida='animacion_critica.gif', saltar_frames=2)

    return cuerda


def comparacion_courant():
    """
    Comparación de diferentes valores de Courant.
    """
    print("\n" + "="*60)
    print("COMPARACIÓN: DIFERENTES VALORES DE COURANT")
    print("="*60 + "\n")

    L = 1.0
    c = 10.0
    T_max = 0.2
    Nx = 100

    courant_vals = [0.3, 0.7, 1.0, 1.2, 1.5]

    fig, axes = plt.subplots(len(courant_vals), 1, figsize=(12, 3*len(courant_vals)))

    for ax, r in zip(axes, courant_vals):
        cuerda = CuerdaVibrante(L, c, T_max, Nx, r)
        cuerda.condicion_inicial_desplazamiento(
            lambda x: desplazamiento_gaussiano(x, A=0.1, sigma=0.05)
        )
        cuerda.condicion_inicial_velocidad(velocidad_cero)
        cuerda.resolver()

        # Graficar solución final
        ax.plot(cuerda.x, cuerda.y[:, -1], 'b-', lw=2)
        ax.set_ylabel('y (m)')
```

```
        ax.set_title(f'Courant = {r:.2f} ({"ESTABLE" if r <= 1 else "INESTABLE"})')
        ax.grid(True, alpha=0.3)

        max_val = np.max(np.abs(cuerda.y[:, -1]))
        ax.set_ylim(-1.5*max_val, 1.5*max_val) if max_val > 0 else ax.set_ylim(-0.1, 0.1)

    axes[-1].set_xlabel('Posición x (m)')
    plt.tight_layout()
    plt.savefig('comparacion_courant.png', dpi=300, bbox_inches='tight')
    plt.show()

    print(":D Gráfica de comparación guardada en 'comparacion_courant.png'\n")



# llamamos al MAIN


if __name__ == "__main__":
    # Crear carpeta para resultados
    os.makedirs('resultados_cuerda', exist_ok=True)
    os.chdir('resultados_cuerda')

    print("\n" + "-"*30)
    print("SIMULACIÓN DE VIBRACIÓN DE CUERDA")
    print("Ecuación de Onda - Análisis de Estabilidad")
    print("-"*30 + "\n")

    # Ejecutar experimentos
    print("\nEjecutando experimentos...\n")

    # Experimento 1: Estable
    cuerda1 = experimento_1_estable()

    # Experimento 2: Inestable
    cuerda2 = experimento_2_inestable()

    # Experimento 3: Crítico
    cuerda3 = experimento_3_critico()

    # Comparación
    comparacion_courant()

    print("\n" + "="*60)
    print("RESUMEN DE RESULTADOS")
    print("="*60)
    print("""
CONCLUSIONES:

1. Condición de Courant r leq 1:
   - r = c·Delta t/Delta x leq 1
   - Equivalente a: Delta t leq Delta x/c

2. Interpretación física:
   - La onda no puede viajar más de Delta x en un tiempo Delta t
   - Respeta la causalidad del sistema

3. Resultados experimentales:
   - r < 1: Solución ESTABLE y físicamente correcta
   - r = 1: Marginalmente estable (caso límite)
   - r > 1: Solución INESTABLE (oscilaciones espurias)

4. Recomendación práctica:
   - Usar r approx 0.5 - 0.8 para máxima estabilidad
```

```
- Nunca usar r > 1.0
"""")
print("="*60 + "\n")

print(":D Todos los experimentos completados")
print(f"dir: Resultados guardados en: {os.getcwd()}")
```