# Lab 1 IA – Control de Péndulo Invertido con PID y Control Difuso

Tomás Hernández Martínez
*Ingeniería Mecatrónica*
*Universidad Militar Nueva Granada*
Cajica, Colombia
est.tomas.hernandezm@unimilitar.edu.co

***Abstract*— This report presents a detailed analysis on the design, implementation, and comparison of automatic controllers for stabilizing a cart-mounted inverted pendulum system. A first-principles dynamic simulation was carried out using Python, with the Pygame library for visualization and validation. Three control strategies were tested: a manual system to establish a baseline, a Proportional-Integral-Derivative (PID) controller representing classical control, and a Fuzzy Logic controller based on a Mamdani inference system. The results indicate that although the optimized PID controller provides fast and accurate stabilization, its performance is highly dependent on gain tuning and can be mechanically aggressive. On the other hand, the Fuzzy Logic controller showed noticeably smoother performance and greater robustness to external disturbances, achieving elegant stabilization through a set of rules.**

**Keywords—Inverted Pendulum, Control Systems, PID Controller, Fuzzy Logic, Mamdani Inference System, Dynamic Simulation, Python, Robotics.**

## I. INTRODUCCION

Dentro de la ingeniería de control hay bastantes desafíos interesantes para imponer una forma de controlar un caos que parece incontrolable con métodos sin precisión, siendo un reto interesante. Un sistema que obedece al caos propio en el control es el péndulo invertido, siendo un reto para controlar con cualquier tipo de controlador, notando que las perturbaciones añaden un caos en su control, por lo que puede resultar en un entorno vivo para probar algoritmos propios y el desempeño de diferentes controladores que puede abarcar el control tradicional y los fundamentos de IA.

En este informe se documenta el desafío de dominar al sistema desde el propio manejo manual y sus controladores, no solo partiendo del desarrollo teórico, sino ensamblando una propia simulación donde las propias físicas del sistema están en el total control del programador. En el primer paso, es fundamental evidenciar que el control manual del sistema puede llegar a ser desafiante y elemental para comprender el comportamiento del modelo, en el cual se busca saber la dificultad en la precisión que require cada toque de fuerza que se le ejerce en la simulación, para despues poder conocer que el ámbito del control puede llegar a ser bastante necesario y que puede haber más o menos efectividad dependiendo de los controladores y sus parámetros de control.

Este documento abarca la exploración propia de las técnicas de control y sus eficacias con base a este simple pero dificil sistema de controlar, que puede abarcar desde un control con matemática y control tradicional, o el seguimiento de reglas que puede parecerse más a la propia intuición y experiencia humana. Al final se piensa en una clara comparación entre métodos: ¿Que estrategia prima sobre la otra? ¿Es más efectiva la teoría de control, o un sistema flexible de reglas e intuición para esta aplicación? Este informe buscará responder a las preguntas y como tal evidenciar el desafío propio, comparando las eficiencias de los controladores.

## II. MATERIALES Y METODOLOGÍA

Para este informe se utilizará solamente software de programación y computadores con Buena capacidad computacional para las simulaciones, basando los códigos en python y usando como opción extra un aplicativo como Webots que regula las físicas de una manera más confinable que un programa general.

La metodología expuesta por el laboratorio es la siguiente:

a. Animación con control manual en python. Realizar una visualización que permita ejercer el control manual, con magnitudes de las variables en valores cercanos a la realidad y variables de entrada configurables como angulo, velocidad angular, velocidad del vehiculo y demas. Implementar el control de fuerza al carro que sostiene el pendulo y poder controlar la direccion del vehiculo , por ejemplo con las teclas “A” y “D” para que el vehiculo se mueva en la direccion deseada y el usuario intente mantener el pendulo de forma vertical. Usando el modelo generado en el punto anterior como base fisica.

b. Animación con control manual en webots usando python. Realizar una visualización que permita ejercer el control manual, con magnitudes de las variables en valores cercanos a la realidad y variables de entrada configurables como angulo, velocidad angular, velocidad del vehiculo y demas. Implementar el control de fuerza al carro que sostiene el pendulo y poder controlar la direccion del

vehiculo , por ejemplo con las teclas “A” y “D” para que el vehiculo se mueva en la direccion deseada y el usuario intente mantener el pendulo de forma vertical. Usando el modelo fisico interno de webots como base fisica.

c. Implementar conjuntos difusos para el procesamiento de las condiciones del pendulo y tomar acciones sobre el carro de control. Modelar conjuntos difusos de entrada , un motor de inferencia de Mamdani , conjuntos difusos de salida y defusificacion , usando python. Simular y aplicar el controlador difuso sobre la planta virtual, de forma que el robot pendulo se mantenga en el angulo deseado a partir del movimiento del vehiculo que lo sostiene. Optimizar los conjuntos difusos para minimizar el error.

Y así mismo el desarrollo de un controlador PID en Webots para mostrar de forma más evidente el funcionamiento de un controlador en un entorno más estandarizado y real.

## III. DESARROLLO Y ANÁLISIS

Para poder desarrollar bien la experimentación toca definer varios conceptos que son relevantes para la implemetación de los controladores y del propio conocimiento total del sistema, para después emplear los controladores en el sistema seleccionado.

### *a. Control PID*

El muy conocido controlador PID (Proporcional, Integral y Derivativo) es uno de los métodos de control más conocidos y utilizados en la industria, así mismo utilizado para fines académicos por su eficiencia, eficacia y sencillez de aplicar en muchos sistemas que necesitan tener un control integrado. Su sistema se basa en calcular una señal de control a partir de tres parámetros clave para este tipo de controlador:

1. **Proporcional:** Responde al error del sistema de forma inmediata mediante una ganancia, dando Buena rapidez en la respuesta del controlador, aunque si resulta mal parametrizado puede generar oscilaciones o respuestas muy lentas del sistema ante diferentes situaciones.
2. **Integral:** Acumula el error del sistema a lo largo del tiempo y teniendo como objetivo corregirlo, ayudando al error estacionario que no puede ajustar por si solo el proporcional.
3. **Derivativo:** Anticipa el comportamiento future del error, ayudando a reducer oscilaciones dando como final una mejora en la estabilidad propia del sistema.

Al aplicar los tres parámetros de control permite que el PID se desempeñe de forma ideal ante cualquier perturbación en el sistema a controlar. Para poder ajustar sus funciones de forma óptima se editan sus ganancias hasta una que pueda dar un desdempeño óptimo dependiendo de la necesidad que se tenga.

### *b. Control Difuso*

Por parte del control difuso no resulta ser igual que el PID con unas ganancias específicas dadas por una ecuación, sino que presenta una cercanía a la propia intuición humana, siendo empleado mediante reglas y condiciones que describen el comportamiento deseado dependiendo de la necesidad del usuario, hacienda que dependa netamente de la intuición la efectividad del control, pero añadiendo versatilidad a la toma de decisiones para su funcionamiento.

Los conceptos clave para aprender sobre el control difuso son los siguientes:

1. Reglas del control difuso: Contiene las reglas necesarias para poder operar el control difuso relacionando entradas y salidas.
2. Fuzzificación: Contiene los valores de entrada del sistema y los convierte en valores difusos mediante funciones de pertenencia.
3. Defuzzificación: Vuelve el valor difuso en una propia acción de control.
4. Inferencia: Combina las reglas y determina cual debe de ser la salida propia que debe emplear el sistema de control.

Los sistemas de control difuso reduce la complejidad que pueden tener procedimientos basados en sistemas de control tradicionales como el PID más que todo en sistemas no lineales, los cuales resultan ser dificiles de linealizar una sola sección o desarrollar un control para todas las zonas no lineales, por lo que solo es necesaria la intuición y una Buena práctica en el desarrollo de las reglas.

### *c. Simulaciones, Modelo y Experimentación*

Para empezar, se necesita del modelo del péndulo invertido para poder conocer como se comporta en sus diferentes variables y así poder determinar como ejecutar sus controladores.

Las variables utilizadas son las siguientes:

M: Masa del carro (Kg)

m: Masa de la lenteja del péndulo (Kg)

L: Longitud desde el pivote hasta el centro de masa del péndulo (m)

g: Aceleración de la gravedad (m/s^2)

I: Momento de inercia del péndulo respecto a su centro de masa

b: Coeficiente de fricción viscosa en la articulación del péndulo

x: Posición horizontal del carro (m)

θ: Ángulo del péndulo con respecto a la vertical (θ=0 es vertical hacia arriba)

F: Fuerza horizontal aplicada al carro (N)

Xp: Velocidad del carro (m/s)

Θp: Velocidad angular del carro (rad)

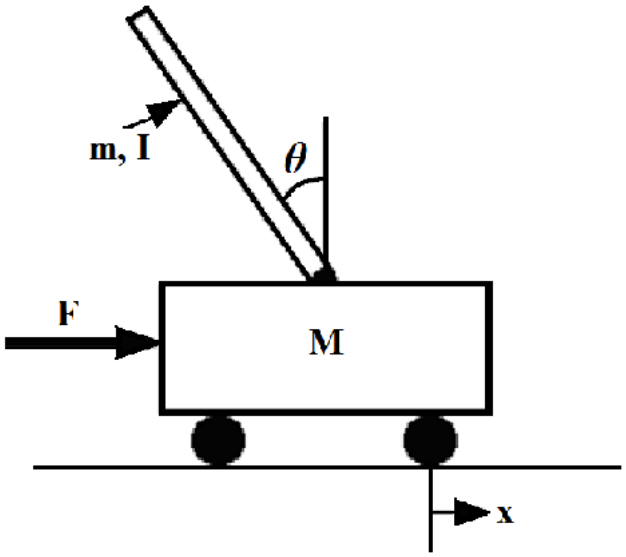


*Fig. 1. Péndulo invertido con variables simplificadas. Tomado de: https://www.researchgate.net/figure/Figura-54-Pendulo-invertido-sobre-un-carro_fig2_260248290*

El modelado resultó en las siguientes ecuaciones:

$$\ddot{\theta} = \frac{(M+m)mgLsin\theta - mLcos\theta\left(F + mL\dot{\theta}^2 sin\theta\right) - (M+m)b\dot{\theta}}{(M+m)(I+mL^2) - (mLcos\theta)^2}$$

$$\ddot{x} = \frac{(F + mL\dot{\theta}^2 sin\theta)(I+mL^2) - (mLcos\theta)(mgLsin\theta - b\dot{\theta})}{(M+m)(I+mL^2) - (mLcos\theta)^2}$$

Las ecuaciones anteriores dejan ver la dinámica del sistema que va a acojer en todas las simulaciones.

### 1. *Control manual*

Este programa solo cuenta con el control por las teclas A y D para poder controlar el carro de forma manual con una fuerza específica determinada por el usuario en el código. Al principio el programa debe de capturar la intención de control llamando a la función "modelo_pendulo_carro" entregándole dos parámetros clave, siendo: el estado actual del sistema y la fuerza aplicada. El modelado obedece a las constantes del sistema y los parámetros dados por la función, devolviendo un vector de derivadas y no netamente la nueva posición. Conociendo los cambios en el sistema se predice como será después, implementando un sistema de integración numérica conocida como el Método de Euler Explícito, para poder simular el paso del tiempo para que se puedan efectuar los cambios.

Y aplicando lógica dentro del manejo del módulo de pygame se pueden realizar las físicas y movimientos necesarios para la simulación, representando los estados en gráficos.

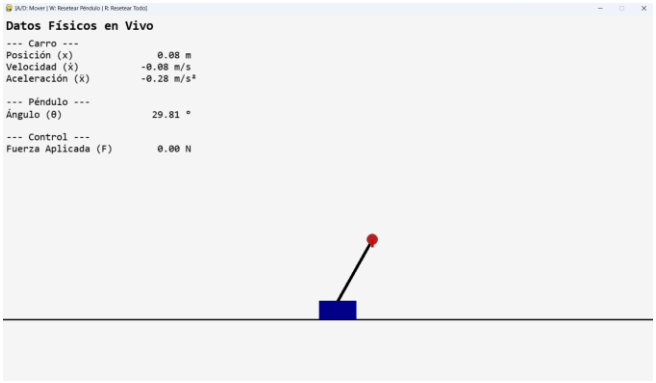


*Fig. 2. Control manual en Pygame. Elaboración propia.*

La figura 2 muestra los parámetros que se encuentran en la actualidad del sistema del carro, así como también el ángulo del péndulo en grados. Dejando en el sistema del juego un control por fuerza aplicada por el teclado y la tecla "w" como parámetro de regreso al punto inicial del péndulo.

Ya dentro de la manipulación del programa el mayor desafío en el control resultó dado por la latencia de reacción humana, siendo determinante para poder controlarlo de una buena manera. Al ser un sistema tan rápido este resultó complejo de controlar aplicando un control irregular y para nada cercano a la intuición humana, los ajustes finos y proporcionales resultarían efectivos en ese control para evitar oscilaciones evitables que con la propia mano humana ocurren. Así mismo la única manera de poder hacer un sistema más fácil de controlar fue aplicando constantes diferentes en el carro y aplicando una fuerza coherente con el sistema para evitar el descontrol.

### 2. *Control PID*

Ya en el control PID se plica la teoría de control tradicional y las ganancias correspondientes al sistema para poder mejorar su comportamiento, y teniendo como único objetivo final reducer el error a cero teniendo como referencia el ángulo cero en el péndulo mediante "SETPOINT_ANGULO". El PID calcula la fuerza que debe de aplicar al carro basandose en el error en perspectiva presente, pasada y futura.

El proporcionaltoma en cuenta el estado para poder actuar y sacar el error actual teniendo a theta y el setpoint como su operando. Ya el control proporcional lo único que hace es multiplicar su ganancia Kp con el error aplicando las fuerzas respectivas al carro que crea pertinentes en su sistema, teniendo ganancia alta para respuesta rápida y bajas para una más suave y lenta.

El derivative obedece a la misma lógica de multiplicación por su ganancia, aunque la variable “derivative” viene equipada con la operacion de resta entre su error y el error anterior tomando su tasa de cambio en el tiempo, suavizando oscilaciones. Y la integral resulta ser un acumulador del error con respect al cambio en el tiempo, por lo que si se ajusta mal puede acumular demasiado error.

Al inicio del programa se lee el estado actual del sistema enfatizando en el ángulo del péndulo. Usandoe l ángulo actual calcula la “fuerza_pid” y suma los tres términos aplicados del PID para ver cuanta fuerza realmente se necesita para el control del sistema. Así mismo, se aplico una fuerza de perturbación con las teclas “A” y “D” para poder añadrile dificultad al sistema. Todos esos parámetros de lo que ocurre al sistema al final pasa al modelado que los representa matemáticamente y el Pygame lo hace gráfico.

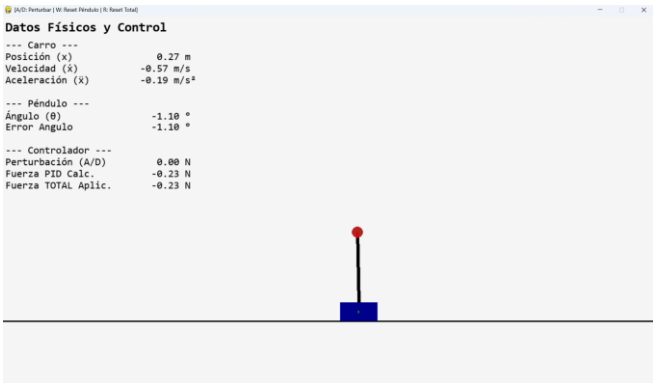


*Fig. 3. Control PID por Python. Elaboración propia.*

Esete sistema mostrado en la figura 3 resulta ser bastante efectivo si se maneja de forma coherente las ganancias, ya que puede que los controladores no salgan efectivos si son ganancias altas o bajas si no se toman en cuenta las constantes. En el caso de ese sistema unas ganancias de Kp de 70, Kd de 45 y Ki de 30 resultaron efectivos para un sistema con las siguientes constantes:

M = 1.0 Kg
m = 0.2 Kg
L = 0.6 m
g = 9.81 m/s^2
b = 0.1

Por lo que si se alteran las ganancias puede que el sistema oscile más o no responda de forma efectiva a las perturbaciones de 50N que se le aplican en la simulación.

### *3. Control difuso*

Este controlador resulta ser el más Complejo en su funcionamiento en el cual toma “valores crisp” del mundo real y las convierte en los conceptos lingüísticos y “difusos” para tomar decisiones basadas en reglas de sentido común.

El programa se divide en tres etapas clave todas implementadas en el método de “calcular_fuerza”.

Para empezar, el programa empieza con la Fuzzificación, tomando dos entradas numéricas de ángulo y velocidad angular. Dividiendo ángulo en 4 conjuntos:

- Amn: Ángulo muy negativo.
- An: Ángulo negativo.
- Ap: Ángulo positivo.
- Amp: Ángulo muy positivo.

Y dividiendo a la velocidad angular en dos conjuntos:

- Vn: Velocidad negativa.
- Vp: Velocidad positiva.

El programa usa “_funcion_membresia_trapezoidal”, en el cual por cada valor numérico ingresado calcula su grado de pertenencia en cada uno de los conjuntos difusos. Teniendo al final un conjunto de activaciones para cada categoría.

Ya para el motor de inferencia se aplican las reglas planeadas en el código, siendo 8 en total. El “min()” simula un operador lógico AND, para luego ser usadas las respuestas de cada regla en una combinación final para determinar la fuerza de activación de cada conjunto difuso de salida.

Y para la Defuzzificación convierte lo que es una idea hasta ahora en una vuelta a un valor numérico coherente en Newtons que la simulación pueda ejecutar dependiendo la necesidad del control. La variable de salida es la fuerza que se divide en 5 conjuntos:

- Xmn: Muy negativa.
- Xn: Negativa.
- X0: Cero.
- Xp: Positiva.
- Xmp: Muy positiva.

Este controlador utiliza el método de centroide, la cual crea una forma geométrica compuesta (superficie_agregada) cortando cada función de membresía de salida a la altura de su fuerza de activación calculada anteriormente, para después calcular el centro de gravedad y la posición de este centroide es el valor numérico final que debe de ser utilizado como fuerza final en el controlador. Este control produce una respuesta suave, rapida y equilibrada.

Las reglas y valores utilizados fueron los siguientes:

**Ángulo (rad):**

- 'amn' (muy negativo): [-0.8, -0.8, -0.5, -0.3]
- 'an' (negativo): [-0.5, -0.3, -0.2, 0.0]
- 'ap' (positivo): [0.0, 0.2, 0.3, 0.5]

- 'amp' (muy positivo): [0.3, 0.5, 0.8, 0.8]

**Velocidad Angular (rad/s):**

- 'vn' (negativa): [-3.0, -3.0, -1.0, 0.0]
- 'vp' (positiva): [0.0, 1.0, 3.0, 3.0]

**Funciones de Membresía de Salida (Trapezoidales):**

- 'xmn' (muy negativa): [-15, -15, -12, -10]
- 'xn' (negativa): [-12, -10, -8, -5]
- 'x0' (cero): [-5, -2, 2, 5]
- 'xp' (positiva): [5, 8, 10, 12]
- 'xmp' (muy positiva): [10, 12, 15, 15]

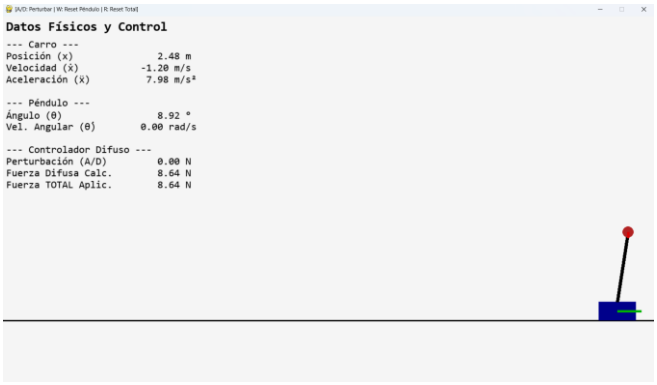


*Fig. 4. Control difuso en Python. Elaboración propia.*

La figura 4 muestra el desarrollo del control difuso con las mismas gráficas, esta dentro de la experiencia de la simulación tiene respuestas un poco más demoradas con respecto al movimiento del carro, aunque resulta tener nula oscilación y mucha estabilidad final en su aplicación, resultando ventajoso ese aspecto por encima del PID. Ajustando major las reglas se pueden llegar a respuestas más rápidas tieniendo en cuenta el valor del ángulo.

### *4. Control Webots PID*

Este controlador PID deja evidenciar en ambiente real como se comporta un controlador y un sistema bien acondicionado en un ambiente más realista que una simulación realizada por Python. Conectando sensors y acoples al carro para poder efectuar mediciones para su controlador, así mismo se emplea una ligera inclinación al inicio para que pueda empezar inestable y se pueda ver sus movimientos, y dentro de sus funciones así mismo se consideran aún perturbaciones con las teclas “A” y “D” para poder estimular mucho el uso del controlador en el programa. Las ganancias usadas en el sistema de PID fueron de Kp de 200, Ki de 0.05 y Kd de 0.5, dejando más fuerza a la respuesta rápida y no tanto a tomar en cuenta la acumulación de error de forma tan drástica y una corrección anticipada leve.

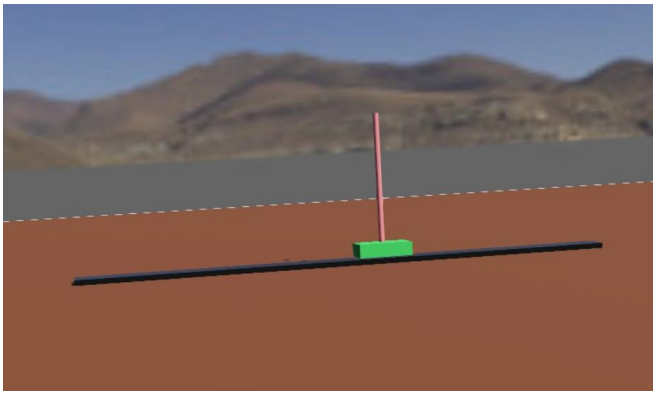

*Fig. 5. Simulación por Webots. Elaboración propia.*

La figura 5 muestra el ambiente realista del webots, manejando sólidos ya en 3D, dimensionando major los pesos, la estructura del péndulo y dejando el modelo más preciso. El control PID en este ambiente resultó ser más caótico al tener más variables a controlar, pero se pudo recalar en un PID optimizado y bien reducido en oscilaciones. Las ganancias dejaron ver un sistema rápido, con bajas oscilaciones por sus propias constantes características, resultando en la major optimización posible para el controlador.

## IV. CONCLUSIONES

Tras la implementación y prueba de dos estrategias de control automático, este proyecto llega a una conclusión clara. Tanto el controlador PID como el de Lógica Difusa lograron estabilizar exitosamente el sistema de péndulo invertido, pero lo hicieron de maneras fundamentalmente distintas.

El controlador PID fue el más rápido. Su enfoque matemático le permitió reaccionar con fuerza e inmediatez ante cualquier error, devolviendo el péndulo a la vertical en un tiempo mínimo. Sin embargo, esta velocidad tuvo un costo: su comportamiento fue visiblemente oscilante y brusco, con correcciones que se sentían mecánicamente agresivas.

Por otro lado, el controlador de Lógica Difusa priorizó la suavidad sobre la velocidad. Aunque tardó un poco más en alcanzar la estabilidad perfecta, su trayectoria hacia ella fue impecablemente fluida y sin ninguna oscilación. La respuesta del sistema se sintió más natural e inteligente, absorbiendo las perturbaciones de manera controlada.

La conclusión final es que la elección del controlador depende del objetivo de la aplicación. Mientras el PID ofrece velocidad y precisión a través de una fuerza bruta algorítmica, el controlador de Lógica Difusa ofrece una estabilidad de mayor calidad: suave, predecible y robusta. Para la mayoría de las aplicaciones físicas y robóticas, este tipo de control superior justificaría con creces el ligero sacrificio en el tiempo de respuesta.

## V. REFERENCIAS

[1] K. Ogata, Modern Control Engineering, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.

[2] T. J. Ross, Fuzzy Logic with Engineering Applications, 4th ed. Chichester, UK: John Wiley & Sons, 2017.

[3] G. J. Klir and B. Yuan, Fuzzy Sets and Fuzzy Logic: Theory and Applications. Upper Saddle River, NJ, USA: Prentice Hall, 1995.

ANEXOS

**Control Manual Python**

```
import pygame
import numpy as np
import math

#
================================================================================
# SECCIÓN 1: VARIABLES CONFIGURABLES
#
================================================================================
# --- Física del Sistema ---
M, m, L, g = 1.0, 0.2, 0.6, 9.81
I = (1/3) * m * L**2
b = 0.1

# --- Parámetros de Simulación y Control ---
FUERZA_MAG = 10
dt = 0.01
estado_inicial = np.array([0.0, 0.0, math.radians(0.1), 0.0]) # Empezar arriba

# --- Parámetros de Visualización de Vectores ---
FUERZA_VECTOR_ESCALA = 8  # Píxeles por Newton.
COLOR_FUERZA = (0, 180, 0)
COLOR_GRAVEDAD = (220, 0, 0)

#
================================================================================
# SECCIÓN 2: MODELO DINÁMICO (Sin cambios)
#
================================================================================
def modelo_pendulo_carro(estado_actual, fuerza_aplicada):
    _, _, theta, theta_dot = estado_actual
    sin_t, cos_t = math.sin(theta), math.cos(theta)
    A, B = M + m, fuerza_aplicada + m * L * (theta_dot**2) * sin_t
    den = A * (I + m * L**2) - (m * L * cos_t)**2
    if abs(den) < 1e-6: return np.array([0, 0, 0, 0])
    theta_ddot = ((A*m*g*L*sin_t) - (m*L*cos_t*B) - (A*b*theta_dot)) / den
    x_ddot = ((B*(I+m*L**2)) - (m*L*cos_t)*(m*g*L*sin_t-b*theta_dot)) / den
    return np.array([estado_actual[1], x_ddot, estado_actual[3], theta_ddot])

def actualizar_estado(estado_actual, fuerza_aplicada, dt):
    derivada = modelo_pendulo_carro(estado_actual, fuerza_aplicada)
    return estado_actual + derivada * dt

#
================================================================================
# SECCIÓN 3: SIMULACIÓN CON HUD Y VISUALIZACIÓN DE FUERZAS
#
================================================================================
def ejecutar_sandbox_visual():
    pygame.init()
    ANCHO, ALTO = 1400, 800
```

```
PANTALLA = pygame.display.set_mode((ANCHO, ALTO))
pygame.display.set_caption("[A/D: Mover | W: Resetear Péndulo | R: Resetear Todo]")
RELOJ = pygame.time.Clock()
FUENTE_DATOS = pygame.font.SysFont('Consolas', 22)
FUENTE_TITULO = pygame.font.SysFont('Consolas', 28, bold=True)
ESCALA_MUNDO = 250

estado = np.copy(estado_inicial)

while True:
    # --- Manejo de Eventos ---
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT: pygame.quit(); return
        if evento.type == pygame.KEYDOWN:
            if evento.key == pygame.K_r: estado = np.copy(estado_inicial)
            if evento.key == pygame.K_w:
                estado[2], estado[3] = estado_inicial[2], estado_inicial[3]

    # --- Control y Física ---
    teclas = pygame.key.get_pressed()
    fuerza_aplicada = 0.0
    if teclas[pygame.K_a]: fuerza_aplicada = -FUERZA_MAG
    if teclas[pygame.K_d]: fuerza_aplicada = FUERZA_MAG

    derivada_actual = modelo_pendulo_carro(estado, fuerza_aplicada)
    estado = estado + derivada_actual * dt
    x, x_dot, theta, _ = estado
    x_ddot = derivada_actual[1]

    # --- LÓGICA DE PAREDES (CORREGIDA) ---
    carro_ancho_px = 80
    pos_carro_px = ANCHO/2 + x * ESCALA_MUNDO
    lim_izq, lim_der = carro_ancho_px/2, ANCHO - carro_ancho_px/2

    if pos_carro_px <= lim_izq:
        estado[0] = (lim_izq - ANCHO/2) / ESCALA_MUNDO
        if estado[1] < 0: estado[1] = 0
    elif pos_carro_px >= lim_der:
        estado[0] = (lim_der - ANCHO/2) / ESCALA_MUNDO
        if estado[1] > 0: estado[1] = 0

    # --- Dibujado en Pantalla ---
    PANTALLA.fill((245, 245, 245))
    suelo_y = ALTO-150
    pygame.draw.line(PANTALLA, (0,0,0), (0, suelo_y), (ANCHO, suelo_y), 3)

    # Objetos
    pos_carro_px_final = ANCHO/2 + estado[0]*ESCALA_MUNDO
    carro_rect = pygame.Rect(pos_carro_px_final-carro_ancho_px/2, suelo_y-40, carro_ancho_px, 40)
    pygame.draw.rect(PANTALLA, (0,0,139), carro_rect)
    pivote_x, pivote_y = carro_rect.centerx, carro_rect.top
    extremo_x = pivote_x + L*ESCALA_MUNDO*math.sin(theta)
    extremo_y = pivote_y - L*ESCALA_MUNDO*math.cos(theta)
    pygame.draw.line(PANTALLA, (0,0,0), (pivote_x, pivote_y), (extremo_x, extremo_y), 7)
    pygame.draw.circle(PANTALLA, (178,34,34), (int(extremo_x), int(extremo_y)), 12)
```

```
        # Dibujar Vectores de Fuerza
        if abs(fuerza_aplicada) > 0:
            longitud_vector_f = fuerza_aplicada * FUERZA_VECTOR_ESCALA
            pygame.draw.line(PANTALLA, COLOR_FUERZA,
                             (carro_rect.centerx, carro_rect.centery),
                             (carro_rect.centerx + longitud_vector_f, carro_rect.centery), 5)

        fuerza_g = m * g
        longitud_vector_g = fuerza_g * FUERZA_VECTOR_ESCALA
        pygame.draw.line(PANTALLA, COLOR_GRAVEDAD,
                         (extremo_x, extremo_y),
                         (extremo_x, extremo_y + longitud_vector_g), 5)

        # --- Dibujar HUD de Física ---
        # --- CORRECCIÓN SUTIL PARA CLARIDAD DEL ÁNGULO ---
        # Normalizamos el ángulo a un rango de -pi a +pi antes de mostrarlo.
        # Esto no afecta la física, solo la visualización para que sea más intuitiva.
        angulo_visual = (theta + np.pi) % (2 * np.pi) - np.pi

        hud_items = [
            ("--- Carro ---", ""),
            ("Posición (x)", f"{x:8.2f} m"),
            ("Velocidad (ẋ)", f"{x_dot:8.2f} m/s"),
            ("Aceleración (ẍ)", f"{x_ddot:8.2f} m/s²"),
            ("", ""),
            ("--- Péndulo ---", ""),
            ("Ángulo (θ)", f"{math.degrees(angulo_visual):8.2f} °"), # Usamos la variable normalizada
            ("", ""),
            ("--- Control ---", ""),
            ("Fuerza Aplicada (F)", f"{fuerza_aplicada:8.2f} N")
        ]

        PANTALLA.blit(FUENTE_TITULO.render("Datos Físicos en Vivo", True, (0,0,0)), (10, 10))
        for i, (label, value) in enumerate(hud_items):
            texto = f"{label:<20} {value:>12}"
            render_texto = FUENTE_DATOS.render(texto, True, (0,0,0))
            PANTALLA.blit(render_texto, (10, 50 + i * 25))

        pygame.display.flip()
        RELOJ.tick(1 / dt)

if __name__ == "__main__":
    ejecutar_sandbox_visual()
```

**Control PID Python**

```
import pygame
import numpy as np
import math

#
# ==============================================================================
# SECCIÓN 1: VARIABLES CONFIGURABLES
#
# ==============================================================================
# --- Física del Sistema ---
M, m, L, g = 1.0, 0.2, 0.6, 9.81
I = (1/3) * m * L**2
b = 0.1 # Fricción en la articulación del péndulo

# --- Parámetros del Controlador PID ---
Kp = 70.0
Ki = 30.0
Kd = 45.0
SETPOINT_ANGULO = 0.0

# --- Parámetros de Simulación y Control ---
FUERZA_MAG = 15.0 # LÍMITE máximo de fuerza del motor.
# --- ¡NUEVO! Fuerza para las perturbaciones del teclado ---
FUERZA_PERTURBACION_MAG = 50 # Empuje pequeño y constante al presionar A/D.

dt = 0.01
estado_inicial = np.array([0.0, 0.0, math.radians(1.5), 0.0])

# --- Parámetros de Visualización de Vectores ---
FUERZA_VECTOR_ESCALA = 6
COLOR_FUERZA = (0, 180, 0)
COLOR_GRAVEDAD = (220, 0, 0)

#
# ==============================================================================
# SECCIÓN 2: MODELO DINÁMICO (Sin cambios)
#
# ==============================================================================
def modelo_pendulo_carro(estado_actual, fuerza_aplicada):
    _, _, theta, theta_dot = estado_actual
    sin_t, cos_t = math.sin(theta), math.cos(theta)
    A, B = M + m, fuerza_aplicada + m * L * (theta_dot**2) * sin_t
    den = A * (I + m * L**2) - (m * L * cos_t)**2
    if abs(den) < 1e-6: return np.array([0, 0, 0, 0])
    theta_ddot = ((A*m*g*L*sin_t) - (m*L*cos_t*B) - (A*b*theta_dot)) / den
    x_ddot = ((B*(I+m*L**2)) - (m*L*cos_t)*(m*g*L*sin_t-b*theta_dot)) / den
    return np.array([estado_actual[1], x_ddot, estado_actual[3], theta_ddot])

def actualizar_estado(estado_actual, fuerza_aplicada, dt):
    derivada = modelo_pendulo_carro(estado_actual, fuerza_aplicada)
    return estado_actual + derivada * dt

#
# ==============================================================================
```

```
# SECCIÓN 3: SIMULACIÓN CON CONTROLADOR PID Y PERTURBACIONES
#
================================================================================
def ejecutar_sandbox_visual():
    pygame.init()
    ANCHO, ALTO = 1400, 800
    PANTALLA = pygame.display.set_mode((ANCHO, ALTO))
    pygame.display.set_caption("[A/D: Perturbar | W: Reset Péndulo | R: Reset Total]")
    RELOJ = pygame.time.Clock()
    FUENTE_DATOS = pygame.font.SysFont('Consolas', 22)
    FUENTE_TITULO = pygame.font.SysFont('Consolas', 28, bold=True)
    ESCALA_MUNDO = 250

    estado = np.copy(estado_inicial)
    integral = 0.0
    previous_error = 0.0

    while True:
        # --- Manejo de Eventos ---
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT: pygame.quit(); return
            if evento.type == pygame.KEYDOWN:
                if evento.key == pygame.K_r:
                    estado = np.copy(estado_inicial)
                    integral, previous_error = 0.0, 0.0
                if evento.key == pygame.K_w:
                    estado[2], estado[3] = estado_inicial[2], estado_inicial[3]

        # --- ¡MODIFICADO! Lógica de Control ---

        # 1. Capturar la intención de perturbación del usuario
        teclas = pygame.key.get_pressed()
        fuerza_perturbacion = 0.0
        if teclas[pygame.K_a]: fuerza_perturbacion = -FUERZA_PERTURBACION_MAG
        if teclas[pygame.K_d]: fuerza_perturbacion = FUERZA_PERTURBACION_MAG

        # 2. El controlador PID calcula la fuerza que CREE necesaria para estabilizar
        x, x_dot, theta, theta_dot = estado
        error = theta - SETPOINT_ANGULO
        integral += error * dt
        derivative = (error - previous_error) / dt
        fuerza_pid = (Kp * error) + (Ki * integral) + (Kd * derivative)

        # 3. La fuerza total es la suma de la corrección del PID y la perturbación del usuario
        fuerza_aplicada = fuerza_pid + fuerza_perturbacion

        # 4. Saturar la FUERZA TOTAL para que no exceda los límites del motor
        fuerza_aplicada = np.clip(fuerza_aplicada, -FUERZA_MAG, FUERZA_MAG)

        # 5. Actualizar el estado del PID para la siguiente iteración
        previous_error = error

        # --- Actualización de la Física ---
        derivada_actual = modelo_pendulo_carro(estado, fuerza_aplicada)
        estado = estado + derivada_actual * dt
        x_ddot = derivada_actual[1]
```

```
        # --- LÓGICA DE PAREDES (sin cambios) ---
        carro_ancho_px = 80
        pos_carro_px = ANCHO/2 + x * ESCALA_MUNDO
        lim_izq, lim_der = carro_ancho_px/2, ANCHO - carro_ancho_px/2
        if pos_carro_px <= lim_izq:
            estado[0] = (lim_izq - ANCHO/2) / ESCALA_MUNDO
            if estado[1] < 0: estado[1] = 0
        elif pos_carro_px >= lim_der:
            estado[0] = (lim_der - ANCHO/2) / ESCALA_MUNDO
            if estado[1] > 0: estado[1] = 0

        # --- Dibujado en Pantalla ---
        PANTALLA.fill((245, 245, 245))
        suelo_y = ALTO-150
        pygame.draw.line(PANTALLA, (0,0,0), (0, suelo_y), (ANCHO, suelo_y), 3)

        # Objetos (sin cambios)
        pos_carro_px_final = ANCHO/2 + estado[0]*ESCALA_MUNDO
        carro_rect = pygame.Rect(pos_carro_px_final-carro_ancho_px/2, suelo_y-40, carro_ancho_px, 40)
        pygame.draw.rect(PANTALLA, (0,0,139), carro_rect)
        pivote_x, pivote_y = carro_rect.centerx, carro_rect.top
        extremo_x = pivote_x + L*ESCALA_MUNDO*math.sin(theta)
        extremo_y = pivote_y - L*ESCALA_MUNDO*math.cos(theta)
        pygame.draw.line(PANTALLA, (0,0,0), (pivote_x, pivote_y), (extremo_x, extremo_y), 7)
        pygame.draw.circle(PANTALLA, (178,34,34), (int(extremo_x), int(extremo_y)), 12)

        # Vectores de Fuerza (sin cambios)
        if abs(fuerza_aplicada) > 0.01:
            longitud_vector_f = fuerza_aplicada * FUERZA_VECTOR_ESCALA
            pygame.draw.line(PANTALLA, COLOR_FUERZA,
                             (carro_rect.centerx, carro_rect.centery),
                             (carro_rect.centerx + longitud_vector_f, carro_rect.centery), 5)
        fuerza_g = m * g
        longitud_vector_g = fuerza_g * FUERZA_VECTOR_ESCALA
            pygame.draw.line(PANTALLA, COLOR_GRAVEDAD, (extremo_x, extremo_y), (extremo_x,
extremo_y + longitud_vector_g), 5)

        # --- ¡MODIFICADO! HUD de Física y Control ---
        angulo_visual = (theta + np.pi) % (2 * np.pi) - np.pi
        hud_items = [
            ("--- Carro ---", ""),
            ("Posición (x)", f"{x:8.2f} m"),
            ("Velocidad (ẋ)", f"{x_dot:8.2f} m/s"),
            ("Aceleración (ẍ)", f"{x_ddot:8.2f} m/s²"),
            ("", ""),
            ("--- Péndulo ---", ""),
            ("Ángulo (θ)", f"{math.degrees(angulo_visual):8.2f} °"),
            ("Error Angulo", f"{math.degrees(error):8.2f} °"),
            ("", ""),
            ("--- Controlador ---", ""),
            ("Perturbación (A/D)", f"{fuerza_perturbacion:8.2f} N"),
            ("Fuerza PID Calc.", f"{fuerza_pid:8.2f} N"),
            ("Fuerza TOTAL Aplic.", f"{fuerza_aplicada:8.2f} N"),
        ]
```

```
        PANTALLA.blit(FUENTE_TITULO.render("Datos Físicos y Control", True, (0,0,0)), (10, 10))
        for i, (label, value) in enumerate(hud_items):
            texto = f"{label:<20} {value:>12}"
            render_texto = FUENTE_DATOS.render(texto, True, (0,0,0))
            PANTALLA.blit(render_texto, (10, 50 + i * 25))

        pygame.display.flip()
        RELOJ.tick(1 / dt)

if __name__ == "__main__":
    ejecutar_sandbox_visual()
```

**Control Difuso Python**

```
import pygame
import numpy as np
import math

#
# ============================================================================
# SECCIÓN 1: CONTROLADOR DIFUSO (Sin cambios)
#
# ============================================================================
class ControladorDifuso:
    """
    Controlador difuso Mamdani con inferencia Max-Min para el péndulo invertido.
    No utiliza ninguna biblioteca externa de lógica difusa.
    """
    def __init__(self):
        # --- 1. Definir universos de discurso (rangos de operación) ---
        self.universo_angulo = [-0.8, 0.8]  # rad (aprox. -45 a +45 grados)
        self.universo_vel_angular = [-3, 3] # rad/s
        self.universo_fuerza = [-15, 15]   # Newtons (coincide con FUERZA_MAG)

        # --- 2. Definir los conjuntos difusos de ENTRADA (Membresía Trapezoidal) ---
        self.mf_angulo = {
            'amn': [-0.8, -0.8, -0.5, -0.3], 'an': [-0.5, -0.3, -0.2, 0.0],
            'ap': [0.0, 0.2, 0.3, 0.5], 'amp': [0.3, 0.5, 0.8, 0.8]
        }
        self.mf_vel_angular = { 'vn': [-3.0, -3.0, -1.0, 0.0], 'vp': [0.0, 1.0, 3.0, 3.0] }

        # --- 3. Definir los conjuntos difusos de SALIDA (Membresía Trapezoidal) ---
        self.mf_fuerza = {
            'xmn': [-15, -15, -12, -10], 'xn': [-12, -10, -8, -5],
            'x0': [-5, -2, 2, 5], 'xp': [5, 8, 10, 12], 'xmp': [10, 12, 15, 15]
        }
        self.puntos_salida = np.linspace(self.universo_fuerza[0], self.universo_fuerza[1], 100)

    def _funcion_membresia_trapezoidal(self, x, params):
        a, b, c, d = params
        if x <= a or x >= d: return 0.0
        elif a < x < b: return (x - a) / (b - a) if b != a else 1.0
        elif b <= x <= c: return 1.0
        elif c < x < d: return (d - x) / (d - c) if d != c else 1.0
        return 0.0
```

```
    def calcular_fuerza(self, angulo, vel_angular):
        memb_angulo = {k: self._funcion_membresia_trapezoidal(angulo, p) for k, p in self.mf_angulo.items()}
        memb_vel = {k: self._funcion_membresia_trapezoidal(vel_angular, p) for k, p in self.mf_vel_angular.items()}

        fuerza_r1 = min(memb_angulo['ap'], memb_vel['vp'])
        fuerza_r2 = min(memb_angulo['ap'], memb_vel['vn'])
        fuerza_r3 = min(memb_angulo['an'], memb_vel['vp'])
        fuerza_r4 = min(memb_angulo['an'], memb_vel['vn'])
        fuerza_r5 = min(memb_angulo['amp'], memb_vel['vp'])
        fuerza_r6 = min(memb_angulo['amp'], memb_vel['vn'])
        fuerza_r7 = min(memb_angulo['amn'], memb_vel['vp'])
        fuerza_r8 = min(memb_angulo['amn'], memb_vel['vn'])

        fuerza_activacion = {
            'xmp': fuerza_r5, 'xp': max(fuerza_r1, fuerza_r6),
            'x0': max(fuerza_r2, fuerza_r3), 'xn': max(fuerza_r4, fuerza_r7),
            'xmn': fuerza_r8
        }

        superficie_agregada = np.zeros_like(self.puntos_salida)
        for i, z in enumerate(self.puntos_salida):
            max_grado_membresia = 0.0
            for consecuente, fuerza_regla in fuerza_activacion.items():
                grado_memb = self._funcion_membresia_trapezoidal(z, self.mf_fuerza[consecuente])
                membresia_activada = min(fuerza_regla, grado_memb)
                if membresia_activada > max_grado_membresia:
                    max_grado_membresia = membresia_activada
            superficie_agregada[i] = max_grado_membresia

        numerador = np.sum(self.puntos_salida * superficie_agregada)
        denominador = np.sum(superficie_agregada)
        return 0.0 if abs(denominador) < 1e-6 else numerador / denominador

#
# ============================================================================
# SECCIÓN 2: VARIABLES CONFIGURABLES (Sin cambios)
#
# ============================================================================
M, m, L, g = 1.0, 0.2, 0.6, 9.81
I = (1/3) * m * L**2
b = 0.1
FUERZA_MAG = 15.0
FUERZA_PERTURBACION_MAG = 25
dt = 0.01
estado_inicial = np.array([0.0, 0.0, math.radians(1.5), 0.0])
FUERZA_VECTOR_ESCALA = 6
COLOR_FUERZA = (0, 180, 0)
COLOR_GRAVEDAD = (220, 0, 0)

#
# ============================================================================
# SECCIÓN 3: MODELO DINÁMICO (Sin cambios)
#
# ============================================================================
```

```
def modelo_pendulo_carro(estado_actual, fuerza_aplicada):
    _, _, theta, theta_dot = estado_actual
    sin_t, cos_t = math.sin(theta), math.cos(theta)
    A, B = M + m, fuerza_aplicada + m * L * (theta_dot**2) * sin_t
    den = A * (I + m * L**2) - (m * L * cos_t)**2
    if abs(den) < 1e-6: return np.array([0, 0, 0, 0])
    theta_ddot = ((A*m*g*L*sin_t) - (m*L*cos_t*B) - (A*b*theta_dot)) / den
    x_ddot = ((B*(I+m*L**2)) - (m*L*cos_t)*(m*g*L*sin_t-b*theta_dot)) / den
    return np.array([estado_actual[1], x_ddot, estado_actual[3], theta_ddot])

#
==============================================================================
# SECCIÓN 4: SIMULACIÓN CON CONTROLADOR DIFUSO Y PERTURBACIONES
#
==============================================================================
def ejecutar_sandbox_visual():
    pygame.init()
    ANCHO, ALTO = 1400, 800
    PANTALLA = pygame.display.set_mode((ANCHO, ALTO))
    pygame.display.set_caption("[A/D: Perturbar | W: Reset Péndulo | R: Reset Total]")
    RELOJ = pygame.time.Clock()
    FUENTE_DATOS = pygame.font.SysFont('Consolas', 22)
    FUENTE_TITULO = pygame.font.SysFont('Consolas', 28, bold=True)
    ESCALA_MUNDO = 250

    estado = np.copy(estado_inicial)
    controlador_difuso = ControladorDifuso()

    while True:
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT: pygame.quit(); return
            if evento.type == pygame.KEYDOWN:
                if evento.key == pygame.K_r: estado = np.copy(estado_inicial)
                if evento.key == pygame.K_w: estado[2], estado[3] = estado_inicial[2], estado_inicial[3]

        teclas = pygame.key.get_pressed()
        fuerza_perturbacion = 0.0
        if teclas[pygame.K_a]: fuerza_perturbacion = -FUERZA_PERTURBACION_MAG
        if teclas[pygame.K_d]: fuerza_perturbacion = FUERZA_PERTURBACION_MAG

        x, x_dot, theta, theta_dot = estado
        fuerza_controlador = controlador_difuso.calcular_fuerza(theta, theta_dot)
        fuerza_aplicada = fuerza_controlador + fuerza_perturbacion
        fuerza_aplicada = np.clip(fuerza_aplicada, -FUERZA_MAG, FUERZA_MAG)

        derivada_actual = modelo_pendulo_carro(estado, fuerza_aplicada)
        estado = estado + derivada_actual * dt
        x_ddot = derivada_actual[1]

        # --- ¡¡¡MODIFICADO!!! LÓGICA DE PAREDES CON REBOTE ---
        COEF_RESTITUCION = 0.5  # Factor de rebote (0=no rebota, 1=rebote perfecto)
        carro_ancho_px = 80
        pos_carro_px = ANCHO / 2 + estado[0] * ESCALA_MUNDO
        lim_izq, lim_der = carro_ancho_px / 2, ANCHO - carro_ancho_px / 2

        # Si se pasa del límite IZQUIERDO y se está MOVIENDO hacia la izquierda
```

```
        if pos_carro_px <= lim_izq and estado[1] < 0:
            estado[0] = (lim_izq - ANCHO / 2) / ESCALA_MUNDO  # Reposicionar en el borde
            estado[1] *= -COEF_RESTITUCION  # Invertir y amortiguar la velocidad (rebote)

        # Si se pasa del límite DERECHO y se está MOVIENDO hacia la derecha
        elif pos_carro_px >= lim_der and estado[1] > 0:
            estado[0] = (lim_der - ANCHO / 2) / ESCALA_MUNDO  # Reposicionar en el borde
            estado[1] *= -COEF_RESTITUCION  # Invertir y amortiguar la velocidad (rebote)

        # --- Dibujado en Pantalla (sin cambios) ---
        PANTALLA.fill((245, 245, 245))
        suelo_y = ALTO - 150
        pygame.draw.line(PANTALLA, (0, 0, 0), (0, suelo_y), (ANCHO, suelo_y), 3)

        pos_carro_px_final = ANCHO / 2 + estado[0] * ESCALA_MUNDO
        carro_rect = pygame.Rect(pos_carro_px_final - carro_ancho_px / 2, suelo_y - 40, carro_ancho_px, 40)
        pygame.draw.rect(PANTALLA, (0, 0, 139), carro_rect)
        pivote_x, pivote_y = carro_rect.centerx, carro_rect.top
        extremo_x = pivote_x + L * ESCALA_MUNDO * math.sin(theta)
        extremo_y = pivote_y - L * ESCALA_MUNDO * math.cos(theta)
        pygame.draw.line(PANTALLA, (0, 0, 0), (pivote_x, pivote_y), (extremo_x, extremo_y), 7)
        pygame.draw.circle(PANTALLA, (178, 34, 34), (int(extremo_x), int(extremo_y)), 12)

        if abs(fuerza_aplicada) > 0.01:
            long_vector_f = fuerza_aplicada * FUERZA_VECTOR_ESCALA
            pygame.draw.line(PANTALLA, COLOR_FUERZA, (carro_rect.centerx, carro_rect.centery),
(carro_rect.centerx + long_vector_f, carro_rect.centery), 5)

        fuerza_g = m * g
        long_vector_g = fuerza_g * FUERZA_VECTOR_ESCALA
        pygame.draw.line(PANTALLA, COLOR_GRAVEDAD, (extremo_x, extremo_y), (extremo_x,
extremo_y + long_vector_g), 5)

        hud_items = [
            ("--- Carro ---", ""), ("Posición (x)", f"{estado[0]:8.2f} m"),
            ("Velocidad (ẋ)", f"{estado[1]:8.2f} m/s"), ("Aceleración (ẍ)", f"{x_ddot:8.2f} m/s²"),
            ("", ""), ("--- Péndulo ---", ""), ("Ángulo (θ)", f"{math.degrees(theta):8.2f} °"),
            ("Vel. Angular (θ̇)", f"{theta_dot:8.2f} rad/s"), ("", ""),
            ("--- Controlador Difuso ---", ""), ("Perturbación (A/D)", f"{fuerza_perturbacion:8.2f} N"),
            ("Fuerza Difusa Calc.", f"{fuerza_controlador:8.2f} N"), ("Fuerza TOTAL Aplic.",
f"{fuerza_aplicada:8.2f} N"),
        ]

        PANTALLA.blit(FUENTE_TITULO.render("Datos Físicos y Control", True, (0, 0, 0)), (10, 10))
        for i, (label, value) in enumerate(hud_items):
            texto = f"{label:<20} {value:>12}"
            PANTALLA.blit(FUENTE_DATOS.render(texto, True, (0, 0, 0)), (10, 50 + i * 25))

        pygame.display.flip()
        RELOJ.tick(1 / dt)

if __name__ == "__main__":
    ejecutar_sandbox_visual()
```

**Control PID Webots**

```
from controller import Robot, Keyboard

# --- Parámetros de Simulación y PID ---
TIME_STEP = 16
MAX_SPEED = 100.0  # Límite de velocidad para el motor

# --- Constantes del PID (VALORES RE-SINTONIZADOS) ---
# Kp reacciona al error actual (ángulo y posición).
Kp = 200  #150
# Ki corrige errores residuales a largo plazo. Se mantiene bajo.
Ki = 0.05  #0.1
# Kd amortigua la reacción y previene oscilaciones.
Kd = 0.5  #25.0

# --- Pesos para el error combinado ---
# La prioridad es mantener el ángulo (mayor peso).
ANGLE_WEIGHT = 0.8
# La segunda prioridad es mantener el carro centrado.
POSITION_WEIGHT = 0.2

# --- Parámetros de Perturbación ---
PERTURBATION_FORCE = 25 # Aumentamos un poco para que se note el empuje

# Inicialización de variables del PID
integral = 0.0
previous_error = 0.0

# --- Creación y Configuración del Robot ---
robot = Robot()

# Obtener dispositivos
pole_angle_sensor = robot.getDevice("pole position sensor")
cart_position_sensor = robot.getDevice("cart position sensor")
cart_motor = robot.getDevice("cart motor")

# Habilitar el teclado
keyboard = robot.getKeyboard()
keyboard.enable(TIME_STEP)

# Habilitar los sensores
pole_angle_sensor.enable(TIME_STEP)
cart_position_sensor.enable(TIME_STEP)

# Configurar el motor para control de fuerza/velocidad
cart_motor.setPosition(float('inf'))
cart_motor.setVelocity(0.0)

print("Controlador PID corregido iniciado para Webots R2025a.")
print("Presiona la ventana de simulación y usa 'A' y 'D' para empujar el carro.")
print(f"Sintonización actual: Kp={Kp}, Ki={Ki}, Kd={Kd}")

# --- Bucle de Control Principal ---
while robot.step(TIME_STEP) != -1:
    # --- 1. Lectura de Sensores ---
```

```
pole_angle = pole_angle_sensor.getValue()
cart_position = cart_position_sensor.getValue()

# --- 2. Cálculo del PID (se ejecuta siempre) ---
# El error es una combinación ponderada del ángulo y la posición del carro.
# El objetivo es que tanto el ángulo como la posición sean 0.
error = (ANGLE_WEIGHT * pole_angle) + (POSITION_WEIGHT * cart_position)

# El término integral solo se acumula si el error es significativo,
# para evitar "windup" cuando el sistema está casi estable.
if abs(error) > 0.01:
    integral += error * (TIME_STEP / 1000.0)

derivative = (error - previous_error) / (TIME_STEP / 1000.0)

# Calcular la fuerza del PID
# El signo es positivo: si el péndulo se inclina a la derecha (+ángulo),
# el carro debe moverse a la derecha (+fuerza) para compensar.
pid_force = (Kp * error) + (Ki * integral) + (Kd * derivative)

previous_error = error

# --- 3. Gestión de Perturbaciones Manuales ---
manual_force = 0.0
key = keyboard.getKey()
if key == ord('A') or key == ord('a'):
    manual_force = -PERTURBATION_FORCE
elif key == ord('D') or key == ord('d'):
    manual_force = PERTURBATION_FORCE

# --- 4. Aplicación de la Fuerza Total ---
# La fuerza total es la del PID más la perturbación manual.
total_force = pid_force + manual_force

# Limitar la velocidad para que la simulación sea más estable
current_velocity = cart_motor.getVelocity()
if total_force > 0 and current_velocity > MAX_SPEED:
    total_force = 0
elif total_force < 0 and current_velocity < -MAX_SPEED:
    total_force = 0

cart_motor.setForce(total_force)
```

**Configuración Mundo Webots**

```
#VRML_SIM R2025a utf8

EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/protos/Textured
Background.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/protos/Textured
BackgroundLight.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/floors/protos/Floor.proto"

WorldInfo {
 info [
  "Inverted pendulum on a track"
 ]
 title "Inverted Pendulum"
 basicTimeStep 16
}
Viewpoint {
 orientation 0.2289007734433325 -0.1341884972551655 -0.9641565656683798 3.9481241236692872
 position 3.176136682798244 -4.2974290170900815 2.173719868214627
}
TexturedBackground {}
TexturedBackgroundLight {}
# CAMBIO: Se ha modificado el Floor para cambiar su apariencia.
Floor {
 size 10 10
 appearance PBRAppearance {
  baseColor 0.4 0.2 0.1  # Color marrón oscuro
  roughness 0.8
  metalness 0
 }
}
DEF TRACK Solid {
 translation 0 0 0.01
 children [
  Shape {
   appearance PBRAppearance {
    baseColor 0.2 0.2 0.2
    roughness 0.2
   }
   geometry Box {
    size 2 0.1 0.02
   }
  }
 ]
 name "track"
 boundingObject Box {
  size 2 0.1 0.02
 }
}
Robot {
 children [
  SliderJoint {
```

```
jointParameters JointParameters {
  position 0
  axis 1 0 0
  minStop -0.8
  maxStop 0.8
}
device [
  LinearMotor {
    name "cart motor"
    maxForce 20
  }
  PositionSensor {
    name "cart position sensor"
  }
]
endPoint DEF CART Solid {
  translation 0 0 0.05
  rotation 0 0 1 0
  children [
    Shape {
      appearance PBRAppearance {
        # CAMBIO: El color base del carro ahora es verde.
        baseColor 0.2 0.8 0.3
        roughness 1
        metalness 0
      }
      geometry Box {
        size 0.2 0.1 0.05
      }
    }
    HingeJoint {
      jointParameters HingeJointParameters {
        position 0.02
        axis 0 1 0
        anchor 0 0 0.025
      }
      device [
        PositionSensor {
          name "pole position sensor"
        }
      ]
      endPoint Solid {
        translation 0 0 0.25
        rotation 0 0 1 0
        children [
          DEF POLE Shape {
            appearance PBRAppearance {
              baseColor 1 0.5 0.5
              roughness 1
              metalness 0
            }
            geometry Cylinder {
              height 0.5
              radius 0.01
            }
          }
```

```
            ]
            name "pole"
            boundingObject Cylinder {
              height 0.5
              radius 0.01
            }
            physics Physics {
              mass 0.1
            }
          }
        }
      ]
      name "cart"
      boundingObject Box {
        size 0.2 0.1 0.05
      }
      physics Physics {
        mass 1
      }
    }
  }
 ]
 name "inverted_pendulum"
 controller "inverted_pendulum_controller"
}
```