Control PID y Difuso

Julián Ballesteros Rodríguez ,*est.julian.ballest@unimilitar.edu.co, Ingeniería Mecatrónica*
Universidad Militar Nueva Granada
Cajicá, Colombia

***Abstract*— This report presents the design, implementation, and comparative analysis of two control strategies applied to the inverted pendulum problem: a classical PID controller and a fuzzy logic controller. The PID controller was implemented and simulated in the Webots environment, achieving system stabilization through proper parameter tuning. The fuzzy controller was developed in Python using membership functions and fuzzy rules, enabling flexible control without requiring an exact mathematical model. The results show that the PID offers simplicity and good performance under linear and predictable conditions, while the fuzzy controller provides robustness and adaptability when dealing with the nonlinear nature of the system. Thus, both approaches are considered complementary tools for solving control problems in complex dynamic systems.**

Keywords— Inverted Pendulum, PID Control, Fuzzy Logic, Dynamic Systems, Webots Simulation, Nonlinear Control.

I. INTRODUCCION

En el ámbito del control automático, los controladores juegan un papel fundamental en la estabilidad y desempeño de los sistemas dinámicos. Entre las estrategias más utilizadas se encuentran los controladores PID (Proporcional–Integral–Derivativo), ampliamente empleados en la industria debido a su simplicidad, robustez y facilidad de implementación. Este tipo de control se basa en el ajuste de tres parámetros que permiten reducir el error, corregir desviaciones acumuladas y anticipar variaciones futuras del sistema.

Por otro lado, con el avance de la inteligencia artificial y las técnicas de lógica computacional, ha surgido el control difuso, el cual no requiere un modelo matemático preciso del sistema, sino que se apoya en reglas lingüísticas y funciones de pertenencia que imitan la toma de decisiones humana. Esta estrategia resulta especialmente útil en sistemas no lineales o con incertidumbre.

En este informe se presentan dos enfoques de control aplicados y simulados mediante herramientas computacionales: un controlador PID implementado y probado en el entorno de simulación Webots, y un controlador difuso desarrollado íntegramente en Python. A través de estas implementaciones se busca analizar las características de cada método, comparar sus ventajas y limitaciones, y resaltar su aplicabilidad en diferentes escenarios de la ingeniería de control.

II. DESARROLLO

Control PID

El controlador PID (Proporcional–Integral–Derivativo) es uno de los métodos de control más utilizados en la industria debido a su sencillez y eficacia. Su principio se basa en calcular la señal de control a partir de tres acciones fundamentales:

Acción proporcional (P): responde de manera inmediata al error actual (la diferencia entre la referencia y la salida del sistema). Un aumento en la ganancia proporcional mejora la rapidez de respuesta, pero puede generar oscilaciones si es demasiado alto.

Acción integral (I): acumula el error a lo largo del tiempo y lo corrige, eliminando el error estacionario que no puede compensar la acción proporcional por sí sola.

Acción derivativa (D): anticipa el comportamiento futuro del error al considerar su tasa de cambio, lo que contribuye a reducir las oscilaciones y mejora la estabilidad del sistema.

La combinación de estas tres acciones permite que el controlador PID equilibre rapidez, estabilidad y precisión. En la práctica, el ajuste adecuado de sus parámetros (Kp, Ki, Kd) es esencial para garantizar un desempeño óptimo. Este controlador se implementó en Webots, un entorno de simulación robótica que permitió evaluar su comportamiento en un sistema dinámico bajo diferentes condiciones.

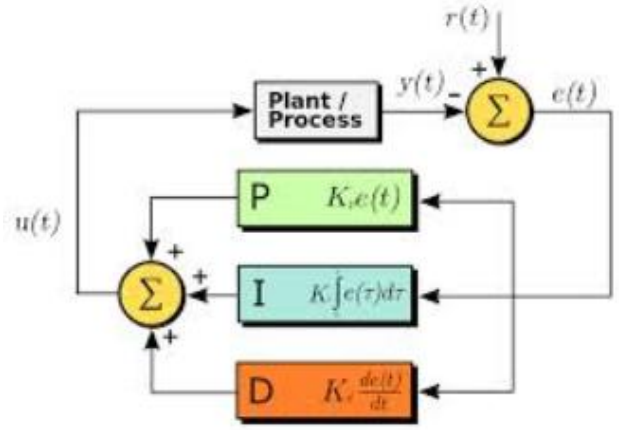

Figura 1. Diagrama de bloques básico PID

Control Difuso

El control difuso es una técnica de control que se basa en la lógica difusa, la cual permite trabajar con valores imprecisos o inciertos, imitando la manera en que los humanos toman decisiones. A diferencia del control PID, no requiere un modelo matemático exacto del sistema; en su lugar, se diseña un conjunto de reglas del tipo “Si–Entonces” que describen el comportamiento deseado.

Los elementos principales de un sistema de control difuso son:

Fuzzificación: convierte las entradas del sistema en valores difusos mediante funciones de pertenencia (por ejemplo: ángulo pequeño, grande, positivo, negativo).

Base de reglas: contiene las reglas lingüísticas que relacionan entradas con salidas (ejemplo: Si el ángulo es positivo y la velocidad es grande, entonces aplicar fuerza negativa).

Inferencia: combina las reglas activadas y determina la salida difusa.

Defuzzificación: transforma el valor difuso resultante en una acción concreta de control (por ejemplo, una fuerza o una señal de voltaje).

El control difuso es especialmente útil en sistemas no lineales, con incertidumbre o difíciles de modelar, ya que permite capturar el conocimiento experto en forma de reglas. En este proyecto, se implementó en Python, lo que facilitó la creación de funciones de pertenencia, reglas y la simulación del comportamiento del sistema bajo diferentes condiciones de operación.

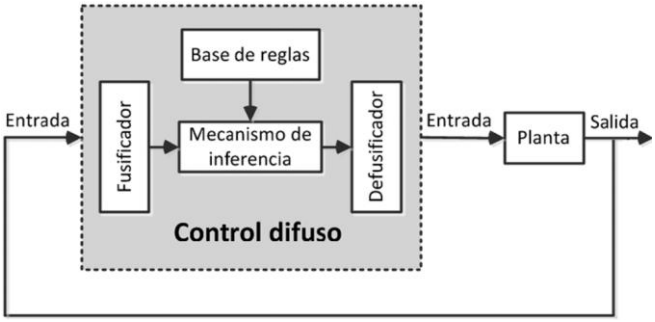

Figura 2. Diagrama de bloques Control difuso

III. ANALISIS

Modelo del péndulo invertido

El péndulo invertido es un sistema dinámico no lineal ampliamente utilizado como caso de estudio en control automático, debido a su inestabilidad natural. El modelo matemático parte de las leyes de Newton aplicadas tanto al carro como al péndulo.

Definimos las variables:

$x = posicion\ horizontal\ del\ carro\ (m)$

$\dot{x} = velocidad\ del\ carro\ (\frac{m}{s})$

$\theta = angulo\ del\ pendulo\ respecto\ a\ la\ vertical(rad)$

$\dot{\theta} = velocidad\ angular\ del\ pendulo\ \left(\frac{rad}{s}\right)$

$M = masa\ del\ carro(kg)$

$m = masa\ del\ pendulo(kg)$

$L = longitud\ del\ pendulo(m)$

$g = acaleracion\ de\ la\ gravedad\ \left(\frac{m}{s^2}\right)$

$F = Fuerza\ horizontal\ del\ carro(N)$

El modelo dinámico se obtiene considerando el equilibrio de fuerzas y momentos:

$$\ddot{\theta} = \frac{g\sin(\theta) - \cos(\theta)\left(\frac{F + mL\dot{\theta}^2\sin(\theta)}{M+m}\right)}{L\left(\frac{4}{3} - \frac{m\cos(\theta)}{M+m}\right)}$$

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

Estas ecuaciones describen la dinámica acoplada del sistema del péndulo invertido.

Para este punto se utiliza el modelo anteriormente descrito y se usa la función inverted_pendulum_model(), la cual calcula la evolución temporal del sistema usando método de integración de Euler.

Utilizando la librería pygame se valida el modelo físico del péndulo, luego se realiza un control manual aplicando fuerza externa mediante teclado, al presionar A se aplica una fuerza hacia la izquierda y al presionar D se aplica una fuerza hacia la derecha.

De esta forma, el usuario actúa como controlador, intentando mantener el péndulo en posición vertical estable. Esta etapa es fundamental porque permite observar la inestabilidad natural del péndulo y la dificultad de mantenerlo equilibrado sin un controlador automático, lo que justifica posteriormente la necesidad de implementar técnicas de control como el PID o el control difuso.

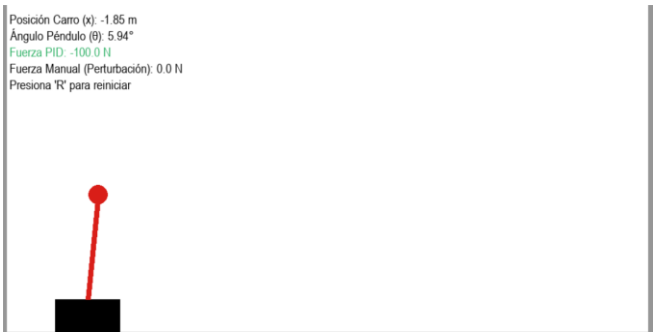

Figura 3. Simulación péndulo invertido con teclas

En la figura 3 se observa la implementación hecha en Python con datos de posición, angulo y fuerza aplicada.

Control PID mediante Webots y Python

Modelo físico Webots

El código del mundo webots que se encuentra en anexos se compone de un escenario compuesto por un riel, un carro y un péndulo unido al mismo mediante una articulación de tipo

HingeJoint. El carro se desplaza a lo largo del riel gracias a un motor lineal, mientras que el péndulo rota libremente en el plano vertical.

Se incorporan sensores de posición tanto para el carro como para el ángulo del péndulo, los cuales entregan la información necesaria al controlador. El sistema parte de una condición inicial con el péndulo ligeramente desviado de la vertical (0.02 rad), lo que garantiza que el equilibrio es inestable y se requiere acción de control desde el inicio.

Este modelo en Webots representa el comportamiento dinámico del péndulo invertido de manera realista, delegando la resolución de las ecuaciones de movimiento al motor físico del simulador.

Controlador PID en Python

El controlador se implementa en Python, conectado directamente a los sensores y al actuador del carro. Para mantener el equilibrio se utiliza un control PID, el cual combina acciones proporcional, integral y derivativa.

El error de control se define como una suma ponderada del ángulo del péndulo y la posición del carro, asignando mayor peso al ángulo (0.8) que a la posición (0.2). De esta forma, la prioridad del sistema es mantener el péndulo en la vertical, aunque también busca mantener el carro cercano al centro de la pista.

Las constantes del PID se ajustaron a valores kp=150, ki=0.1 kd=25, lo que permitió obtener una respuesta estable sin oscilaciones excesivas. Además, se incorporó una estrategia anti–windup para evitar la acumulación excesiva del término integral cuando el error es muy pequeño.

El sistema admite perturbaciones manuales a través del teclado. Presionando las teclas A o D se aplican fuerzas externas hacia la izquierda o derecha respectivamente, las cuales se suman a la acción del PID. Esto permite evaluar la robustez del controlador frente a perturbaciones externas.

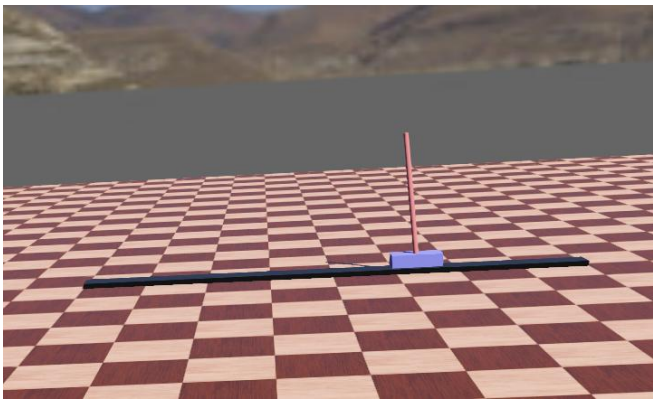

Figura 4. Mundo del péndulo invertido en software Webots

Control difuso

El código agregado en anexos implementa un control difuso para estabilizar un péndulo invertido montado sobre un carro. A diferencia del PID, este enfoque no se basa en una ecuación matemática explícita para la retroalimentación, sino en reglas heurísticas y funciones de pertenencia que modelan la lógica humana al enfrentar el problema del equilibrio.

Se diseñaron funciones de pertenencia trapezoidales (trapmf) para representar los estados lingüísticos de las variables de entrada y salida:

Ángulo del péndulo (θ): dividido en “muy negativo”, “negativo”, “positivo” y “muy positivo”, lo que permite identificar en qué rango se encuentra la inclinación del péndulo.

Velocidad angular (ω): representada como “negativa” o “positiva”, según el sentido del movimiento del péndulo.

Fuerza de control (F): clasificada en cinco niveles desde muy negativa hasta muy positiva, lo que define la magnitud y dirección de la fuerza aplicada sobre el carro.

Estas funciones permiten traducir valores numéricos en etiquetas lingüísticas, facilitando la aplicación de reglas de tipo si-entonces.

El controlador se rige por un conjunto de ocho reglas difusas que combinan el estado del ángulo y la velocidad angular para determinar la fuerza de salida. Por ejemplo:

Si el ángulo es positivo y la velocidad también es positiva, entonces aplicar una fuerza positiva media (xp).

Si el ángulo es negativo y la velocidad es negativa, entonces aplicar una fuerza negativa (xn).

De este modo, la lógica difusa intenta imitar la intuición humana: empujar el carro en la dirección adecuada dependiendo de hacia dónde se incline y se mueva el péndulo.

Las reglas se combinan mediante el operador mínimo (AND difuso) y luego se agregan con el máximo (OR difuso). Finalmente, se aplica el método de defuzzificación del centroide, que transforma la salida difusa en un valor de fuerza. Este valor corresponde a la acción de control que mueve el carro.

El modelo dinámico del péndulo invertido se resuelve mediante las ecuaciones no lineales del sistema, integradas con el método de Runge-Kutta de cuarto orden (RK4) para mayor precisión numérica.

La simulación gráfica se realiza con matplotlib.animation, representando el carro como un rectángulo azul y el péndulo como una barra roja. Además, se muestra en tiempo real el ángulo, la fuerza aplicada y el tiempo de simulación.

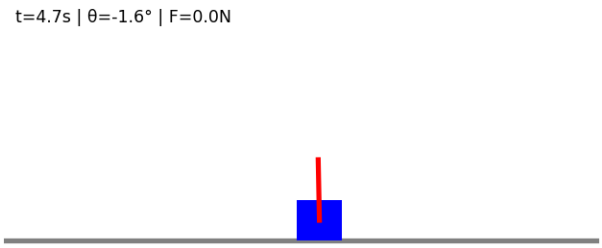

Figura 5. Simulación de controlador difuso

El controlador difuso consigue estabilizar el péndulo alrededor de la vertical, aplicando fuerzas variables en función del estado dinámico. A diferencia del PID, que requiere una sintonización precisa de parámetros Kp Ki y Kd, el control difuso depende de la definición de funciones de pertenencia y reglas heurísticas, lo que le otorga flexibilidad y robustez frente a la no linealidad del sistema.

IV. CONCLUSIONES

La implementación de los controladores PID y difuso para el problema del péndulo invertido permitió evidenciar dos enfoques distintos en el diseño de sistemas de control. El PID, aplicado en Webots, mostró ser efectivo para estabilizar el sistema siempre que sus parámetros estuvieran correctamente sintonizados, ofreciendo una respuesta rápida y estable frente a perturbaciones. Por su parte, el control difuso implementado en Python demostró ser una alternativa flexible y robusta, capaz de manejar la naturaleza no lineal del sistema mediante reglas lingüísticas y funciones de pertenencia, sin requerir un modelo matemático exacto. Si bien el PID destaca por su simplicidad y facilidad de implementación en sistemas bien modelados, el control difuso ofrece mayor adaptabilidad y un comportamiento más cercano a la lógica humana. En conjunto, ambos enfoques resaltan la importancia de seleccionar la estrategia de control adecuada en función de las características del sistema y los objetivos de desempeño, reafirmando que tanto las técnicas clásicas como las basadas en inteligencia computacional son herramientas complementarias en la ingeniería de control.

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, 3rd ed. Hoboken, NJ, USA: Wiley, 2010.

[3] J. S. R. Jang, C. T. Sun, and E. Mizutani, Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

ANEXOS

Codigo modelo:

```
import pygame
import math
import sys

# --------------------------------------------------------------------------
# PARTE 1: MODELO FÍSICO DEL PÉNDULO INVERTIDO (Sin cambios aquí)
# --------------------------------------------------------------------------

def inverted_pendulum_model(state, force, M, m, L, g, dt):
    """d
    Calcula el siguiente estado del péndulo invertido usando el método de Euler.
    """
    x, x_dot, theta, theta_dot = state
    # Asegurarse de que theta esté en el rango [-pi, pi] para un mejor comportamiento
    theta = (theta + math.pi) % (2 * math.pi) - math.pi

    sin_theta = math.sin(theta)
    cos_theta = math.cos(theta)

    # Ecuaciones del movimiento
    temp = (force + m * L * theta_dot**2 * sin_theta) / (M + m)

    theta_dot_dot_num = g * sin_theta - cos_theta * temp
    theta_dot_dot_den = L * (4/3 - (m * cos_theta**2) / (M + m))
    theta_dot_dot = theta_dot_dot_num / theta_dot_dot_den

    x_dot_dot = temp - (m * L * theta_dot_dot * cos_theta) / (M + m)

    # Integración de Euler
    x_dot += x_dot_dot * dt
    x += x_dot * dt
    theta_dot += theta_dot_dot * dt
    theta += theta_dot * dt

    return (x, x_dot, theta, theta_dot)

# --------------------------------------------------------------------------
# PARTE 2: CONTROLADOR PID
# --------------------------------------------------------------------------

class PIDController:
    """
    Clase para implementar un controlador PID.
    """
    def __init__(self, Kp, Ki, Kd, setpoint=0):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.setpoint = setpoint
        self._prev_error = 0
        self._integral = 0
```

```
    def update(self, current_value, dt):
        """
        Calcula la salida del PID (la fuerza) para un valor de entrada dado.
        """
        # Calcular error
        error = self.setpoint - current_value

        # Término Proporcional
        P_out = self.Kp * error

        # Término Integral
        self._integral += error * dt
        I_out = self.Ki * self._integral

        # Término Derivativo
        derivative = (error - self._prev_error) / dt
        D_out = self.Kd * derivative

        # Salida total del PID
        output = P_out + I_out + D_out

        # Guardar error para la siguiente iteración
        self._prev_error = error

        return output

# ------------------------------------------------------------------------
# PARTE 3: VISUALIZACIÓN Y CONTROL CON PYGAME
# ------------------------------------------------------------------------

def main():
    pygame.init()

    # --- Configuración de la simulación ---
    WIDTH, HEIGHT = 1000, 600
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Péndulo Invertido con PID | Perturbación: A/D | Reiniciar: R")
    font = pygame.font.SysFont("Arial", 20)

    # Colores
    WHITE = (255, 255, 255)
    BLACK = (0, 0, 0)
    RED = (217, 30, 24)
    GREEN_PID = (39, 174, 96)
    GRAY_WALL = (150, 150, 150)

    # --- Parámetros físicos configurables ---
    M = 100    # Masa del carro (kg) - Reducida para un control más ágil
    m = 0.1  # Masa del péndulo (kg)
    L = 0.8     # Longitud del péndulo (m)
    g = 9.81    # Gravedad (m/s^2)

    # --- Variables de estado iniciales ---
    x_init = 0.0
    x_dot_init = 0.0
    theta_init = 0.1  # Angulo inicial en radianes (ligeramente inclinado)
```

```
theta_dot_init = 0.0
state = (x_init, x_dot_init, theta_init, theta_dot_init)

# --- Parámetros de control y simulación ---
force_manual = 0
force_magnitude_manual = 250.0 # Fuerza para la perturbación manual
dt = 1/60

# --- Configuración del Controlador PID ---
# Estos valores pueden requerir ajuste para diferentes parámetros físicos
Kp = 1000  #150
Ki = 100   #0.5
Kd = 270  #30
pid = PIDController(Kp, Ki, Kd, setpoint=0)
max_force = 100 # Limitar la fuerza máxima que el PID puede aplicar

# --- Definición de los límites del mundo ---
scale = 200
cart_width_pixels = 100
world_boundary = (WIDTH - cart_width_pixels) / 2 / scale

# Bucle principal
running = True
clock = pygame.time.Clock()

while running:
    # Manejo de eventos
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
            if event.key == pygame.K_r: # Tecla de reinicio
                state = (x_init, x_dot_init, theta_init, theta_dot_init)
                pid._integral = 0 # Reiniciar el término integral del PID
                pid._prev_error = 0

    # Control manual (perturbación)
    keys = pygame.key.get_pressed()
    force_manual = 0
    if keys[pygame.K_a]:
        force_manual = -force_magnitude_manual
    elif keys[pygame.K_d]:
        force_manual = force_magnitude_manual

    # --- Lógica del Controlador PID ---
    # El estado del péndulo a controlar es theta (state[2])
    force_pid = pid.update(state[2], dt)

    # Limitar la fuerza del PID para evitar inestabilidad
    force_pid = max(-max_force, min(max_force, force_pid))

    # La fuerza total es la suma de la del PID y la perturbación manual
    total_force = force_pid + force_manual
```

```
        # Actualizar el estado del péndulo
        state = inverted_pendulum_model(state, total_force, M, m, L, g, dt)

        # Lógica para limitar el carro a la pantalla
        x, x_dot, theta, theta_dot = state
        if x > world_boundary:
            x = world_boundary
            x_dot = 0
        elif x < -world_boundary:
            x = -world_boundary
            x_dot = 0
        state = (x, x_dot, theta, theta_dot)

        # --- Dibujado en pantalla ---
        screen.fill(WHITE)

        floor_y = HEIGHT - 100
        pygame.draw.line(screen, BLACK, (0, floor_y), (WIDTH, floor_y), 2)
        pygame.draw.line(screen, GRAY_WALL, (0, 0), (0, floor_y), 10)
        pygame.draw.line(screen, GRAY_WALL, (WIDTH - 5, 0), (WIDTH - 5, floor_y), 10)

        cart_height = 50
        cart_x = state[0] * scale + WIDTH / 2 - cart_width_pixels / 2
        cart_y = floor_y - cart_height
        pygame.draw.rect(screen, BLACK, (cart_x, cart_y, cart_width_pixels, cart_height))

        pendulum_x_start = cart_x + cart_width_pixels / 2
        pendulum_y_start = cart_y
        pendulum_x_end = pendulum_x_start + L * scale * math.sin(state[2])
        pendulum_y_end = pendulum_y_start - L * scale * math.cos(state[2])
            pygame.draw.line(screen, RED, (pendulum_x_start, pendulum_y_start), (pendulum_x_end,
pendulum_y_end), 8)
        pygame.draw.circle(screen, RED, (int(pendulum_x_end), int(pendulum_y_end)), 15)

        # Mostrar información
        info_texts = [
            f"Posición Carro (x): {state[0]:.2f} m",
            f"Ángulo Péndulo (θ): {math.degrees(state[2]):.2f}°",
            f"Fuerza PID: {force_pid:.1f} N",
            f"Fuerza Manual (Perturbación): {force_manual:.1f} N",
            f"Presiona 'R' para reiniciar"
        ]
        for i, text in enumerate(info_texts):
            color = GREEN_PID if "PID" in text else BLACK
            text_surface = font.render(text, True, color)
            screen.blit(text_surface, (10, 10 + i * 25))

        pygame.display.flip()
        clock.tick(60)

    pygame.quit()
    sys.exit()

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

Codigo Control Python para webots

```
"""
Controlador PID para equilibrar un péndulo invertido en Webots R2025a.
Incluye perturbaciones manuales con las teclas A y D (versión corregida y mejorada).

Cambios clave:
- El PID se ejecuta en cada paso, no solo cuando no se presionan teclas.
- La fuerza de perturbación se SUMA a la fuerza del PID, en lugar de reemplazarla.
- Se ha corregido el signo de la fuerza aplicada para una reacción correcta.
- Se ha añadido un control secundario para mantener el carro cerca del centro.
- Se ha añadido un límite a la velocidad del motor para evitar fuerzas irreales.
"""

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 = 150  #150
# Ki corrige errores residuales a largo plazo. Se mantiene bajo.
Ki = 0.1  #0.1
# Kd amortigua la reacción y previene oscilaciones.
Kd = 25  #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 = 10 # 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)
```

Codigo Control difuso:

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

# ========================
# PARÁMETROS DEL SISTEMA
# ========================
g = 9.81
M = 0.5
m = 0.2
l = 0.3
dt = 0.02

# ========================
# FUNCIONES DIFUSAS
# ========================
def trapmf(x, a, b, c, d):
    return np.maximum(
        np.minimum(np.minimum((x-a)/(b-a+1e-6), 1), (d-x)/(d-c+1e-6)), 0
    )

def angulo_mf(theta):
    return {
        "amn": trapmf(theta, -180, -180, -60, -30),
        "an":  trapmf(theta, -60, -30, -5, 0),
        "ap":  trapmf(theta, 0, 5, 30, 60),
        "amp": trapmf(theta, 30, 60, 180, 180),
    }

def vel_mf(omega):
    return {
        "vn": trapmf(omega, -10, -10, -0.5, 0),
        "vp": trapmf(omega, 0, 0.5, 10, 10),
    }

x_vals = np.linspace(-100, 100, 200)
def fuerza_mf():
    return {
        "xmn": trapmf(x_vals, -100, -100, -80, -60),
        "xn":  trapmf(x_vals, -60, -40, -20, 0),
        "x0":  trapmf(x_vals, -10, 0, 0, 10),
        "xp":  trapmf(x_vals, 0, 20, 40, 60),
        "xmp": trapmf(x_vals, 60, 80, 100, 100),
    }

def fuzzy_force(theta_deg, omega):
    mu_theta = angulo_mf(theta_deg)
    mu_omega = vel_mf(omega)
    mu_F = fuerza_mf()

    rules = []
    rules.append(np.minimum(mu_theta["ap"], mu_omega["vp"]) * mu_F["xp"])
    rules.append(np.minimum(mu_theta["ap"], mu_omega["vn"]) * mu_F["x0"])
```

```
    rules.append(np.minimum(mu_theta["an"], mu_omega["vp"]) * mu_F["x0"])
    rules.append(np.minimum(mu_theta["an"], mu_omega["vn"]) * mu_F["xn"])
    rules.append(np.minimum(mu_theta["amp"], mu_omega["vp"]) * mu_F["xmp"])
    rules.append(np.minimum(mu_theta["amp"], mu_omega["vn"]) * mu_F["xp"])
    rules.append(np.minimum(mu_theta["amn"], mu_omega["vp"]) * mu_F["xn"])
    rules.append(np.minimum(mu_theta["amn"], mu_omega["vn"]) * mu_F["xmn"])

    aggregated = np.maximum.reduce(rules)
    if np.sum(aggregated) == 0:
        return 0.0
    return np.sum(aggregated * x_vals) / np.sum(aggregated)

# ========================
# DINÁMICA DEL PÉNDULO
# ========================
def cartpole_dynamics(state, F):
    x, xdot, th, thdot = state
    sin_th, cos_th = math.sin(th), math.cos(th)
    total_mass = M + m
    temp = (F + m * l * thdot**2 * sin_th) / total_mass
    denom = l * (4.0/3.0 - (m * cos_th**2) / total_mass)
    thddot = (g * sin_th - cos_th * temp) / denom
    xddot = temp - (m * l * thddot * cos_th) / total_mass
    return np.array([xdot, xddot, thdot, thddot])

def rk4_step(state, F, dt):
    k1 = cartpole_dynamics(state, F)
    k2 = cartpole_dynamics(state + 0.5 * dt * k1, F)
    k3 = cartpole_dynamics(state + 0.5 * dt * k2, F)
    k4 = cartpole_dynamics(state + dt * k3, F)
    return state + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

# ========================
# ANIMACIÓN
# ========================
state = np.array([0.0, 0.0, np.deg2rad(10), 0.0])  # estado inicial

fig, ax = plt.subplots()
ax.set_xlim(-2, 2)
ax.set_ylim(-1, 1.2)

# QUITAR EJES
ax.axis('off')

# PISO
ax.plot([-3, 3], [-0.1, -0.1], 'k-', lw=3, color="gray")

# Elementos gráficos
car_width, car_height = 0.3, 0.2
cart = plt.Rectangle((-car_width/2, -car_height/2), car_width, car_height, fc='blue')
ax.add_patch(cart)
line, = ax.plot([], [], lw=3, c='red')
time_text = ax.text(0.02, 0.9, '', transform=ax.transAxes, fontsize=10, color="black")

def init():
    cart.set_xy((-car_width/2, -car_height/2))
```

```
    line.set_data([], [])
    time_text.set_text('')
    return cart, line, time_text

def animate(i):
    global state
    theta_deg = np.rad2deg(state[2])
    omega = state[3]
    F = fuzzy_force(theta_deg, omega)

    state[:] = rk4_step(state, F, dt)

    x, _, th, _ = state
    cart.set_x(x - car_width/2)

    # Coordenadas del péndulo (vertical hacia arriba en 0 rad)
    px = x + l * np.sin(th)
    py = 0.0 + l * np.cos(th)
    line.set_data([x, px], [0, py])

    time_text.set_text(f't={i*dt:.1f}s | θ={theta_deg:.1f}° | F={F:.1f}N')
    return cart, line, time_text

ani = animation.FuncAnimation(fig, animate, frames=500, init_func=init,
                              interval=dt*1000, blit=True)
plt.show()
```