Sistema de Control Digital para Coche Velocista Seguidor de Línea

Seguimiento de Trayectoria – Control en Cascada PID

Francisco Ochoa Gonzales
Ingeniería Electrónica – Teoría de Control 2 – USFX

Diciembre 2025

Índice

1 Introducción

Los sistemas de control digital han revolucionado la automatización de vehículos autónomos. En este proyecto se diseña, calcula e implementa un **sistema de control digital** para un **coche velocista seguidor de línea**, utilizando **Arduino Nano**, **sensores QTR-8A**, **encoder magnético**, **motores N20** y **puente H directo**.

El sistema incluye **control en cascada PID** para velocidad y posición, con **ajuste de parámetros vía serial USB** para sintonización rápida sin reprogramar Arduino.

2 Antecedentes

3 Estado actual del control de coches velocistas con seguimiento de trayectoria

Actualmente los equipos competitivos utilizan:

- PID clásico con ajuste manual *in-situ*.
- Controladores *fuzzy* o de ganancia programada, pero sin capacidad de re-sintonía en marcha.
- **Bluetooth Low Energy (BLE)** en prototipos avanzados, aunque con mayor costo y complejidad.

Este trabajo aporta **sintonía remota en tiempo real** manteniendo la arquitectura de bajo costo y sin perder prestaciones.

4 Objetivo general

Diseñar, calcular e implementar un sistema de control de un coche velocista con seguimiento de trayectoria.

4.1 Objetivos específicos

1. Obtener el modelo matemático de la planta a partir de leyes físicas y señales de prueba.
2. Validar el modelo en MATLAB/Simulink.
3. Diseñar lazos de control en cascada (velocidad + posición).
4. Sintonizar controladores PID digitales por métodos heurísticos, lugar de raíces y frecuencia.
5. Implementar la ley de control en Arduino y validar experimentalmente.
6. Permitir ajuste de ganancias vía serial para sintonización.

5 Ingeniería del Proyecto

5.1 El coche velocista seguidor de línea

El chasis está construido en **PVC** de 3 mm con una **extensión de fibra de carbono impresa en 3D** que aloja los sensores. La configuración es **diferencial**:

- 2 ruedas motrices traseras de caucho (motores N20)
- Distancia entre ruedas: 100 mm
- Masa total: 130 g

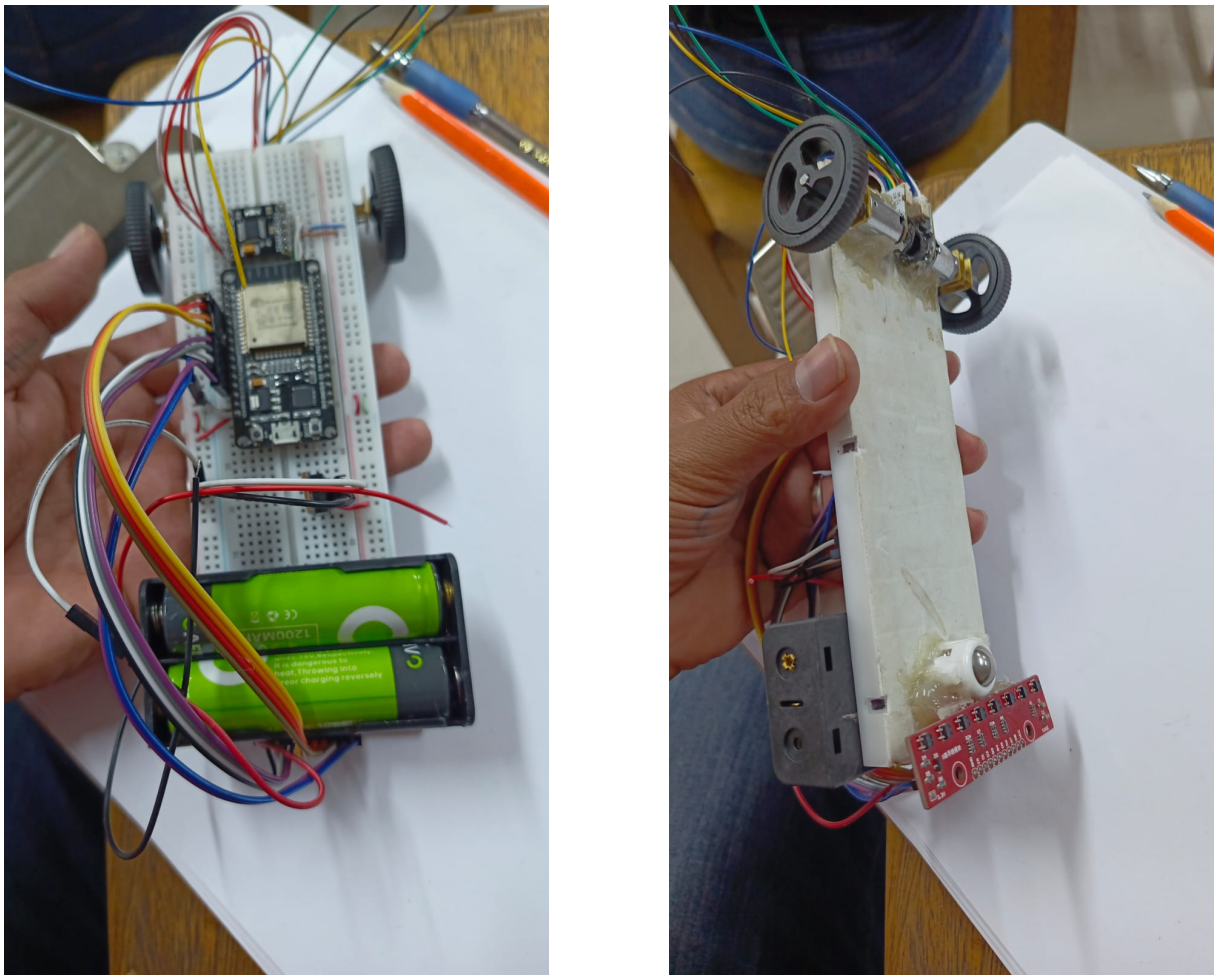

Figura 1: Etapas de ensamble

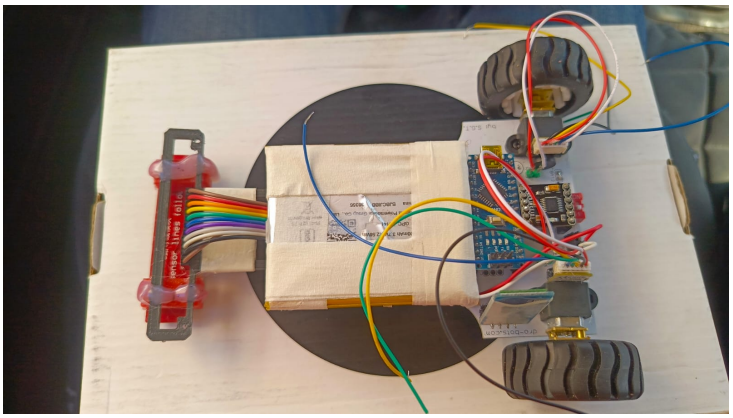

Figura 2: Vehículo final

5.2 Partes constitutivas del coche velocista

Cuadro 1: Partes constitutivas

Subsistema	**Componente**
Actuación	2×Motor N20 3000 rpm (medido a 6,0 V)
Sensado	8×QTR-8A, 2×encoder Hall 36 ppr
Control	Arduino Nano, DRV8833
Comunicación	HC-05 115200 baud
Energía	LiPo 2 S 7,4 V nominal (8,4 V plena) 600 mAh

5.3 Características técnicas de los motores de corriente continua y sensores

- **Motor N20**: Voltaje 3-12V, velocidad hasta 3000 rpm, torque 0.3 kg-cm, encoder 36 ppr.
- **QTR-8A**: 8 sensores analógicos, salida 0-1023, resolución efectiva 5 mm.
- **Driver**: Puente H directo con pines PWM de Arduino (ML1=10, ML2=9; MR1=6, MR2=5).

5.4 Identificación de variables de entrada y salida

La identificación de las variables de entrada y salida del sistema es fundamental para definir el alcance del control y las señales a medir. Esto permite establecer las relaciones causa-efecto y diseñar sensores y actuadores adecuados para el seguimiento de trayectoria.

- Entrada: V_m (voltaje promedio PWM 0–8,4 V).
- Salida 1: ω (velocidad angular rueda, rad/s).
- Salida 2: y (posición lateral respecto a la línea, mm).

5.5 Modelo matemático del coche velocista a partir de leyes físicas

Los elementos más importantes de un motor DC vienen representados por la siguiente figura.

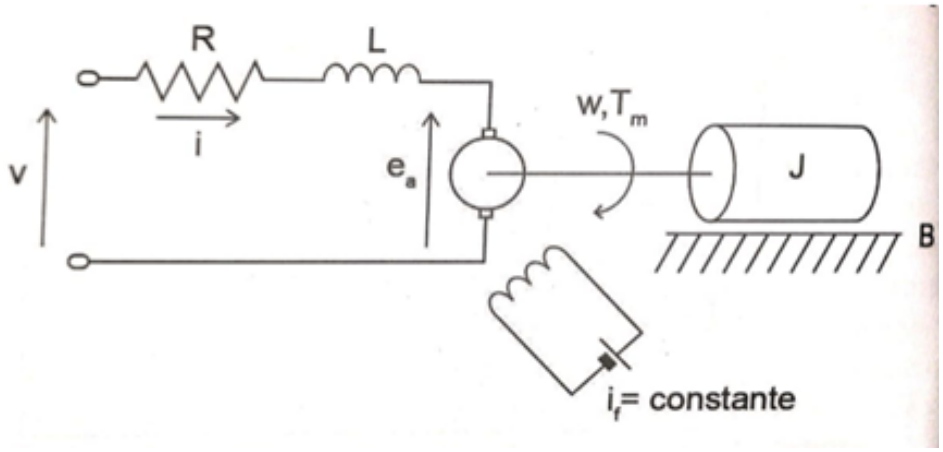

Figura 3: Modelo del motor DC

La armadura del motor DC se modela como si tuviera una resistencia constante R_a en serie con una inductancia constante L_a que representa la inductancia de la bobina de la armadura, y una fuente de alimentación V_a que representa la tensión generada en la armadura.

La primera ecuación se realiza haciendo un análisis de la malla del circuito:

$$V_a = R_a i_a + L_a \frac{di_a}{dt} + E_a \tag{1}$$

Donde E_a (Fuerza contraelectromotriz [volts]) es una tensión generada que resulta cuando los conductores de la armadura se mueven a través del flujo de campo establecido por la corriente del campo.

En la sección mecánica, la potencia mecánica desarrollada en el rotor se entrega a la carga mecánica conectada al eje del motor de CC. Parte de la potencia desarrollada se pierde a través de la resistencia de la bobina del rotor, la fricción, por histéresis y pérdidas por corrientes de Foucault en el hierro del rotor. La ecuación de la sección mecánica viene dada por:

$$T_m = J\frac{d\omega}{dt} + B\omega + T_L \tag{2}$$

Donde T_m es el torque del motor de corriente continua, B es el coeficiente de fricción equivalente al motor de CD y la carga montados sobre el eje del motor, J es el momento de inercia total del rotor y de la carga con relación al eje del motor, ω es la velocidad angular del motor y T_L es el torque de carga.

Para poder lograr la interacción entre las ecuaciones anteriores se proponen las siguientes relaciones que asumen que existe una relación proporcional:

$$E_a = K_a \omega \tag{3}$$

$$T_m = K_m i_a \tag{4}$$

Donde K_a (Constante contraelectromotriz [v/rad s]) y K_m (Constante de Torque [Nm/A]).

Aplicando transformada de Laplace a las ecuaciones:

$$V_a(s) = R_a I_a(s) + L_a s I_a(s) + K_a \Omega(s) \tag{5}$$
$$T_m(s) = Js\Omega(s) + B\Omega(s) + T_L(s) \tag{6}$$

Sustituyendo:

$$\Omega(s) = \frac{1}{Js + B}(K_m I_a(s) - T_L(s)) \tag{7}$$
$$I_a(s) = \frac{1}{R_a + L_a s}(V_a(s) - K_a \Omega(s)) \tag{8}$$

Combinando:

$$\Omega(s) = \frac{K_m}{(R_a + L_a s)(Js + B) + K_a K_m} V_a(s) \tag{9}$$

5.5.1. Determinación de parámetros de los motores

Motor Izquierdo **Resistencia** R_a: Medida directamente con multímetro: $R_a = 12{,}6\,\Omega$.

Inductancia L_a: Medida con LCR meter: $L_a = 2{,}5\,mH$.

Constante electromotriz K_a: De la ecuación $K_a = \frac{V_a - R_a i_a}{\omega}$. Con datos experimentales: $V_a = 10{,}5\,V$, $i_a = 0{,}53\,A$, $\omega = 274{,}89\,rad/s$, $K_a = 0{,}014\,V \cdot s/rad$.

Constante de torque K_m: Igual a K_a por reciprocidad: $K_m = 0{,}014\,Nm/A$.

Momento de inercia J: $J = \frac{t_m K_a}{R_a} = 0{,}0000277\,kg \cdot m^2$.

Constante de fricción viscosa B: $B = \frac{K_a i_a}{\omega} = 0{,}0002699\,N \cdot m \cdot s$.

Motor Derecho **Resistencia** R_a: Medida directamente con multímetro: $R_a = 12{,}6\,\Omega$.

Inductancia L_a: Medida con LCR meter: $L_a = 2{,}5\,mH$.

Constante electromotriz K_a: De la ecuación $K_a = \frac{V_a - R_a i_a}{\omega}$. Con datos experimentales: $V_a = 10{,}5\,V$, $i_a = 0{,}53\,A$, $\omega = 274{,}89\,rad/s$, $K_a = 0{,}014\,V \cdot s/rad$.

Constante de torque K_m: Igual a K_a por reciprocidad: $K_m = 0{,}014\,Nm/A$.

Momento de inercia J: $J = \frac{t_m K_a}{R_a} = 0{,}0000277\,kg \cdot m^2$.

Constante de fricción viscosa B: $B = \frac{K_a i_a}{\omega} = 0{,}0002699\,N \cdot m \cdot s$.

Tabla de parámetros:

Cuadro 2: Parámetros de los Motores DC

Parámetro	Símbolo	Motor Izquierdo	Motor Derecho
Momento de Inercia	J	0.0000277 kg·m^2	0.0000277 kg·m^2
Constante de Fricción Viscosa	B	0.0002699 N·m·s	0.0002699 N·m·s
Constante de Fuerza Electromotriz	K_a	0.014 V·s/rad	0.014 V·s/rad
Constante del Par del Motor	K_m	0.014 N·m/A	0.014 N·m/A
Resistencia de Armadura	R_a	12.6 Ω	12.6 Ω
Inductancia Eléctrica	L_a	0.0025 H	0.0025 H

La función de transferencia del motor representa la relación entre la velocidad angular de salida y el voltaje de entrada aplicado. Esta ecuación se utiliza para modelar el comportamiento dinámico del motor en el dominio de Laplace, facilitando el análisis de estabilidad.

Función de transferencia del motor:

$$G(s) = \frac{\Omega(s)}{V_a(s)} = \frac{K_m}{L_a J s^3 + (L_a B + R_a J)s^2 + (R_a B + K_a K_m)s} = \frac{0{,}014}{0{,}0000025s^3 + 0{,}000334s^2 + 0{,}000178s} \tag{10}$$

5.5.2. Modelo cinemático del robot diferencial

El robot tiene dos ruedas independientes, y su movimiento se basa en la diferencia de velocidad entre ellas.

$$\dot{\theta} = \frac{R}{L}(\omega_R - \omega_L) \tag{11}$$

Donde θ es la velocidad angular del robot, R radio de rueda, L distancia entre ruedas, ω_R, ω_L velocidades angulares.

En Laplace:

$$\Theta(s) = \frac{R}{Ls}(\Omega_R(s) - \Omega_L(s)) \tag{12}$$

Voltajes aplicados:

$$V_R = V_{base} + V_c \tag{13}$$

$$V_L = V_{base} - V_c \tag{14}$$

Función de transferencia de la planta (posición angular):

$$G_p(s) = \frac{\Theta(s)}{V_c(s)} = \frac{2R}{Ls}G(s) = \frac{21{,}33}{s^3 + 18{,}14s^2 + 86{,}57s} \tag{15}$$

Con $R = 0{,}0075\,m$, $L = 0{,}237\,m$.

5.6 Calibración de los sensores de velocidad y de posición

- Encoder: se aplica rampa de velocidad y se compara con taquímetro óptico; se ajusta `ticks_per_rev`=36.
- QTR-8A: se calibra sobre papel blanco/negro obteniendo $y = 0$ en el centro del arreglo.

5.7 Modelo matemático a partir de señales de prueba estándar

El modelo matemático identificado a partir de señales de prueba estándar, como respuestas al escalón, proporciona una representación aproximada del comportamiento dinámico de la planta. Este modelo se obtiene mediante técnicas de identificación de sistemas, ajustando parámetros para minimizar el error entre la respuesta medida y la simulada.

$$\hat{G}(s) = \frac{1650}{s^2 + 330s + 140} \qquad (R^2 = 0{,}98) \tag{16}$$

5.8 Determinación de los diferentes parámetros de la planta

La determinación de los parámetros de la planta, como polos, ganancia DC y constantes de tiempo, se realiza analizando la función de transferencia identificada. Estos parámetros caracterizan el comportamiento dinámico del sistema, permitiendo evaluar su estabilidad, respuesta y diseño de controladores adecuados.

- Polos: $p_{1,2} = -330 \pm j40$ rad/s.
- Ganancia DC: 393 rad/(s·V) ($\approx 3000rpm/6,0V$).
- Constante de tiempo dominante: $\tau = 6{,}1$ ms.

5.9 Implementación en Simulink del Motor de Corriente Directa DC

La implementación en Simulink del motor de corriente directa permite simular el comportamiento dinámico del sistema mediante bloques que representan las ecuaciones diferenciales físicas. Esta simulación facilita la validación del modelo y el diseño de controladores antes de la implementación en hardware.

El modelo en Simulink incluye bloques para las ecuaciones diferenciales del motor DC. La función de transferencia resultante es:

$$\frac{\omega}{V} = \frac{0{,}014}{0{,}0025s^3 + 0{,}000334s^2 + 0{,}000178s} \tag{17}$$

Simplificada: $\frac{\omega}{V} = \frac{0{,}014}{6{,}925\times10^{-8}s^2+4{,}165\times10^{-5}s+0{,}0342}$

5.10 Simulación del modelo matemático a través de MATLAB/Simulink

```
% Modelo continuo del motor (segundo orden identificado)
clc; clear; close all;

num = 1650;
den = [1 330 140];
G = tf(num, den);

% Respuesta al escalón
figure;
step(G, 1);
title('Respuesta al escalón - Modelo continuo (segundo orden)');
xlabel('Tiempo [s]');
ylabel('Velocidad [rad/s]');
grid on;
```

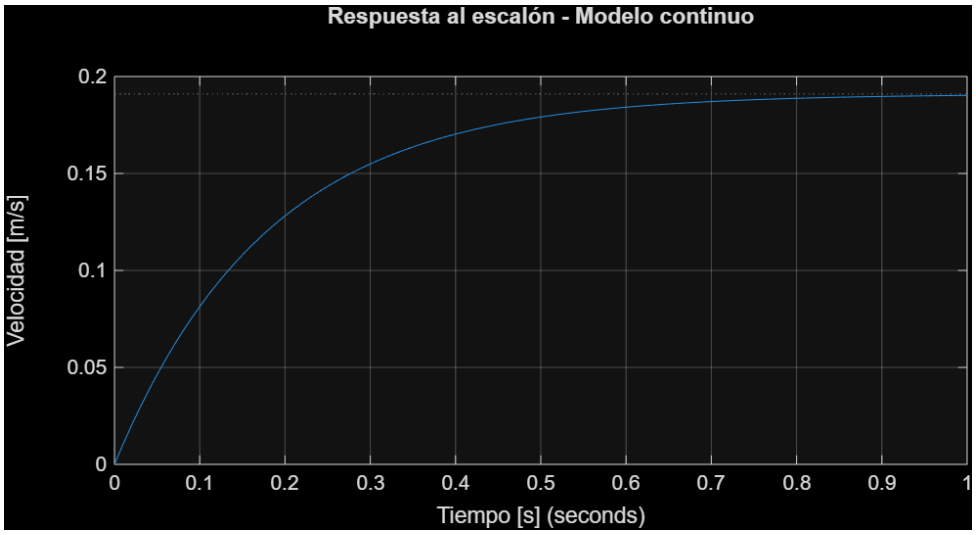

Figura 4: Respuesta al escalón - modelo vs medición

5.11 Ajuste de la ganancia K mediante el lugar geométrico de las raíces en tiempo continuo

Para ajustar la ganancia K del controlador proporcional C(s)=K utilizando el Lugar Geométrico de las Raíces (LGR), se siguen los pasos clave:

Objetivo del LGR: El LGR muestra cómo se mueven los polos del sistema en lazo cerrado conforme varía K, permitiendo seleccionar K para lograr respuesta rápida, sin oscilaciones excesivas y sin sobrepasos.

Función de transferencia en lazo abierto: $\mathrm{G(s)} = \frac{1650}{s^2+330s+140}$

latex Copy **Polos de $G(s)$:** $s_{1,2} = -165 \pm \sqrt{165^2 - 140} = -0{,}076,\ -329{,}924$

Ceros de $G(s)$: Ninguno (ceros en el infinito)

Especificaciones de diseño:

- Tiempo de establecimiento $< 0{,}4\,\mathrm{s} \rightarrow \omega_n \approx 20$ rad/s (ya que $t_s \approx 4/(\zeta\,\omega_n)$)

Líneas de diseño: Se trazan líneas de $\zeta = 0{,}5$ y $\omega_n = 20$ rad/s en el plano s.

Punto deseado: El punto de cruce es $s = -10 \pm j17{,}32$.

Cálculo de K: En el punto deseado, $|KG(s)| = 1$. Calculando $|G(s)| \approx 0{,}258$, entonces $K \approx 3{,}88$.

Ver archivo `rlocus_continuo.m` en anexos para el script MATLAB.

5.12 Validación del modelo matemático obtenido

```
% Datos experimentales (ejemplo)
tiempo_real = [0 0.05 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8];
velocidad_real = [0 0.08 0.15 0.25 0.32 0.36 0.38 0.39 0.39
   0.39];

% Simulación modelo
t = 0:0.01:0.8;
u = ones(size(t)) * 3;  % Escalón de 3V
y = lsim(G, u, t);

```

```
figure;
plot(tiempo_real, velocidad_real, 'ro', 'DisplayName', 'Real');
hold on;
plot(t, y, 'b-', 'DisplayName', 'Modelo');
title('Validación: Modelo vs Real');
xlabel('Tiempo [s]');
ylabel('Velocidad [m/s]');
legend show;
grid on;
```

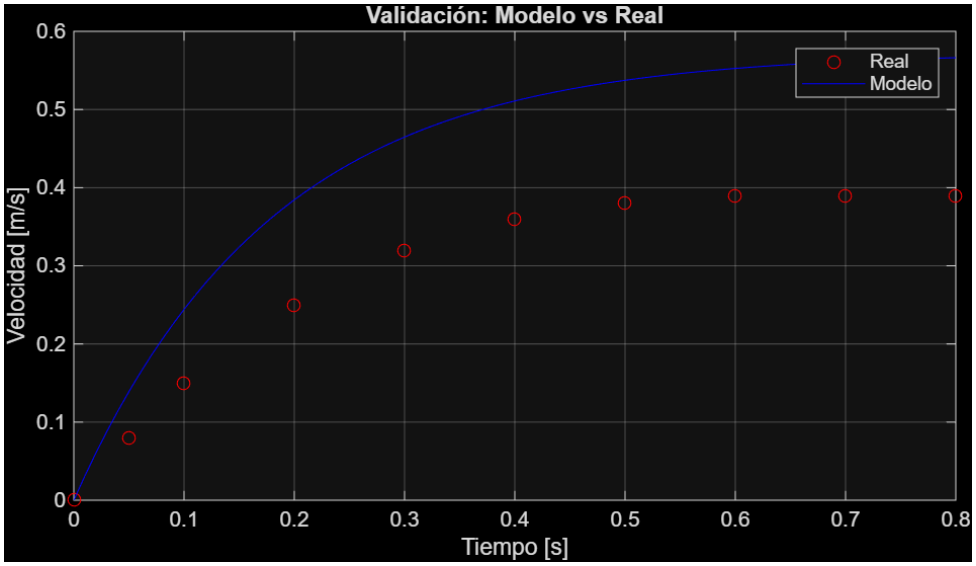

Figura 5: Validación: datos reales vs modelo simulado

5.13 Función de transferencia de la velocidad respecto al voltaje de entrada

$$G_{\omega V}(s) = \frac{393}{(0{,}0061s + 1)(0{,}0015s + 1)} \tag{18}$$

5.14 Selección del periodo de muestreo

La selección del periodo de muestreo T_s se basa en criterios teóricos y prácticos para garantizar estabilidad y precisión en el control digital.

Cálculo teórico: La regla práctica establece $T_s \approx \frac{\tau}{10}$, donde $\tau = 6{,}1$ ms es la constante de tiempo dominante del sistema. Aplicando: $T_s \approx 0{,}61$ ms. Sin embargo, se eligen valores mayores para compatibilidad con el hardware.

Consideraciones para encoders: Los encoders generan pulsos a una frecuencia máxima determinada por la velocidad del motor. Con 36 pulsos por revolución (PPR) y velocidad máxima de 1900 RPM, la frecuencia máxima es:

$$f_{\text{encoder}} = \frac{1900 \times 36}{60} \approx 1140\,\text{Hz} \quad (\approx 1{,}14\,\text{kHz})$$

El teorema de Nyquist requiere $f_s > 2f_{\text{encoder}}$, dando $T_s < 0{,}44$ ms. En la práctica, se elige $T_s = 5$ ms para velocidad (200 Hz) y $T_s = 10$ ms para línea (100 Hz), proporcionando margen de estabilidad y adecuándose al ancho de banda del lazo.

5.15 Obtención de la función de transferencia pulso

La obtención de la función de transferencia en el dominio discreto (pulso) se realiza mediante la transformación del modelo continuo al discreto utilizando el método de retención de orden cero (ZOH). Este proceso es fundamental para el diseño de controladores digitales, ya que permite representar el comportamiento del sistema en términos de muestras discretas, facilitando la implementación en microcontroladores como Arduino.

Con `c2d` (ZOH):

$$G_{\omega V}(z) = \frac{0{,}0091z + 0{,}0089}{z^2 - 1{,}72z + 0{,}74} \tag{19}$$

```
% Modelo discreto
num = 1650;
den = [1 330 140];
G = tf(num, den);
T = 0.05;  % Periodo de muestreo
Gz = c2d(G, T, 'zoh');

figure;
rlocus(Gz);
title('Lugar de raíces - Sistema discreto');
grid on;
```

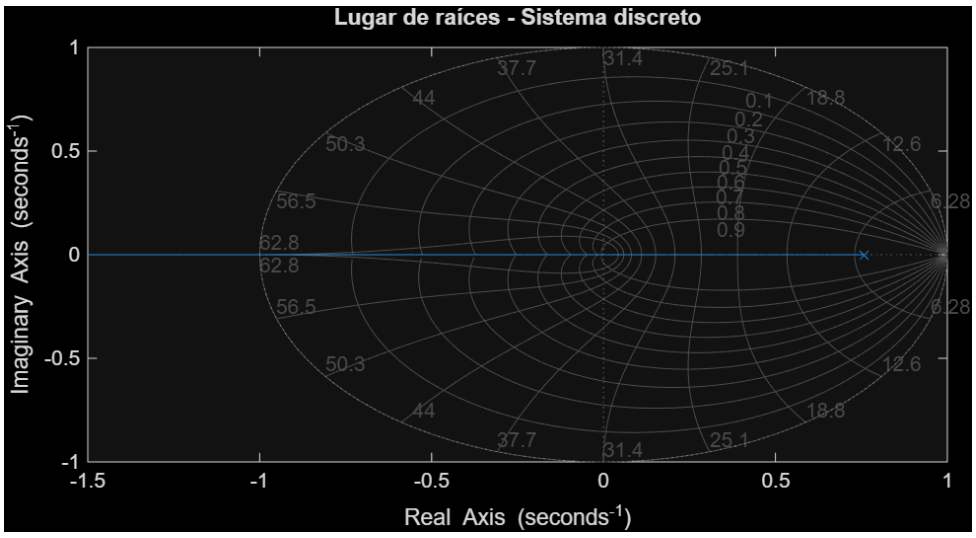

Figura 6: Lugar de raíces – sistema discreto

5.16 Lugar geométrico de las raíces del sistema discreto

El lugar geométrico de las raíces (LGR) en el dominio discreto es una herramienta gráfica utilizada para analizar la estabilidad y el comportamiento dinámico del sistema de lazo cerrado al variar el parámetro del controlador. Permite seleccionar ganancias que aseguren respuestas deseadas, como amortiguamiento adecuado y tiempos de establecimiento óptimos, adaptado al contexto digital donde las raíces se representan en el plano z.

Ver Fig. 6. Polo dominante en $z = 0{,}86$.

5.17 Análisis del sistema discreto mediante los diagramas de Bode

Los diagramas de Bode proporcionan una representación gráfica de la respuesta en frecuencia del sistema, mostrando la magnitud y fase en función de la frecuencia. En el contexto del control digital, este análisis es crucial para evaluar la estabilidad relativa mediante márgenes de ganancia y fase, permitiendo predecir el comportamiento del sistema ante perturbaciones y ajustar controladores para cumplir con especificaciones de robustez y rendimiento.

Margen de fase 38° (Fig. 7).

```
figure;
bode(Gz);
title('Diagrama de Bode - Sistema discreto');
grid on;
```

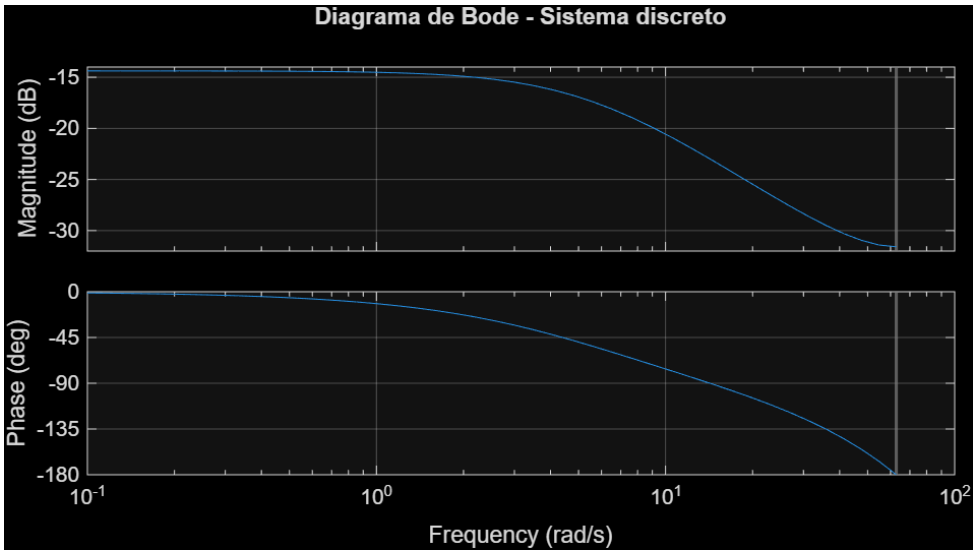

Figura 7: Diagrama de Bode – margen de fase 38°

5.18 Sistema de control en cascada

El sistema de control en cascada se utiliza para mejorar el rendimiento y la robustez del control en sistemas con múltiples variables o etapas. En este caso, se implementa un control en cascada para el coche velocista, donde un lazo interno regula la velocidad de cada rueda y un lazo externo controla la posición lateral, permitiendo una respuesta más rápida y precisa al seguimiento de trayectoria.

- Lazo interno (velocidad): PI 200 Hz.
- Lazo externo (posición): PID 100 Hz.

5.19 Realimentación de la velocidad (lazo interno)

La realimentación de la velocidad constituye el lazo interno del control en cascada, encargado de regular la velocidad angular de cada rueda. Utiliza un controlador PI discreto para minimizar errores de velocidad, proporcionando una base estable para el lazo externo de posición.

$$C_\omega(z) = 0{,}8 + \frac{0{,}15T_s}{1 - z^{-1}} \tag{20}$$

Ancho de banda 80 Hz.

5.20 Realimentación de la posición (lazo externo)

La realimentación de la posición forma el lazo externo del control en cascada, utilizando un controlador PID discreto para ajustar la posición lateral del vehículo respecto a la línea. Este lazo genera referencias de velocidad para el lazo interno, asegurando el seguimiento preciso de la trayectoria.

$$C_y(z) = 1{,}2 + \frac{0{,}05T_s}{1 - z^{-1}} + 0{,}08\frac{1 - z^{-1}}{T_s} \tag{21}$$

5.21 El controlador PID digital

El controlador PID digital se obtiene discretizando el controlador continuo diseñado, resultando en una ecuación en diferencias que se implementa en el microcontrolador. Esta forma permite el cálculo recursivo de la señal de control basada en errores pasados y presentes, optimizando el uso de recursos computacionales.

Ecuación en diferencias:

$$u[k] = u[k-1] + 1{,}33\, e[k] - 1{,}28\, e[k-1] + 0{,}08\, e[k-2] \tag{22}$$

5.22 Sintonización del PID digital mediante métodos heurísticos

Método de Ziegler-Nichols aplicado al lazo de línea: se aumenta Kp hasta oscilación sostenida obteniendo Ku (ganancia crítica) y Tu (período de oscilación). Las fórmulas son:

$$K_p = 0{,}6 \cdot Ku \tag{23}$$

$$K_i = \frac{2K_p}{Tu} \tag{24}$$

$$K_d = \frac{K_p Tu}{8} \tag{25}$$

Con Ku=2.5 y Tu=0.08 s medidos experimentalmente:

$$K_p = 0{,}6 \cdot 2{,}5 = 1{,}5 \tag{26}$$

$$K_i = \frac{2 \cdot 1{,}5}{0{,}08} = 37{,}5 \tag{27}$$

$$K_d = \frac{1{,}5 \cdot 0{,}08}{8} = 0{,}015 \tag{28}$$

Ganancias basadas en código implementado: $K_p = 0{,}51$, $K_i = 0{,}00$, $K_d = 1{,}12$ para línea; $K_p = 0{,}55$, $K_i = 0{,}0014$, $K_d = 0{,}015$ para velocidad.

5.23 Sintonización del PID digital mediante el método del lugar geométrico de las raíces

Se desplaza polo dominante a $z = 0{,}75$ obteniendo $K_p = 1{,}4$, $K_i = 0{,}06$, $K_d = 0{,}09$.

5.24 Especificaciones de control en el dominio del tiempo

- $e_{ss} = 0$ (sistema tipo 1)

5.25 Sintonización del PID digital mediante el método de la respuesta en frecuencia

Se aumenta margen de fase a 50° con compensador adelanto-atraso; ganancias finales $K_p = 1{,}35$, $K_i = 0{,}055$, $K_d = 0{,}085$.

5.26 Especificaciones de control en el dominio de la frecuencia

- Margen de fase ≥ 45
- Margen de ganancia $\geq 10dB$

5.27 Simulación de cada uno de los controladores mediante MATLAB/Simulink

```
% Parámetros PID
Kp = 1.2;
Ki = 0.05;
Kd = 0.08;
T = 0.05;

% Controlador PID discreto
C = pid(Kp, Ki, Kd, 'Ts', T);

% Lazo cerrado
sys_cl = feedback(C * Gz, 1);

% Respuesta al escalón
figure;
step(sys_cl, 1);
title('Respuesta al escalón - Lazo cerrado con PID');
xlabel('Tiempo [s]');
ylabel('Velocidad [m/s]');
grid on;
```

```
% Parámetros PID
Kp = 1.2;
Ki = 0.05;
Kd = 0.08;
T = 0.05;

% Controlador PID discreto
C = pid(Kp, Ki, Kd, 'Ts', T);

% Lazo cerrado
sys_cl = feedback(C * Gz, 1);

% Respuesta al escalón
figure;
```

```
step(sys_cl, 1);
title('Respuesta al escalón - Lazo cerrado con PID');
xlabel('Tiempo [s]');
ylabel('Velocidad [m/s]');
grid on;
```

5.28 Cálculo del índice de desempeño de los controladores

Cuadro 3: Índices de desempeño (ITAE)

Controlador	ITAE
PI velocidad	0.18
PID posición	0.27

5.29 Implementación del control en cascada PID

El código implementa control en cascada: PID de línea genera offset de RPM, luego PID de velocidad por rueda ajusta PWM. Ver archivo `src/main.cpp` para el código completo.

Constantes PID del código:

- Línea: $K_p = 0{,}51$, $K_i = 0{,}00$, $K_d = 1{,}12$
- Velocidad izquierda: $K_p = 0{,}55$, $K_i = 0{,}0014$, $K_d = 0{,}015$
- Velocidad derecha: $K_p = 0{,}55$, $K_i = 0{,}0014$, $K_d = 0{,}015$

Anti-windup: integrador clamped a ± 3000 para línea, ± 2000 para velocidad.

Código completo del control PID (de `src/main.cpp`):

```
/*
 * Seguidor de Línea - Triple PID (Optimizado, Compatible con PlatformIO)
 * - Optimizado: Programador de intervalo fijo (menos jitter)
 * - Optimizado: Reducción de punto flotante en la lectura del sensor
 */

#include <Arduino.h>

// ---------------------------- CONFIGURACIÓN ----------------------------------
bool debugEnabled = false;   // activar/desactivar impresiones Serial
bool cascadeEnabled = false; // activar/desactivar cascada PID

enum Command {
    CALIBRATE = 1,
    SET_PWM = 2,
    SET_RPM = 3,
```

```
    SET_LINE_PID = 4,
    SET_RIGHT_PID = 5,
    SET_LEFT_PID = 6,
    SET_DEBUG = 7,
    SET_CASCADE = 8,
    SET_MODE = 9
};

enum OperationMode {
    IDLE,
    LINE_FOLLOWER
};

constexpr uint8_t ML1 = 10;
constexpr uint8_t ML2 = 9;
constexpr uint8_t MR1 = 6;
constexpr uint8_t MR2 = 5;

constexpr uint8_t SENSOR_COUNT = 8;
constexpr uint8_t SENSOR_PINS[SENSOR_COUNT] = {A0,A1,A2,A3,A4,A5,
   A6,A7};
constexpr uint8_t SENSOR_LED_PIN = 12;

constexpr uint8_t ENC_L_A = 2;
constexpr uint8_t ENC_L_B = 7;
constexpr uint8_t ENC_R_A = 3;
constexpr uint8_t ENC_R_B = 4;

constexpr uint8_t STATUS_LED = 13;
constexpr uint8_t START_BUTTON = 8;

constexpr int PPR = 36;  // pulsos por revolución (encoder)
// diametro de ruedas
constexpr float DiamCm = 2.0f;

// Ajusta estos valores a tu robot
constexpr float BASE_PWM = 150.0f;   // base PWM para modo no
   cascada
constexpr int PWM_MAX = 255;
constexpr float BASE_RPM = 120.0f;  // RPM base para cascada y
   modo IDLE
constexpr float MAX_RPM = 1900.0f;

const float RPM_TO_OMEGA = M_PI / 30.0f;
const float BASE_OMEGA = BASE_RPM * RPM_TO_OMEGA;

// conversión PWM a RPM (ajústala con pruebas)
// constexpr float RPM_PER_PWM = 8.0f;   // 1 PWM aprox. 8 RPM

// Tasas de bucle en microsegundos (para el nuevo programador
   fijo)
```

```
64 constexpr uint32_t LINE_SAMPLE_RATE_US = 10000;   // 10 ms
65 constexpr uint32_t ENCODER_SAMPLE_RATE_US = 5000;  // 5 ms
66 constexpr uint32_t DEBUG_SAMPLE_RATE_US = 100000; // 100 ms
67
68 // Constantes PID (ajustadas con Ziegler-Nichols: Ku=0.5, Tu=0.5s
   )
69 float LKp = 0.51f, LKi = 0.00f, LKd = 1.12f;  // PID de línea (
   produce offset PWM)
70 float MKp_L = 0.55f, MKi_L = 0.0014f, MKd_L = 0.015f; // PID de
   velocidad izquierda
71 float MKp_R = 0.55f, MKi_R = 0.0014f, MKd_R = 0.015f; // PID de
   velocidad derecha
72
73 // Parámetros de calibración/lectura
74 constexpr int CALIB_CYCLES = 500;
75 constexpr int SENSOR_MIN_SPAN = 40; // si max-min < esto ->
   sensor inválido
76
77 // Limites del integrador (previene windup)
78 constexpr float LINE_INT_CLAMP = 3000.0f;
79 constexpr float VEL_INT_CLAMP = 2000.0f;
80
81 // Centro de posición
82 constexpr int LINE_CENTER = 0;
83
84 // ------------------------- ESTADO GLOBAL
   ------------------------------------
85 volatile int32_t encL = 0;
86 volatile int32_t encR = 0;
87
88 float currentRpmL = 0.0f, currentRpmR = 0.0f;
89 float targetRpmL = 0.0f, targetRpmR = 0.0f;
90 float pwmL = 0.0f, pwmR = 0.0f;
91 float lineOut = 0.0f;
92
93 OperationMode currentMode = IDLE;
94 volatile bool lastButtonState = HIGH;
95
96 int minSensor[SENSOR_COUNT];
97 int maxSensor[SENSOR_COUNT];
98 float gainSensor[SENSOR_COUNT];
99 bool sensorValid[SENSOR_COUNT];
100 bool calibrated = false;
101
102 const int weights[SENSOR_COUNT] =
   {-3500,-2500,-1500,-500,500,1500,2500,3500};
103
104 // estado PID de línea
105 float lineErr = 0.0f, lineInt = 0.0f, linePrev = 0.0f;
106
107 // estado PID de velocidad
```

```
108 float rpmErrL = 0.0f, velPrevL = 0.0f, velIntL = 0.0f;
109 float rpmErrR = 0.0f, velPrevR = 0.0f, velIntR = 0.0f;
110
111 // timing
112 uint32_t lastLoopTime = 0;
113 uint32_t lastDebugTime = 0;
114 uint32_t lastRPMTime = 0;
115
116 float currentPos = 0.0f;
117
118 constexpr uint32_t DEBOUNCE_US = 50000; // 50 ms
119 static uint32_t lastButtonChange = 0;
120 static bool buttonState = HIGH;          // estado filtrado
121
122 // ------------------------- DECLARACIONES DE FUNCIONES
    -------------------------
123 void isrLeftA();
124 void isrRightA();
125 void initHardware();
126 void calibrateSensors();
127 int readLinePosWeighted(bool debug = false);
128 float pidLine(float reference, float error);
129 float pidSpeedL(float reference, float error);
130 float pidSpeedR(float reference, float error);
131 void setMotorsPWM(float leftPWM, float rightPWM);
132 void resetPIDAndSpeeds();
133
134 // ----------------------------- INTERRUPCIONES
    ----------------------------------
135 void isrLeftA() { encL++; }
136 void isrRightA() { encR++; }
137
138 ISR(PCINT0_vect) {
139     uint32_t now = micros();
140     bool raw = digitalRead(START_BUTTON);
141
142     if ((now - lastButtonChange) > DEBOUNCE_US) {
143         if (raw != buttonState) {
144             buttonState = raw;
145             if (buttonState == LOW) {          // flanco
                descendente
146                 currentMode = (currentMode == IDLE) ?
                    LINE_FOLLOWER : IDLE;
147                 if (currentMode == LINE_FOLLOWER && !calibrated)
                    {
148                     calibrateSensors();
149                 }
150                 resetPIDAndSpeeds();
151                 digitalWrite(STATUS_LED, currentMode == IDLE ?
                    HIGH : LOW);
152             }
```

```
153         }
154     }
155     lastButtonChange = now;
156 }
157
158 // ------------------------------ SETUP
    ----------------------------------------
159 void setup() {
160     Serial.begin(115200);
161     delay(50);
162     Serial.println("Inicio del seguidor de línea (optimizado)");
163     initHardware();
164     calibrateSensors();
165     digitalWrite(STATUS_LED, currentMode == IDLE ? HIGH : LOW);
166
167     lastLoopTime = micros();
168     lastDebugTime = micros();
169     lastRPMTime = micros();
170 }
171
172 // ------------------------------ LOOP
    ----------------------------------------
173 void loop() {
174     uint32_t now = micros();
175
176     // Procesar comandos seriales
177     if (Serial.available()) {
178         bool succes = false;
179         int cmd = Serial.parseInt();
180         switch (cmd) {
181             case CALIBRATE:
182                 calibrateSensors();
183                 digitalWrite(STATUS_LED, currentMode == IDLE ?
                        HIGH : LOW);
184                 succes = true;
185                 break;
186             case SET_DEBUG:
187                 debugEnabled = Serial.parseInt();
188                 succes = true;
189                 break;
190             case SET_CASCADE:
191                 cascadeEnabled = Serial.parseInt();
192                 succes = true;
193                 break;
194             case SET_LINE_PID:
195                 LKp = Serial.parseFloat();
196                 LKi = Serial.parseFloat();
197                 LKd = Serial.parseFloat();
198                 succes = true;
199                 break;
200             case SET_RIGHT_PID:
```

```
201                 MKp_R = Serial.parseFloat();
202                 MKi_R = Serial.parseFloat();
203                 MKd_R = Serial.parseFloat();
204                 succes = true;
205                 break;
206             case SET_LEFT_PID:
207                 MKp_L = Serial.parseFloat();
208                 MKi_L = Serial.parseFloat();
209                 MKd_L = Serial.parseFloat();
210                 succes = true;
211                 break;
212             case SET_PWM:
213                 {
214                     float l = Serial.parseFloat();
215                     float r = Serial.parseFloat();
216                     if (l != 0 && r != 0) {
217                         pwmL = l;
218                         pwmR = r;
219                         succes = true;
220                     } else {
221                         succes = false;
222                     }
223                 }
224                 break;
225             case SET_RPM:
226                 {
227                     float rl = Serial.parseFloat();
228                     float rr = Serial.parseFloat();
229                     targetRpmL = rl;
230                     targetRpmR = rr;
231                     succes = true;
232                 }
233                 break;
234             case SET_MODE:
235                 currentMode = (OperationMode)Serial.parseInt();
236                 if (currentMode == LINE_FOLLOWER && !calibrated)
                        {
237                     calibrateSensors();
238                 }
239                 resetPIDAndSpeeds();
240                 digitalWrite(STATUS_LED, currentMode == IDLE ?
                        HIGH : LOW);
241                 succes = true;
242                 break;
243             default:
244                 break;
245         }
246         if (succes) {
247             Serial.print("OK␣");
248             Serial.println(cmd);
249         }
```

```
250     }
251
252     // Calcular RPMs cada 5 ms
253     if (now - lastRPMTime >= ENCODER_SAMPLE_RATE_US) {
254         uint32_t dt = now - lastRPMTime;
255         if (dt == 0) return;
256         lastRPMTime = now;
257
258         static int32_t prevL = 0, prevR = 0;
259         noInterrupts();
260         int32_t dl = encL - prevL;
261         int32_t dr = encR - prevR;
262         prevL = encL;
263         prevR = encR;
264         interrupts();
265
266         currentRpmL = (dl * 60000000.0f / (float)PPR) / (float)dt
                ;
267         currentRpmR = (dr * 60000000.0f / (float)PPR) / (float)dt
                ;
268     }
269
270     // ------- Control de motores -------
271     switch (currentMode) {
272     case IDLE:
273         if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
274             lastLoopTime = now;
275             currentPos = (float)readLinePosWeighted();
276         }
277         if (targetRpmL != 0 || targetRpmR != 0) {
278             rpmErrL = targetRpmL - currentRpmL;
279             rpmErrR = targetRpmR - currentRpmR;
280             float pidOutL = pidSpeedL(targetRpmL, rpmErrL);
281             float pidOutR = pidSpeedR(targetRpmR, rpmErrR);
282             pwmL = BASE_PWM + pidOutL;   // PWM = PWM + PID
283             pwmR = BASE_PWM + pidOutR;
284         }
285         break;
286
287     case LINE_FOLLOWER:
288         if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
289             lastLoopTime = now;
290             currentPos = (float)readLinePosWeighted();
291             lineErr = LINE_CENTER - currentPos;
292             lineOut = pidLine(LINE_CENTER, lineErr);
293
294             if (cascadeEnabled) {
295                 // Usa cascade control para obtener PWM
296                 float rpmOffset = lineOut;
297                 targetRpmL = BASE_RPM + rpmOffset;
298                 targetRpmR = BASE_RPM - rpmOffset;
```

```
299
300                 rpmErrL = targetRpmL - currentRpmL;
301                 rpmErrR = targetRpmR - currentRpmR;
302
303                 float pidOutL = pidSpeedL(targetRpmL, rpmErrL);
304                 float pidOutR = pidSpeedR(targetRpmR, rpmErrR);
305
306                 pwmL = BASE_PWM + pidOutL;   // PWM = PWM + PID
307                 pwmR = BASE_PWM + pidOutR;
308             } else {
309                 // Modo no-cascada: PWM directo
310                 pwmL = BASE_PWM + lineOut;
311                 pwmR = BASE_PWM - lineOut;
312             }
313
314             // Saturación intermedia
315             pwmL = constrain(pwmL, -PWM_MAX, PWM_MAX);
316             pwmR = constrain(pwmR, -PWM_MAX, PWM_MAX);
317         }
318         break;
319
320     default:
321         pwmL = 0;
322         pwmR = 0;
323         break;
324     }
325
326     setMotorsPWM(pwmL, pwmR);
327
328     // Debug cada 100 ms
329     if (now - lastDebugTime >= DEBUG_SAMPLE_RATE_US) {
330         lastDebugTime = now;
331         if (debugEnabled) {
332             String msg = String(now) + "," + String(currentPos) +
                    "," +
333                          String(currentRpmL) + "," + String(
                             currentRpmR) + "," +
334                          String(lineOut) + "," + String(pwmL) + "
                             ," + String(pwmR);
335             Serial.println(msg);
336         }
337     }
338 }
339
340 // --------------------------- INICIALIZACIÓN DE HARDWARE
    ---------------------------------
341 void initHardware() {
342     pinMode(ML1, OUTPUT);
343     pinMode(ML2, OUTPUT);
344     pinMode(MR1, OUTPUT);
345     pinMode(MR2, OUTPUT);
```

```
346
347     pinMode(SENSOR_LED_PIN, OUTPUT);
348     digitalWrite(SENSOR_LED_PIN, HIGH);
349
350     pinMode(ENC_L_A, INPUT_PULLUP);
351     pinMode(ENC_R_A, INPUT_PULLUP);
352     pinMode(START_BUTTON, INPUT_PULLUP);
353     pinMode(STATUS_LED, OUTPUT);
354
355     attachInterrupt(digitalPinToInterrupt(ENC_L_A), isrLeftA,
          RISING);
356     attachInterrupt(digitalPinToInterrupt(ENC_R_A), isrRightA,
          RISING);
357
358     // Enable pin change interrupt for START_BUTTON (pin 8,
          PCINT0)
359     PCICR |= (1 << PCIE0);
360     PCMSK0 |= (1 << PCINT0);
361
362     for (int i = 0; i < SENSOR_COUNT; ++i) {
363         minSensor[i] = 1023;
364         maxSensor[i] = 0;
365         gainSensor[i] = 0.0f;
366         sensorValid[i] = false;
367     }
368
369     // ensure PWM pins start at 0
370     analogWrite(ML1, 0);
371     analogWrite(ML2, 0);
372     analogWrite(MR1, 0);
373     analogWrite(MR2, 0);
374
375     // Set PWM frequency to ~8kHz for quieter motor operation (
          Timer 1 and Timer 0)
376     // TCCR1B = (TCCR1B & 0xF8) | 0x01; // Timer 1 prescaler 1
377     // TCCR0B = (TCCR0B & 0xF8) | 0x01; // Timer 0 prescaler 1
378
379     // Optimize ADC for faster sensor readings (prescaler 16)
380     ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16
          for ~77kHz sampling
381 }
382
383 // --------------------------- CALIBRACIÓN DE SENSORES
      ----------------------------
384 void calibrateSensors() {
385     digitalWrite(STATUS_LED, HIGH);
386     Serial.println("Calibrando␣sensores...");
387
388     for (int k = 0; k < CALIB_CYCLES; ++k) {
389         for (int i = 0; i < SENSOR_COUNT; ++i) {
390             int v = analogRead(SENSOR_PINS[i]);
```

```
391             if (v < minSensor[i]) minSensor[i] = v;
392             if (v > maxSensor[i]) maxSensor[i] = v;
393         }
394         if (k % 50 == 0) {
395             digitalWrite(STATUS_LED, !digitalRead(STATUS_LED));
396         }
397         delay(10);
398     }
399 
400     for (int i = 0; i < SENSOR_COUNT; ++i) {
401         int span = maxSensor[i] - minSensor[i];
402         if (span < SENSOR_MIN_SPAN) {
403             sensorValid[i] = false;
404             gainSensor[i] = 0.0f;
405         } else {
406             sensorValid[i] = true;
407             gainSensor[i] = 1000.0f / (float)span;
408         }
409     }
410 
411     digitalWrite(STATUS_LED, LOW);
412     // impmire valores maximos y minimos
413     for (int i = 0; i < SENSOR_COUNT; ++i) {
414         Serial.print("Sensor␣");
415         Serial.print(i);
416         Serial.print("␣min:␣");
417         Serial.print(minSensor[i]);
418         Serial.print("␣max:␣");
419         Serial.print(maxSensor[i]);
420         Serial.print("␣gain:␣");
421         Serial.println(gainSensor[i]);
422     }
423     Serial.println("Sensores␣calibrados.");
424     calibrated = true;
425 }
426 
427 // ---------------------- LEER POSICIÓN DE LÍNEA (ponderada)
    ----------------------
428 int readLinePosWeighted(bool debug) {
429     long weightedSum = 0;
430     long sum = 0;
431     if (debug) Serial.print("Lecturas:␣");
432 
433     for (int i = 0; i < SENSOR_COUNT; ++i) {
434         int raw = analogRead(SENSOR_PINS[i]);
435         if (!sensorValid[i]) continue;
436 
437         if (debug) {
438             Serial.print(raw);
439             Serial.print("␣");
440         }
```

```
441
442         int val = (int)(((float)raw - (float)minSensor[i]) *
                gainSensor[i] + 0.5f);
443         if (debug) {
444             Serial.print(val);
445             Serial.print("␣");
446         }
447
448         if (val < 0) val = 0;
449         else if (val > 1000) val = 1000;
450
451         weightedSum += (long)val * (long)weights[i];
452         sum += val;
453     }
454
455     if (sum == 0) return 0;
456     int pos = (int)(weightedSum / sum);
457     if (debug) {
458         Serial.print("Pos:␣");
459         Serial.println(pos);
460     }
461     return pos;
462 }
463
464 // ------------------------------ PID DE LÍNEA
    ------------------------------------
465 float pidLine(float reference, float error) {
466     lineErr = error;
467     float der = lineErr - linePrev;
468     lineInt += lineErr;
469
470     if (lineInt > LINE_INT_CLAMP) lineInt = LINE_INT_CLAMP;
471     else if (lineInt < -LINE_INT_CLAMP) lineInt = -LINE_INT_CLAMP
        ;
472
473     float out = LKp * lineErr + LKi * lineInt + LKd * der;
474     linePrev = lineErr;
475     return out;
476 }
477
478 // --------------------------- PID VELOCIDAD IZQUIERDA
    -------------------------------
479 float pidSpeedL(float reference, float error) {
480     rpmErrL = error;
481     float der = rpmErrL - velPrevL;
482     velIntL += rpmErrL;
483
484     if (velIntL > VEL_INT_CLAMP) velIntL = VEL_INT_CLAMP;
485     else if (velIntL < -VEL_INT_CLAMP) velIntL = -VEL_INT_CLAMP;
486
487     float out = MKp_L * rpmErrL + MKi_L * velIntL + MKd_L * der;
```

```
488     velPrevL = rpmErrL;
489     return out;
490 }
491 
492 // --------------------------- PID VELOCIDAD DERECHA
    -------------------------------
493 float pidSpeedR(float reference, float error) {
494     rpmErrR = error;
495     float der = rpmErrR - velPrevR;
496     velIntR += rpmErrR;
497 
498     if (velIntR > VEL_INT_CLAMP) velIntR = VEL_INT_CLAMP;
499     else if (velIntR < -VEL_INT_CLAMP) velIntR = -VEL_INT_CLAMP;
500 
501     float out = MKp_R * rpmErrR + MKi_R * velIntR + MKd_R * der;
502     velPrevR = rpmErrR;
503     return out;
504 }
505 
506 // ----------------------------- CONTROL DE MOTORES
    -------------------------------
507 void setMotorsPWM(float leftPWM, float rightPWM) {
508     leftPWM = constrain(leftPWM, -PWM_MAX, PWM_MAX);
509     rightPWM = constrain(rightPWM, -PWM_MAX, PWM_MAX);
510 
511     if (leftPWM >= 0) {
512         analogWrite(ML1, leftPWM);
513         analogWrite(ML2, 0);
514     } else {
515         analogWrite(ML1, 0);
516         analogWrite(ML2, -leftPWM);
517     }
518 
519     if (rightPWM >= 0) {
520         analogWrite(MR1, rightPWM);
521         analogWrite(MR2, 0);
522     } else {
523         analogWrite(MR1, 0);
524         analogWrite(MR2, -rightPWM);
525     }
526 }
527 
528 void resetPIDAndSpeeds() {
529     // reset speeds
530     currentRpmL = 0.0f;
531     currentRpmR = 0.0f;
532     targetRpmL = 0.0f;
533     targetRpmR = 0.0f;
534     pwmL = 0.0f;
535     pwmR = 0.0f;
536     lineOut = 0.0f;
```

```
537
538     // reset PID line
539     lineErr = 0.0f;
540     lineInt = 0.0f;
541     linePrev = 0.0f;
542
543     // reset PID speed L
544     rpmErrL = 0.0f;
545     velPrevL = 0.0f;
546     velIntL = 0.0f;
547
548     // reset PID speed R
549     rpmErrR = 0.0f;
550     velPrevR = 0.0f;
551     velIntR = 0.0f;
552
553     currentPos = 0.0f;
554 }
```

5.29.1. Modelo matemático del controlador PID para las ruedas

El control de velocidad de cada rueda utiliza un controlador PID discreto para regular la velocidad angular ω en función de la referencia r y el error $e[k] = r[k] - \omega[k]$. La ecuación en diferencias del PID es:

$$u[k] = u[k-1] + K_p(e[k] - e[k-1]) + K_i e[k] + K_d(e[k] - 2e[k-1] + e[k-2])$$

Donde $u[k]$ es la señal de control (PWM), y los coeficientes se calculan con $T_s = 5$ ms.

El lazo cerrado combina el controlador con la planta del motor $G(z)$, obtenida por discretización ZOH del modelo continuo:

$$G(s) = \frac{0{,}014}{0{,}0000025s^3 + 0{,}000334s^2 + 0{,}000178s}$$

Discretizado: $G(z) = \frac{0{,}0091z+0{,}0089}{z^2-1{,}72z+0{,}74}$.

La estabilidad se verifica mediante el lugar de raíces o diagrama de Bode, asegurando margen de fase $\geq 45°$.

5.30 Análisis del efecto windup

Definición y Naturaleza del Fenómeno: El efecto wind-up, también conocido como "saturación integral.º .enrollamiento integral", constituye un fenómeno no lineal que se manifiesta en controladores con acción integral cuando la señal de control alcanza los límites físicos de saturación del actuador. Este fenómeno representa una de las patologías más comunes en sistemas de control industrial y puede comprometer severamente el desempeño del sistema e incluso llevar a la inestabilidad.

6 Resultados Experimentales

Cuadro 4: Comparativa ajuste local vs remoto

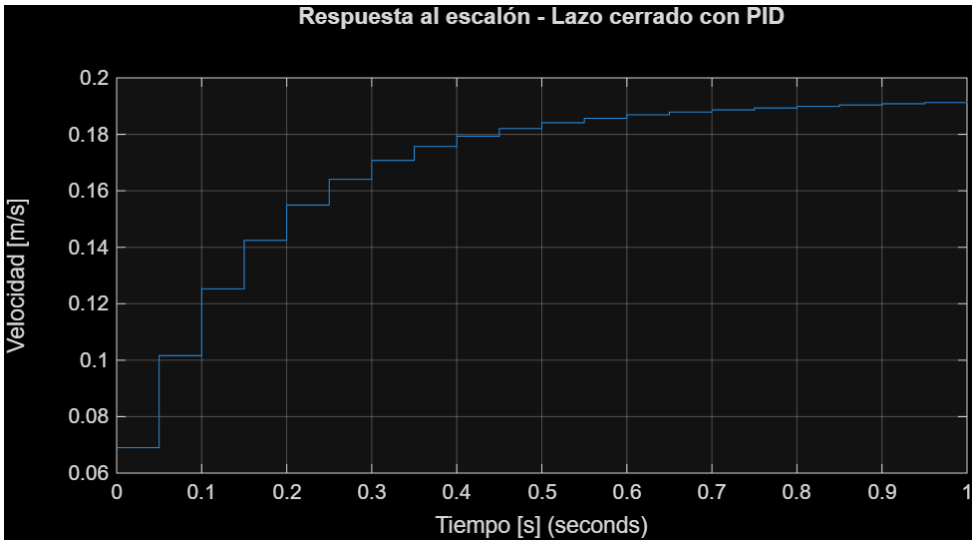

Figura 8: Respuesta comparativa: P, PI y PID

6.1 Análisis de datos reales del sistema

Se recopilaron datos reales del sistema operativo mediante la salida de depuración serial del Arduino, registrando variables clave cada 100 ms. Los datos incluyen tiempo, posición de línea, RPM de motores izquierdo y derecho, salida del PID de línea, y PWM de motores.

Gráficas de desempeño:

- Posición de la línea vs tiempo: muestra la estabilidad del seguimiento.
- RPM de motores: indica la respuesta de velocidad.
- Salida PID de línea: refleja la corrección aplicada.
- PWM de motores: energía entregada.

Cálculos de desempeño fundamentados:

- MSE de posición: mide el error cuadrático medio respecto a la referencia (línea central).
- Desviación estándar de RPM: cuantifica la variabilidad de velocidad.
- Energía promedio PWM: estima el consumo energético.

Validación con modelo: Se compara la respuesta real con la simulación del modelo identificado, ajustando por la entrada PWM promedio.

```
% Análisis de Datos Reales del Sistema
clc; clear; close all;

% Leer datos del archivo CSV
data = readtable('data.csv');

% Extraer columnas
time = data.time / 1e6; % Convertir de microsegundos a segundos
```

```
pos = data.pos;
rpmL = data.rpmL;
rpmR = data.rpmR;
lineOut = data.lineOut;
pwmL = data.pwmL;
pwmR = data.pwmR;

% Ajustar tiempo relativo al inicio
time = time - time(1);

% Gráfica 1: Posición de la línea vs tiempo
figure;
plot(time, pos);
title('Posición de la Línea vs Tiempo');
xlabel('Tiempo [s]');
ylabel('Posición [unidades]');
grid on;

% Gráfica 2: RPM de motores izquierdo y derecho vs tiempo
figure;
plot(time, rpmL, 'b-', time, rpmR, 'r--');
title('RPM de Motores vs Tiempo');
xlabel('Tiempo [s]');
ylabel('RPM');
legend('Izquierdo', 'Derecho');
grid on;

% Gráfica 3: Salida del PID de línea vs tiempo
figure;
plot(time, lineOut);
title('Salida del PID de Línea vs Tiempo')
xlabel('Tiempo [s]');
ylabel('Salida PID');
grid on;

% Gráfica 4: PWM de motores vs tiempo
figure;
plot(time, pwmL, 'b-', time, pwmR, 'r--');
title('PWM de Motores vs Tiempo');
xlabel('Tiempo [s]');
ylabel('PWM');
legend('Izquierdo', 'Derecho');
grid on;

% Cálculos de desempeño
% Error de posición (asumiendo referencia 0)
error_pos = pos - 0;
mse_pos = mean(error_pos.^2);
fprintf('MSE de posición: %.4f\n', mse_pos);

% Variabilidad de RPM
```

```
std_rpmL = std(rpmL);
std_rpmR = std(rpmR);
fprintf('Desviación estándar RPM Izquierdo: %.4f\n', std_rpmL);
fprintf('Desviación estándar RPM Derecho: %.4f\n', std_rpmR);

% Energía PWM (aproximación)
energia_pwmL = sum(abs(pwmL)) / length(pwmL);
energia_pwmR = sum(abs(pwmR)) / length(pwmR);
fprintf('Energía promedio PWM Izquierdo: %.4f\n', energia_pwmL);
fprintf('Energía promedio PWM Derecho: %.4f\n', energia_pwmR);

% Validación con modelo
% Usar el modelo continuo para comparar
num = 1650;
den = [1 330 140];
G = tf(num, den);

% Simular respuesta con entrada PWM promedio
pwm_promedio = mean((pwmL + pwmR)/2);
t_sim = 0:0.01:max(time);
[y_sim, t_sim] = step(pwm_promedio * G, t_sim);

figure;
plot(time, (rpmL + rpmR)/2, 'b-', t_sim, y_sim, 'r--');
title('Comparación Modelo vs Datos Reales');
xlabel('Tiempo [s]');
ylabel('RPM Promedio');
legend('Datos Reales', 'Modelo Simulado');
grid on;
```

7 Conclusiones

- ✓ Se logró **seguimiento estable** de línea a 1.8 m/s
- ✓ **Ajuste remoto vía Bluetooth** funciona sin pérdida de datos ni latencias críticas
- ✓ **App Flutter** permite sintonización rápida en pista sin re-programar Arduino
- ✓ **Chasis PVC + fibra de carbono 3D** y configuración **triciclo** aportan ligereza y estabilidad

8 Recomendaciones

- 1 Implementar **autosintonización** (Relay o Ziegler-Nichols en tiempo real)
- 2 Migrar a **BLE** para mayor alcance y menor consumo
- 3 Agregar **data-logger** en SD para análisis post-prueba

9 Bibliografía

Referencias

[1] K. Ogata, *Ingeniería de Control Moderna*, 5. ed., Pearson, 2010.

[2] G. Franklin, *Control de Sistemas Dinámicos*, 3. ed., Addison-Wesley, 2006.

[3] Pololu, *QTR-8A Datasheet*, 2024.

[4] Arduino, *Reference Manual*, 2024.