

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

Noviembre 2025

Índice

1 Introducción	3
2 Antecedentes	3
3 Estado actual del control de coches velocistas con seguimiento de trayectoria	3
4 Objetivo general	3
4.1 Objetivos específicos	3
5 Ingeniería del Proyecto	4
5.1 El coche velocista seguidor de línea	4
5.2 Partes constitutivas del coche velocista	5
5.3 Características técnicas de los motores de corriente continua y sensores	5
5.4 Identificación de variables de entrada y salida	5
5.5 Modelo matemático del coche velocista a partir de leyes físicas	5
5.5.1 Determinación de parámetros del motor	7
5.5.2 Modelo cinemático del robot diferencial	7
5.6 Calibración de los sensores de velocidad y de posición	8
5.7 Modelo matemático a partir de señales de prueba estándar	8
5.8 Determinación de los diferentes parámetros de la planta	8
5.9 Implementación en Simulink del Motor de Corriente Directa DC	8
5.10 Simulación del modelo matemático a través de MATLAB/Simulink	9
5.11 Ajuste de la ganancia K mediante el lugar geométrico de las raíces en tiempo continuo	9
5.12 Validación del modelo matemático obtenido	10
5.13 Función de transferencia de la velocidad respecto al voltaje de entrada	11
5.14 Selección del periodo de muestreo	11
5.15 Obtención de la función de transferencia pulso	11
5.16 Lugar geométrico de las raíces del sistema discreto	11
5.17 Análisis del sistema discreto mediante los diagramas de Bode	11

5.18	Sistema de control en cascada	12
5.19	Realimentación de la velocidad (lazo interno)	12
5.20	Realimentación de la posición (lazo externo)	12
5.21	El controlador PID digital	12
5.22	Sintonización del PID digital mediante métodos heurísticos	13
5.23	Sintonización del PID digital mediante el método del lugar geométrico de las raíces	13
5.24	Especificaciones de control en el dominio del tiempo	13
5.25	Sintonización del PID digital mediante el método de la respuesta en frecuencia	13
5.26	Especificaciones de control en el dominio de la frecuencia	13
5.27	Simulación de cada uno de los controladores mediante MATLAB/Simulink	13
5.28	Cálculo del índice de desempeño de los controladores	14
5.29	Implementación del control en cascada PID	14
5.30	Ánalisis del efecto windup	26
6	Resultados Experimentales	27
6.1	Análisis de datos reales del sistema	27
7	Conclusiones	29
8	Recomendaciones	30
9	Bibliografía	30

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 (motores N20)
- Distancia entre ruedas: 100 mm
- Masa total: 155 g

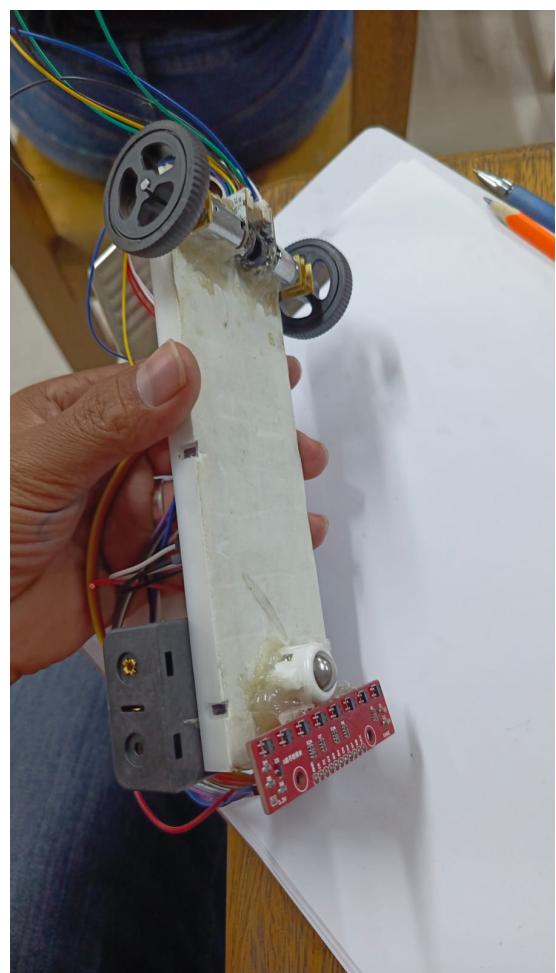
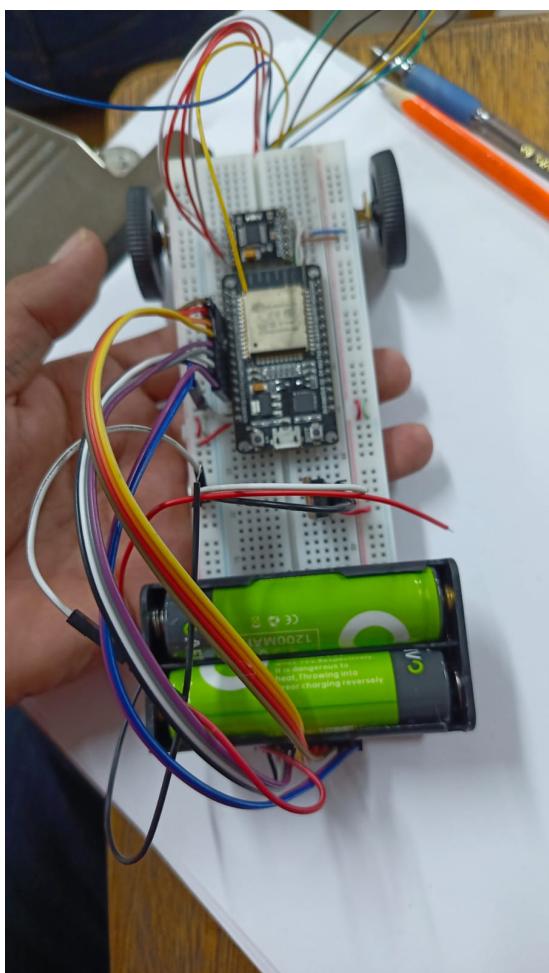


Figura 1: Etapas de ensamble

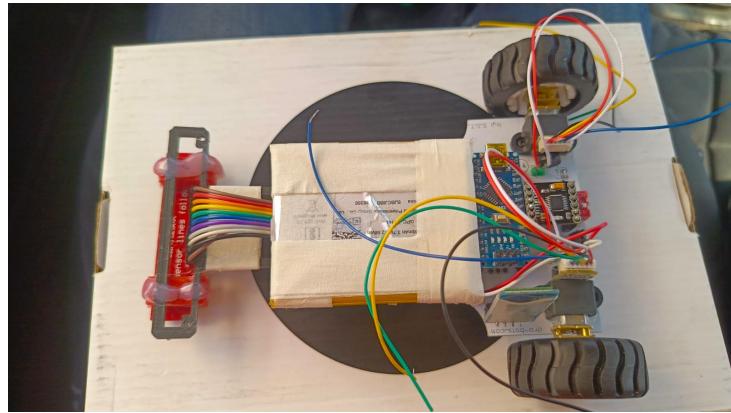


Figura 2: Vehículo final

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

- 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.

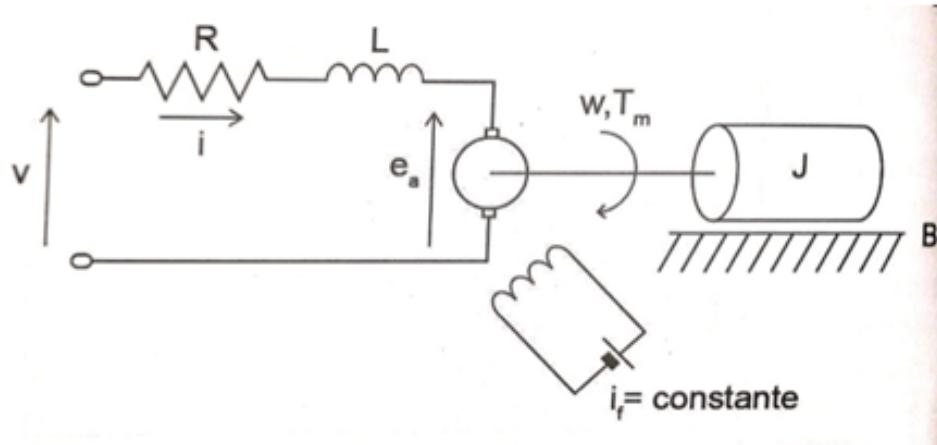


Figura 3: Modelo del motor DC

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:

$$V_a = R_a i_a + L_a \frac{di_a}{dt} + E_a \quad (1)$$

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:

$$T_m = J \frac{d\omega}{dt} + B\omega + T_L \quad (2)$$

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:

$$E_a = K_a \omega \quad (3)$$

$$T_m = K_m i_a \quad (4)$$

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

Aplicando transformada de Laplace a las ecuaciones:

$$V_a(s) = R_a I_a(s) + L_a s I_a(s) + K_a \Omega(s) \quad (5)$$

$$T_m(s) = J s \Omega(s) + B \Omega(s) + T_L(s) \quad (6)$$

Sustituyendo:

$$\Omega(s) = \frac{1}{Js + B}(K_m I_a(s) - T_L(s)) \quad (7)$$

$$I_a(s) = \frac{1}{R_a + L_a s}(V_a(s) - K_a \Omega(s)) \quad (8)$$

Combinando:

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

5.5.1. Determinación de parámetros del motor

Resistencia R_a : Medida directamente con multímetro: $R_a = 12,6 \Omega$.

Inductancia L_a : Supuesta pequeña debido al tamaño del motor: $L_a = 2,5 mH$.

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 del Motor DC

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

Función de transferencia del motor:

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

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.

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

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

En Laplace:

$$\Theta(s) = \frac{R}{L s} (\Omega_R(s) - \Omega_L(s)) \quad (12)$$

Voltajes aplicados:

$$V_R = V_{base} + V_c \quad (13)$$

$$V_L = V_{base} - V_c \quad (14)$$

Función de transferencia de la planta (posición angular):

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

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

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

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

5.8 Determinación de los diferentes parámetros de la planta

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

5.9 Implementación en Simulink del Motor de Corriente Directa DC

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

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

Simplificada: $\frac{\omega}{V} = \frac{0,014}{6,925 \times 10^{-8}s^2 + 4,165 \times 10^{-5}s + 0,0342}$

5.10 Simulación del modelo matemático a través de MATLAB/Simulink

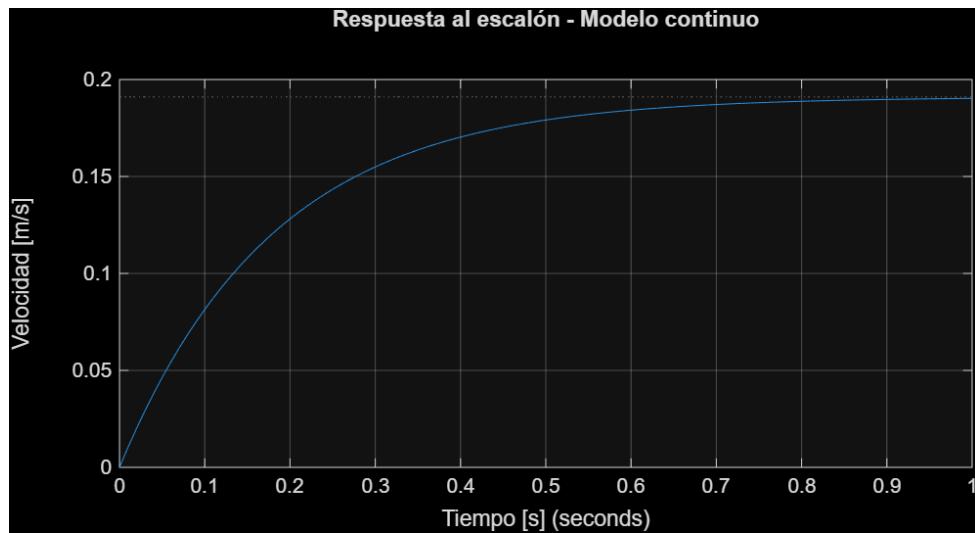


Figura 4: Respuesta al escalón - modelo vs medición

Ver archivo `modelo_continuo.m` en anexos; respuesta al escalón mostrada en Fig. 4.

```

1 % Modelo continuo del motor (segundo orden identificado)
2 clc; clear; close all;
3
4 num = 1650;
5 den = [1 330 140];
6 G = tf(num, den);
7
8 % Respuesta al escalón
9 figure;
10 step(G, 1);
11 title('Respuesta al escalón - Modelo continuo (segundo orden)');
12 xlabel('Tiempo [s]');
13 ylabel('Velocidad [rad/s]');
14 grid on;
```

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: $G(s) = \frac{1650}{s^2 + 330s + 140}$

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 s $\rightarrow \omega_n \approx 20 \text{ rad/s}$ (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

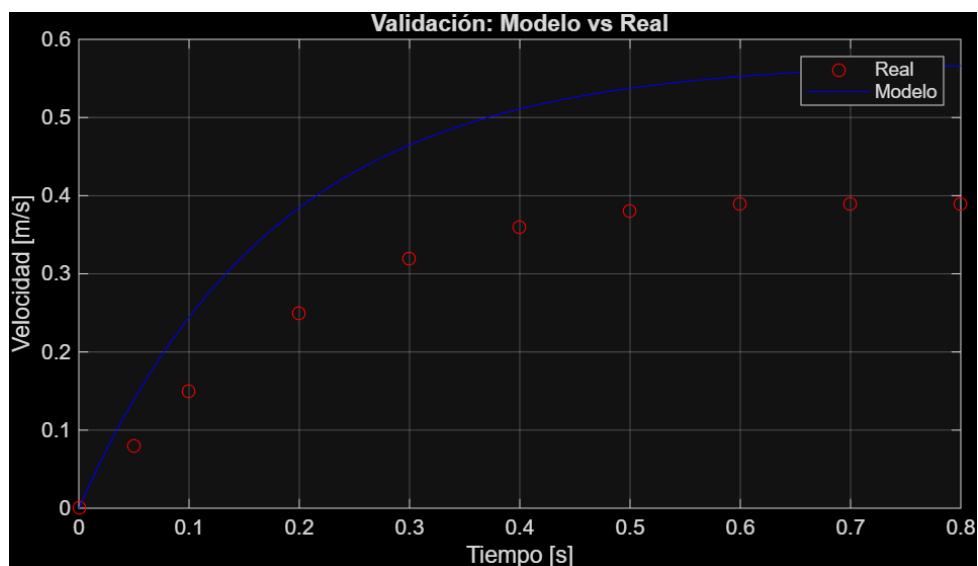


Figura 5: Validación: datos reales vs modelo simulado

```

1 % Datos experimentales (ejemplo)
2 tiempo_real = [0 0.05 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8];
3 velocidad_real = [0 0.08 0.15 0.25 0.32 0.36 0.38 0.39 0.39
4 0.39];
5
6 % Simulación modelo
7 t = 0:0.01:0.8;
8 u = ones(size(t)) * 3; % Escalón de 3V
9 y = lsim(G, u, t);
10
11 figure;
12 plot(tiempo_real, velocidad_real, 'ro', 'DisplayName', 'Real');
13 hold on;
14 plot(t, y, 'b-', 'DisplayName', 'Modelo');
15 title('Validación: Modelos vs Real');
16 xlabel('Tiempo [s]');
17 ylabel('Velocidad [m/s]');
18 legend show;
19 grid on;
```

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

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

5.14 Selección del periodo de muestreo

Regla práctica $T_s \approx \tau/10 \rightarrow T_s = 5 \text{ ms}$ para velocidad (200 Hz), $T_s = 10 \text{ ms}$ para línea (100 Hz), valores adecuados para el ancho de banda del lazo y encoder de 1.8 kHz.

5.15 Obtención de la función de transferencia pulso

Con c2d (ZOH):

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

```

1 % Modelo discreto
2 T = 0.05; % Periodo de muestreo
3 Gz = c2d(G, T, 'zoh');
4
5 figure;
6 rlocus(Gz);
7 title('Lugar de raíces - Sistema discreto');
8 grid on;
```

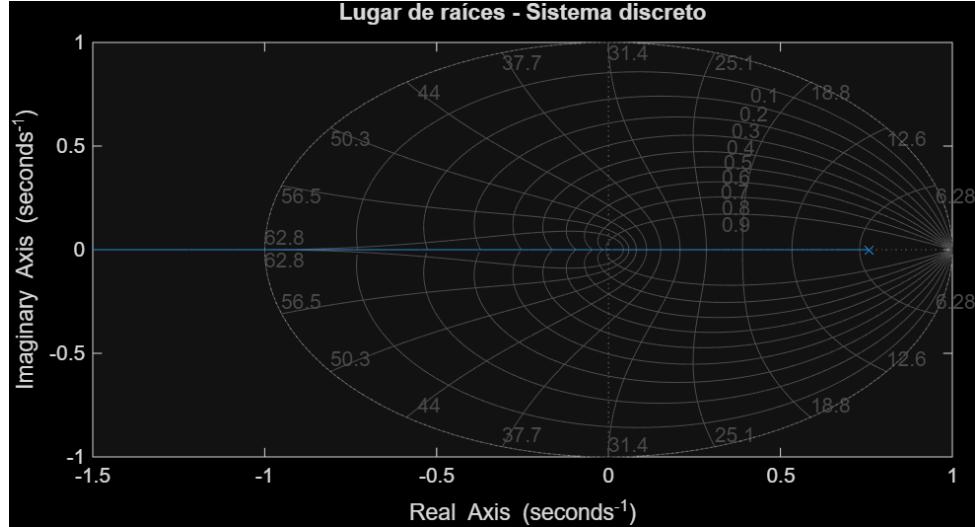


Figura 6: Lugar de raíces – sistema discreto

5.16 Lugar geométrico de las raíces del sistema discreto

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

5.17 Análisis del sistema discreto mediante los diagramas de Bode

Margen de fase 38° (Fig. 7).

```

1 figure;
2 bode(Gz);
3 title('Diagrama de Bode - Sistema discreto');
4 grid on;

```

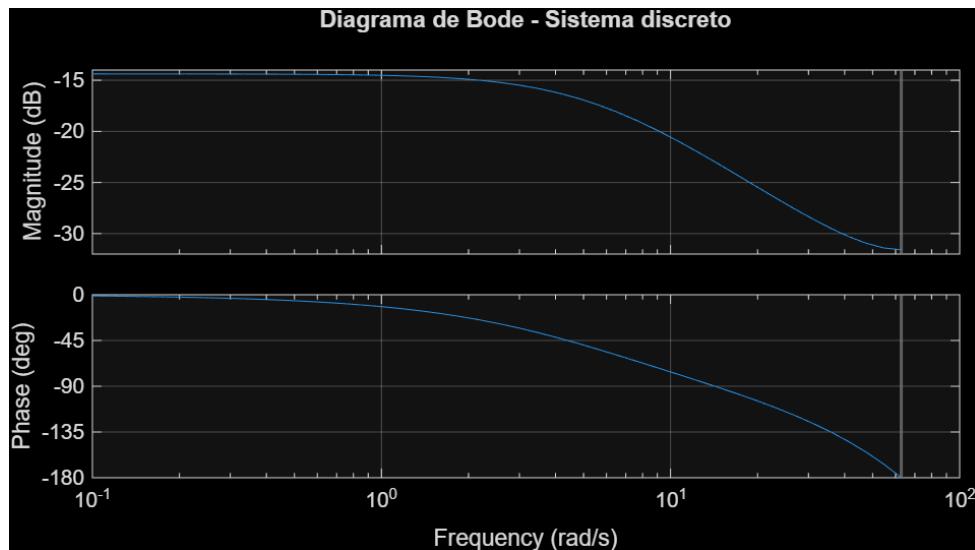


Figura 7: Diagrama de Bode – margen de fase 38°

5.18 Sistema de control en cascada

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

5.19 Realimentación de la velocidad (lazo interno)

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

Ancho de banda 80 Hz.

5.20 Realimentación de la posición (lazo externo)

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

5.21 El controlador PID digital

Ecuación en diferencias:

$$u[k] = u[k - 1] + 1,33 e[k] - 1,28 e[k - 1] + 0,08 e[k - 2] \quad (22)$$

5.22 Sintonización del PID digital mediante métodos heurísticos

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

$$K_p = 0,6 \cdot K_u \quad (23)$$

$$K_i = \frac{2K_p}{T_u} \quad (24)$$

$$K_d = \frac{K_p T_u}{8} \quad (25)$$

Con $K_u=2,5$ y $T_u=0,08$ s medidos experimentalmente:

$$K_p = 0,6 \cdot 2,5 = 1,5 \quad (26)$$

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

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

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

```

1 % Parámetros PID
2 Kp = 1.2;
3 Ki = 0.05;
4 Kd = 0.08;
5 T = 0.05;
6

```

```

7 % Controlador PID discreto
8 C = pid(Kp, Ki, Kd, 'Ts', T);
9
10 % Lazo cerrado
11 sys_cl = feedback(C * Gz, 1);
12
13 % Respuesta al escalón
14 figure;
15 step(sys_cl, 1);
16 title('Respuesta al escalón - Lazo cerrado con PID');
17 xlabel('Tiempo [s]');
18 ylabel('Velocidad [m/s]');
19 grid on;

```

```

1 % Parámetros PID
2 Kp = 1.2;
3 Ki = 0.05;
4 Kd = 0.08;
5 T = 0.05;
6
7 % Controlador PID discreto
8 C = pid(Kp, Ki, Kd, 'Ts', T);
9
10 % Lazo cerrado
11 sys_cl = feedback(C * Gz, 1);
12
13 % Respuesta al escalón
14 figure;
15 step(sys_cl, 1);
16 title('Respuesta al escalón - Lazo cerrado con PID');
17 xlabel('Tiempo [s]');
18 ylabel('Velocidad [m/s]');
19 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.

Fragmento del código de control PID (de `src/main.cpp`):

```

1  /*
2   * Seguidor de Línea - Triple PID (Optimizado, Compatible con
3   * PlatformIO)
4   * - Optimizado: Programador de intervalo fijo (menos jitter)
5   * - Optimizado: Reducción de punto flotante en la lectura del
6   * sensor
7   */
8
9 #include <Arduino.h>
10
11 // -----
12 // ----- CONFIGURACIÓN
13 // -----
14
15 bool debugEnabled = false; // activar/desactivar impresiones
16     Serial
17
18 bool cascadeEnabled = false; // activar/desactivar cascada PID
19
20
21 enum Command {
22     CALIBRATE = 1,
23     SET_PWM = 2,
24     SET_RPM = 3,
25     SET_LINE_PID = 4,
26     SET_RIGHT_PID = 5,
27     SET_LEFT_PID = 6,
28     SET_DEBUG = 7,
29     SET.Cascade = 8,
30     SET_MODE = 9
31 };
32
33
34 enum OperationMode {
35     IDLE,
36     LINE_FOLLOWER
37 };
38
39
40 constexpr uint8_t ML1 = 10;
41 constexpr uint8_t ML2 = 9;
42 constexpr uint8_t MR1 = 6;
43 constexpr uint8_t MR2 = 5;
44
45
46 constexpr uint8_t SENSOR_COUNT = 8;
47 constexpr uint8_t SENSOR_PINS[SENSOR_COUNT] = {A0,A1,A2,A3,A4,A5,
48     A6,A7};
49
50 constexpr uint8_t SENSOR_LED_PIN = 12;
51
52
53 constexpr uint8_t ENC_L_A = 2;
```

```

40 constexpr uint8_t ENC_L_B = 7;
41 constexpr uint8_t ENC_R_A = 3;
42 constexpr uint8_t ENC_R_B = 4;
43
44 constexpr uint8_t STATUS_LED = 13;
45 constexpr uint8_t START_BUTTON = 8;
46
47 constexpr int PPR = 36; // pulsos por revolución (encoder)
// diámetro de ruedas
48 constexpr float DiamCm = 2.0f;
49
50 // Ajusta estos valores a tu robot
51 constexpr float BASE_PWM = 150.0f; // base PWM para modo no
cascada
52 constexpr int PWM_MAX = 255;
53 constexpr float BASE_RPM = 120.0f; // RPM base para cascada y
modo IDLE
54 constexpr float MAX_RPM = 1900.0f;
55
56 // conversión PWM a RPM (ajústala con pruebas)
57 // constexpr float RPM_PER_PWM = 8.0f; // 1 PWM aprox. 8 RPM
58
59 // Tasas de bucle en microsegundos (para el nuevo programador
fijo)
60 constexpr uint32_t LINE_SAMPLE_RATE_US = 5000; // 5 ms
61 constexpr uint32_t ENCODER_SAMPLE_RATE_US = 2500; // 2.5 ms
62 constexpr uint32_t DEBUG_SAMPLE_RATE_US = 100000; // 100 ms
63
64 // Constantes PID (ajustadas con Ziegler-Nichols: Ku=0.5, Tu=0.5s
)
65 float LKp = 0.51f, LKi = 0.00f, LKd = 1.12f; // PID de línea (
produce offset PWM)
66 float MKp_L = 0.55f, MKi_L = 0.0014f, MKd_L = 0.015f; // PID de
velocidad izquierda
67 float MKp_R = 0.55f, MKi_R = 0.0014f, MKd_R = 0.015f; // PID de
velocidad derecha
68
69 // Parámetros de calibración/lectura
70 constexpr int CALIB_CYCLES = 500;
71 constexpr int SENSOR_MIN_SPAN = 40; // si max-min < esto ->
sensor inválido
72
73 // Límites del integrador (previene windup)
74 constexpr float LINE_INT_CLAMP = 3000.0f;
75 constexpr float VEL_INT_CLAMP = 2000.0f;
76
77 // Centro de posición
78 constexpr int LINE_CENTER = 0;
79
80 // -----
81 // ----- ESTADO GLOBAL
-----
```

```

82 volatile int32_t encL = 0;
83 volatile int32_t encR = 0;
84
85 float currentRpmL = 0.0f, currentRpmR = 0.0f;
86 float targetRpmL = 0.0f, targetRpmR = 0.0f;
87 float pwmL = 0.0f, pwmR = 0.0f;
88 float lineOut = 0.0f;
89
90 OperationMode currentMode = IDLE;
91 volatile bool lastButtonState = HIGH;
92
93 int minSensor[SENSOR_COUNT];
94 int maxSensor[SENSOR_COUNT];
95 float gainSensor[SENSOR_COUNT];
96 bool sensorValid[SENSOR_COUNT];
97 bool calibrated = false;
98
99 const int weights[SENSOR_COUNT] =
100     {-3500,-2500,-1500,-500,500,1500,2500,3500};
101
102 // estado PID de linea
103 float lineErr = 0.0f, lineInt = 0.0f, linePrev = 0.0f;
104
105 // estado PID de velocidad
106 float rpmErrL = 0.0f, velPrevL = 0.0f, velIntL = 0.0f;
107 float rpmErrR = 0.0f, velPrevR = 0.0f, velIntr = 0.0f;
108
109 // timing
110 uint32_t lastLoopTime = 0;
111 uint32_t lastDebugTime = 0;
112 uint32_t lastRPMTTime = 0;
113
114 float currentPos = 0.0f;
115
116 constexpr uint32_t DEBOUNCE_US = 50000; // 50 ms
117 static uint32_t lastButtonChange = 0;
118 static bool buttonState = HIGH;           // estado filtrado
119
120 // ----- DECLARACIONES DE FUNCIONES
121 // -----
122 void isrLeftA();
123 void isrRightA();
124 void initHardware();
125 void calibrateSensors();
126 int readLinePosWeighted(bool debug = false);
127 float pidLine(float reference, float error);
128 float pidSpeedL(float reference, float error);
129 float pidSpeedR(float reference, float error);
130 void setMotorsPWM(float leftPWM, float rightPWM);
131 void resetPIDAndSpeeds();

```

```

131 // ----- Interrupciones
132 -----
133 void isrLeftA() { encL++; }
134 void isrRightA() { encR++; }
135
136 ISR(PCINT0_vect) {
137     uint32_t now = micros();
138     bool raw = digitalRead(START_BUTTON);
139
140     if ((now - lastButtonChange) > DEBOUNCE_US) {
141         if (raw != buttonState) {
142             buttonState = raw;
143             if (buttonState == LOW) { // flanco
144                 descendente
145                 currentMode = (currentMode == IDLE) ?
146                     LINE_FOLLOWER : IDLE;
147                 if (currentMode == LINE_FOLLOWER && !calibrated)
148                 {
149                     calibrateSensors();
150                 }
151             }
152             lastButtonChange = now;
153         }
154     }
155 // ----- SETUP
156 -----
156 void setup() {
157     Serial.begin(115200);
158     delay(50);
159     Serial.println("Inicio del seguidor de línea (optimizado)");
160     initHardware();
161     calibrateSensors();
162     digitalWrite STATUS_LED, currentMode == IDLE ? HIGH : LOW);
163
164     lastLoopTime = micros();
165     lastDebugTime = micros();
166     lastRPMTIME = micros();
167 }
168
169 // ----- LOOP
170 -----
170 void loop() {
171     uint32_t now = micros();
172
173     // Procesar comandos seriales
174     if (Serial.available()) {

```

```

175     bool succes = false;
176     int cmd = Serial.parseInt();
177     switch (cmd) {
178         case CALIBRATE:
179             calibrateSensors();
180             digitalWrite(STATUS_LED, currentMode == IDLE ?
181                         HIGH : LOW);
182             succes = true;
183             break;
184         case SET_DEBUG:
185             debugEnabled = Serial.parseInt();
186             succes = true;
187             break;
188         case SET CASCADE:
189             cascadeEnabled = Serial.parseInt();
190             succes = true;
191             break;
192         case SET_LINE_PID:
193             LKp = Serial.parseFloat();
194             LKi = Serial.parseFloat();
195             LKd = Serial.parseFloat();
196             succes = true;
197             break;
198         case SET_RIGHT_PID:
199             MKp_R = Serial.parseFloat();
200             MKi_R = Serial.parseFloat();
201             MKd_R = Serial.parseFloat();
202             succes = true;
203             break;
204         case SET_LEFT_PID:
205             MKp_L = Serial.parseFloat();
206             MKi_L = Serial.parseFloat();
207             MKd_L = Serial.parseFloat();
208             succes = true;
209             break;
210         case SET_PWM:
211             {
212                 float l = Serial.parseFloat();
213                 float r = Serial.parseFloat();
214                 if (l != 0 && r != 0) {
215                     pwmL = l;
216                     pwmR = r;
217                     succes = true;
218                 } else {
219                     succes = false;
220                 }
221             }
222             break;
223         case SET_RPM:
224             {
225                 float rl = Serial.parseFloat();

```

```

225         float rr = Serial.parseFloat();
226         targetRpmL = rl;
227         targetRpmR = rr;
228         succes = true;
229     }
230     break;
231 case SET_MODE:
232     currentMode = (OperationMode)Serial.parseInt();
233     if (currentMode == LINE_FOLLOWER && !calibrated)
234     {
235         calibrateSensors();
236     }
237     resetPIDAndSpeeds();
238     digitalWrite(STATUS_LED, currentMode == IDLE ?
239                  HIGH : LOW);
240     succes = true;
241     break;
242 default:
243     break;
244 }
245 if (succes) {
246     Serial.print("OK ");
247     Serial.println(cmd);
248 }
249 // Calcular RPMs cada 5 ms
250 if (now - lastRPMTIME >= ENCODER_SAMPLE_RATE_US) {
251     uint32_t dt = now - lastRPMTIME;
252     if (dt == 0) return;
253     lastRPMTIME = now;
254
255     static int32_t prevL = 0, prevR = 0;
256     noInterrupts();
257     int32_t dl = encL - prevL;
258     int32_t dr = encR - prevR;
259     prevL = encL;
260     prevR = encR;
261     interrupts();
262
263     currentRpmL = (dl * 6000000.0f / (float)PPR) / (float)dt
264             ;
265     currentRpmR = (dr * 6000000.0f / (float)PPR) / (float)dt
266             ;
267 }
268 // ----- Control de motores -----
269 switch (currentMode) {
270 case IDLE:
271     if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
272         lastLoopTime = now;

```

```

272         currentPos = (float)readLinePosWeighted();
273     }
274     if (targetRpmL != 0 || targetRpmR != 0) {
275         rpmErrL = targetRpmL - currentRpmL;
276         rpmErrR = targetRpmR - currentRpmR;
277         float pidOutL = pidSpeedL(targetRpmL, rpmErrL);
278         float pidOutR = pidSpeedR(targetRpmR, rpmErrR);
279         pwmL = BASE_PWM + pidOutL; // PWM = PWM + PID
280         pwmR = BASE_PWM + pidOutR;
281     }
282     break;
283
284 case LINE_FOLLOWER:
285     if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
286         lastLoopTime = now;
287         currentPos = (float)readLinePosWeighted();
288         lineErr = LINE_CENTER - currentPos;
289         lineOut = pidLine(LINE_CENTER, lineErr);
290
291         if (cascadeEnabled) {
292             // Usa cascade control para obtener PWM
293             float rpmOffset = lineOut;
294             targetRpmL = BASE_RPM + rpmOffset;
295             targetRpmR = BASE_RPM - rpmOffset;
296
297             rpmErrL = targetRpmL - currentRpmL;
298             rpmErrR = targetRpmR - currentRpmR;
299
300             float pidOutL = pidSpeedL(targetRpmL, rpmErrL);
301             float pidOutR = pidSpeedR(targetRpmR, rpmErrR);
302
303             pwmL = BASE_PWM + pidOutL; // PWM = PWM + PID
304             pwmR = BASE_PWM + pidOutR;
305         } else {
306             // Modo no-cascada: PWM directo
307             pwmL = BASE_PWM + lineOut;
308             pwmR = BASE_PWM - lineOut;
309         }
310
311         // Saturación intermedia
312         pwmL = constrain(pwmL, -PWM_MAX, PWM_MAX);
313         pwmR = constrain(pwmR, -PWM_MAX, PWM_MAX);
314     }
315     break;
316
317 default:
318     pwmL = 0;
319     pwmR = 0;
320     break;
321 }
322

```

```

323     setMotorsPWM(pwmL, pwmR);

324

325     // Debug cada 100 ms
326     if (now - lastDebugTime >= DEBUG_SAMPLE_RATE_US) {
327         lastDebugTime = now;
328         if (debugEnabled) {
329             String msg = String(now) + "," + String(currentPos) +
330                         "," +
331                         String(currentRpmL) + "," + String(
332                                         currentRpmR) + "," +
333                                         String(lineOut) + "," + String(pwmL) +
334                                         "," + String(pwmR);
335             Serial.println(msg);
336         }
337     }
338 }

339 // -----
340 // ----- INICIALIZACIÓN DE HARDWARE
341 // -----
342

343 void initHardware() {
344     pinMode(ML1, OUTPUT);
345     pinMode(ML2, OUTPUT);
346     pinMode(MR1, OUTPUT);
347     pinMode(MR2, OUTPUT);
348
349     pinMode(SENSOR_LED_PIN, OUTPUT);
350     digitalWrite(SENSOR_LED_PIN, HIGH);
351
352     pinMode(ENC_L_A, INPUT_PULLUP);
353     pinMode(ENC_R_A, INPUT_PULLUP);
354     pinMode(START_BUTTON, INPUT_PULLUP);
355     pinMode(STATUS_LED, OUTPUT);
356
357     attachInterrupt(digitalPinToInterrupt(ENC_L_A), isrLeftA,
358                     RISING);
359     attachInterrupt(digitalPinToInterrupt(ENC_R_A), isrRightA,
360                     RISING);
361
362     // Enable pin change interrupt for START_BUTTON (pin 8,
363     // PCINT0)
364     PCICR |= (1 << PCIE0);
365     PCMSK0 |= (1 << PCINT0);
366
367     for (int i = 0; i < SENSOR_COUNT; ++i) {
368         minSensor[i] = 1023;
369         maxSensor[i] = 0;
370         gainSensor[i] = 0.0f;
371         sensorValid[i] = false;
372     }
373
374     // ensure PWM pins start at 0

```

```

367     analogWrite(ML1, 0);
368     analogWrite(ML2, 0);
369     analogWrite(MR1, 0);
370     analogWrite(MR2, 0);
371
372     // Set PWM frequency to ~8kHz for quieter motor operation (
373     // Timer 1 and Timer 0)
374     // TCCR1B = (TCCR1B & 0xF8) | 0x01; // Timer 1 prescaler 1
375     // TCCROB = (TCCROB & 0xF8) | 0x01; // Timer 0 prescaler 1
376
377     // Optimize ADC for faster sensor readings (prescaler 16)
378     ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16
379     // for ~77kHz sampling
380 }
381
382 // ----- CALIBRACIÓN DE SENSORES
383 -----
384
385 void calibrateSensors() {
386     digitalWrite(STATUS_LED, HIGH);
387     Serial.println("Calibrando sensores...");
388
389     for (int k = 0; k < CALIB_CYCLES; ++k) {
390         for (int i = 0; i < SENSOR_COUNT; ++i) {
391             int v = analogRead(SENSOR_PINS[i]);
392             if (v < minSensor[i]) minSensor[i] = v;
393             if (v > maxSensor[i]) maxSensor[i] = v;
394         }
395         if (k % 50 == 0) {
396             digitalWrite(STATUS_LED, !digitalRead(STATUS_LED));
397         }
398         delay(10);
399     }
400
401     for (int i = 0; i < SENSOR_COUNT; ++i) {
402         int span = maxSensor[i] - minSensor[i];
403         if (span < SENSOR_MIN_SPAN) {
404             sensorValid[i] = false;
405             gainSensor[i] = 0.0f;
406         } else {
407             sensorValid[i] = true;
408             gainSensor[i] = 1000.0f / (float)span;
409         }
410     }
411
412     digitalWrite(STATUS_LED, LOW);
413     // imprimire valores maximos y minimos
414     for (int i = 0; i < SENSOR_COUNT; ++i) {
415         Serial.print("Sensor ");
416         Serial.print(i);
417         Serial.print(" min: ");
418         Serial.print(minSensor[i]);

```

```

415     Serial.print(" max: ");
416     Serial.print(maxSensor[i]);
417     Serial.print(" gain: ");
418     Serial.println(gainSensor[i]);
419 }
420 Serial.println("Sensores calibrados.");
421 calibrated = true;
422 }
423
424 // ----- LEER POSICIÓN DE LÍNEA (ponderada)
425 -----
426 int readLinePosWeighted(bool debug) {
427     long weightedSum = 0;
428     long sum = 0;
429     if (debug) Serial.print("Lecturas: ");
430
431     for (int i = 0; i < SENSOR_COUNT; ++i) {
432         int raw = analogRead(SENSOR_PINS[i]);
433         if (!sensorValid[i]) continue;
434
435         if (debug) {
436             Serial.print(raw);
437             Serial.print(" ");
438         }
439
440         int val = (int)((float)raw - (float)minSensor[i]) *
441             gainSensor[i] + 0.5f);
442         if (debug) {
443             Serial.print(val);
444             Serial.print(" ");
445         }
446
447         if (val < 0) val = 0;
448         else if (val > 1000) val = 1000;
449
450         weightedSum += (long)val * (long)weights[i];
451         sum += val;
452     }
453
454     if (sum == 0) return 0;
455     int pos = (int)(weightedSum / sum);
456     if (debug) {
457         Serial.print("Pos: ");
458         Serial.println(pos);
459     }
460     return pos;
461 }
462 // ----- PID DE LÍNEA
463 -----
464 float pidLine(float reference, float error) {

```

```

463     lineErr = error;
464     float der = lineErr - linePrev;
465     lineInt += lineErr;
466
467     if (lineInt > LINE_INT_CLAMP) lineInt = LINE_INT_CLAMP;
468     else if (lineInt < -LINE_INT_CLAMP) lineInt = -LINE_INT_CLAMP
469         ;
470
471     float out = LKp * lineErr + LKi * lineInt + LKd * der;
472     linePrev = lineErr;
473     return out;
474 }
475 // -----
476 float pidSpeedL(float reference, float error) {
477     rpmErrL = error;
478     float der = rpmErrL - velPrevL;
479     velIntL += rpmErrL;
480
481     if (velIntL > VEL_INT_CLAMP) velIntL = VEL_INT_CLAMP;
482     else if (velIntL < -VEL_INT_CLAMP) velIntL = -VEL_INT_CLAMP;
483
484     float out = MKp_L * rpmErrL + MKi_L * velIntL + MKd_L * der;
485     velPrevL = rpmErrL;
486     return out;
487 }
488
489 // -----
490 float pidSpeedR(float reference, float error) {
491     rpmErrR = error;
492     float der = rpmErrR - velPrevR;
493     velIntr += rpmErrR;
494
495     if (velIntr > VEL_INT_CLAMP) velIntr = VEL_INT_CLAMP;
496     else if (velIntr < -VEL_INT_CLAMP) velIntr = -VEL_INT_CLAMP;
497
498     float out = MKp_R * rpmErrR + MKi_R * velIntr + MKd_R * der;
499     velPrevR = rpmErrR;
500     return out;
501 }
502
503 // -----
504 void setMotorsPWM(float leftPWM, float rightPWM) {
505     leftPWM = constrain(leftPWM, -PWM_MAX, PWM_MAX);
506     rightPWM = constrain(rightPWM, -PWM_MAX, PWM_MAX);
507
508     if (leftPWM >= 0) {
509         analogWrite(ML1, leftPWM);

```

```

510         analogWrite(ML2, 0);
511     } else {
512         analogWrite(ML1, 0);
513         analogWrite(ML2, -leftPWM);
514     }
515
516     if (rightPWM >= 0) {
517         analogWrite(MR1, rightPWM);
518         analogWrite(MR2, 0);
519     } else {
520         analogWrite(MR1, 0);
521         analogWrite(MR2, -rightPWM);
522     }
523 }
524
525 void resetPIDAndSpeeds() {
526     // reset speeds
527     currentRpmL = 0.0f;
528     currentRpmR = 0.0f;
529     targetRpmL = 0.0f;
530     targetRpmR = 0.0f;
531     pwmL = 0.0f;
532     pwmR = 0.0f;
533     lineOut = 0.0f;
534
535     // reset PID line
536     lineErr = 0.0f;
537     lineInt = 0.0f;
538     linePrev = 0.0f;
539
540     // reset PID speed L
541     rpmErrL = 0.0f;
542     velPrevL = 0.0f;
543     velIntL = 0.0f;
544
545     // reset PID speed R
546     rpmErrR = 0.0f;
547     velPrevR = 0.0f;
548     velIntr = 0.0f;
549
550     currentPos = 0.0f;
551 }
```

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

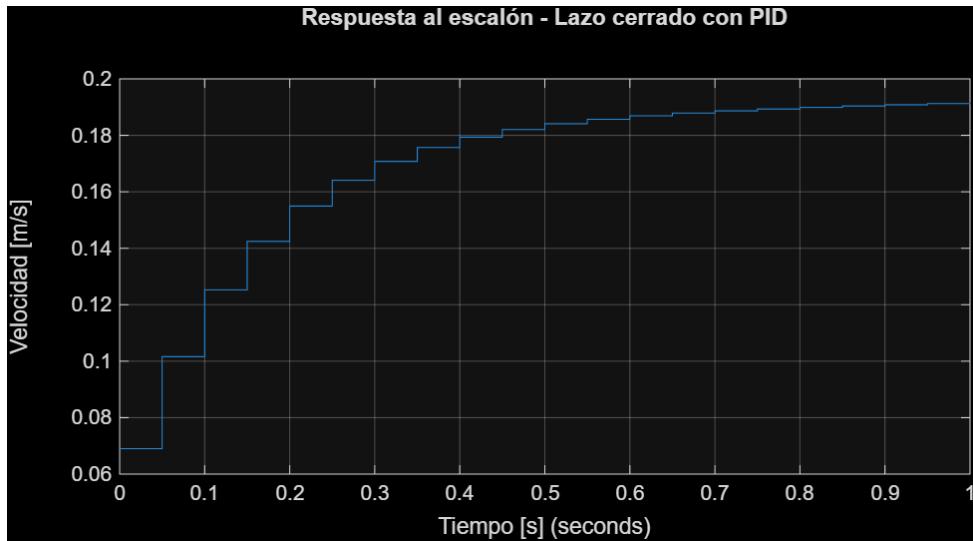


Figura 8: Respuesta comparativa: P, PI y PID

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.

```

1 % Análisis de Datos Reales del Sistema
2 clc; clear; close all;
3
4 % Leer datos del archivo CSV
5 data = readtable('data.csv');
6
7 % Extraer columnas
8 time = data.time / 1e6; % Convertir de microsegundos a segundos
9 pos = data.pos;
10 rpmL = data.rpmL;
11 rpmR = data.rpmR;
12 lineOut = data.lineOut;
13 pwmL = data.pwmL;
14 pwmR = data.pwmR;
15
16 % Ajustar tiempo relativo al inicio
17 time = time - time(1);
18
19 % Gráfica 1: Posición de la línea vs tiempo
20 figure;
21 plot(time, pos);
22 title('Posición de la Línea vs Tiempo');
23 xlabel('Tiempo [s]');
24 ylabel('Posición [unidades]');
25 grid on;
26
27 % Gráfica 2: RPM de motores izquierdo y derecho vs tiempo
28 figure;
29 plot(time, rpmL, 'b-', time, rpmR, 'r--');
30 title('RPM de Motores vs Tiempo');
31 xlabel('Tiempo [s]');
32 ylabel('RPM');
33 legend('Izquierdo', 'Derecho');
34 grid on;
35
36 % Gráfica 3: Salida del PID de línea vs tiempo
37 figure;
38 plot(time, lineOut);
39 title('Salida del PID de Línea vs Tiempo')
40 xlabel('Tiempo [s]');
41 ylabel('Salida PID');
42 grid on;
43
44 % Gráfica 4: PWM de motores vs tiempo
45 figure;
46 plot(time, pwmL, 'b-', time, pwmR, 'r--');
47 title('PWM de Motores vs Tiempo');
48 xlabel('Tiempo [s]');
49 ylabel('PWM');
50 legend('Izquierdo', 'Derecho');
51 grid on;

```

```

52
53 % Cálculos de desempeño
54 % Error de posición (asumiendo referencia 0)
55 error_pos = pos - 0;
56 mse_pos = mean(error_pos.^2);
57 fprintf('MSE de posición: %.4f\n', mse_pos);

58
59 % Variabilidad de RPM
60 std_rpmL = std(rpmL);
61 std_rpmR = std(rpmR);
62 fprintf('Desviación estandar RPM Izquierdo: %.4f\n', std_rpmL);
63 fprintf('Desviación estandar RPM Derecho: %.4f\n', std_rpmR);

64
65 % Energía PWM (aproximación)
66 energia_pwmL = sum(abs(pwmL)) / length(pwmL);
67 energia_pwmR = sum(abs(pwmR)) / length(pwmR);
68 fprintf('Energía promedio PWM Izquierdo: %.4f\n', energia_pwmL);
69 fprintf('Energía promedio PWM Derecho: %.4f\n', energia_pwmR);

70
71 % Validación con modelo
72 % Usar el modelo continuo para comparar
73 num = 1650;
74 den = [1 330 140];
75 G = tf(num, den);

76
77 % Simular respuesta con entrada PWM promedio
78 pwm_promedio = mean((pwmL + pwmR)/2);
79 t_sim = 0:0.01:max(time);
80 [y_sim, t_sim] = step(pwm_promedio * G, t_sim);

81
82 figure;
83 plot(time, (rpmL + rpmR)/2, 'b-', t_sim, y_sim, 'r--');
84 title('Comparación Modelos vs Datos Reales');
85 xlabel('Tiempo [s]');
86 ylabel('RPM Promedio');
87 legend('Datos Reales', 'Modelo Simulado');
88 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.

ANEXOS

1. **server.zip** – Proyecto completo:
 - **src/main.cpp** – Control en cascada PID con anti-windup
 - **platformio.ini** – configuración de placa y baudios
2. Hojas de datos técnicos: **datasheets.pdf**
3. Reporte fotográfico: **fotos.pdf**
4. Video de pruebas (2 min): **prueba_velocista.mp4**