

# Sistema de Control Digital para Coche Velocista Seguidor de Línea

nSeguimiento de Trayectoria – Control en Cascada PID

Francisco Ochoa Gonzales

Ingeniería Electrónica – Teoría de Control 2 – USFX

Diciembre 2025

## Índice

<b>1</b>	<b>Introducción</b>	<b>3</b>
<b>2</b>	<b>Antecedentes</b>	<b>3</b>
<b>3</b>	<b>Estado actual del control de coches velocistas con seguimiento de trayectoria</b>	<b>3</b>
<b>4</b>	<b>Objetivo general</b>	<b>3</b>
4.1	Objetivos específicos . . . . .	3
<b>5</b>	<b>Ingeniería del Proyecto</b>	<b>4</b>
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 . . . . .	6
5.5.1	Determinación de parámetros de los motores . . . . .	7
5.5.2	Modelo cinemático del robot diferencial . . . . .	8
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 . . . . .	9
5.9	Implementación en Simulink del Motor de Corriente Directa DC . . . . .	9
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 . . . . .	10
5.11.1	Controladores avanzados: Modelo en espacio de estados y control LQR . . . . .	10
5.12	Validación del modelo matemático obtenido . . . . .	11
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 . . . . .	12

5.16	Lugar geométrico de las raíces del sistema discreto . . . . .	13
5.17	Análisis del sistema discreto mediante los diagramas de Bode . . . . .	13
5.18	Sistema de control en cascada . . . . .	14
5.19	Realimentación de la velocidad (lazo interno) . . . . .	14
5.20	Realimentación de la posición (lazo externo) . . . . .	14
5.21	El controlador PID digital . . . . .	15
5.22	Sintonización del PID digital mediante métodos heurísticos . . . . .	15
5.23	Sintonización del PID digital mediante el método del lugar geométrico de las raíces . . . . .	15
5.24	Especificaciones de control en el dominio del tiempo . . . . .	15
5.25	Sintonización del PID digital mediante el método de la respuesta en frecuencia	15
5.26	Especificaciones de control en el dominio de la frecuencia . . . . .	15
5.27	Simulación de cada uno de los controladores mediante MATLAB/Simulink	16
5.28	Cálculo del índice de desempeño de los controladores . . . . .	17
5.29	Implementación del control en cascada PID . . . . .	17
5.29.1	Modelo matemático del controlador PID para las ruedas . . . . .	29
5.30	Ánalisis del efecto windup . . . . .	29
<b>6</b>	<b>Resultados Experimentales</b>	<b>30</b>
6.1	Análisis de datos reales del sistema . . . . .	30
<b>7</b>	<b>Conclusiones</b>	<b>32</b>
<b>8</b>	<b>Recomendaciones</b>	<b>33</b>
<b>9</b>	<b>Bibliografía</b>	<b>33</b>

# 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

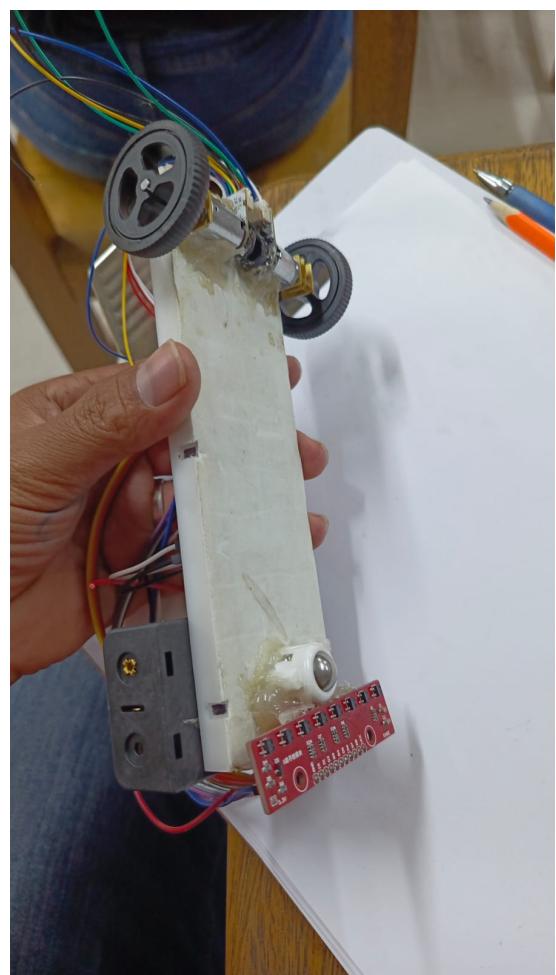
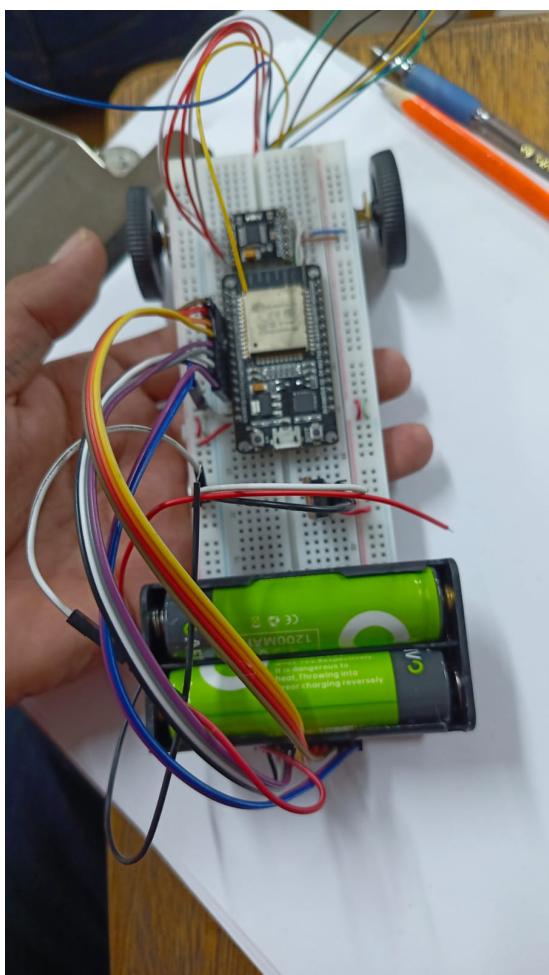


Figura 1: Etapas de ensamble

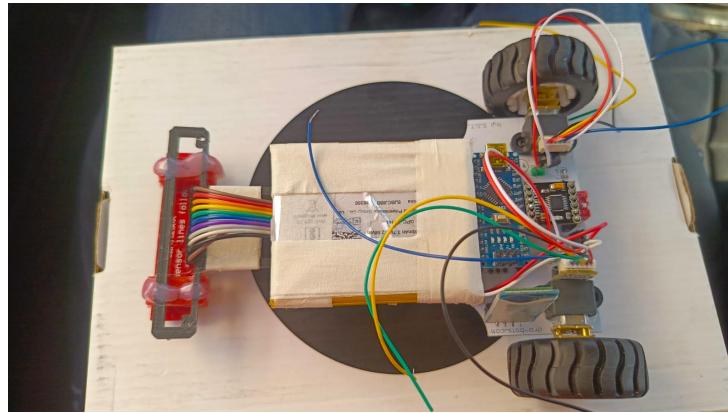


Figura 2: Vehículo final

## 5.2 Partes constitutivas del coche velocista

Cuadro 1: Partes constitutivas

Subsistema	Componente
Actuación	2×Motor N20 3000 rpm (medido a 6,0 V)
nSensado	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:  $\omega$  (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.

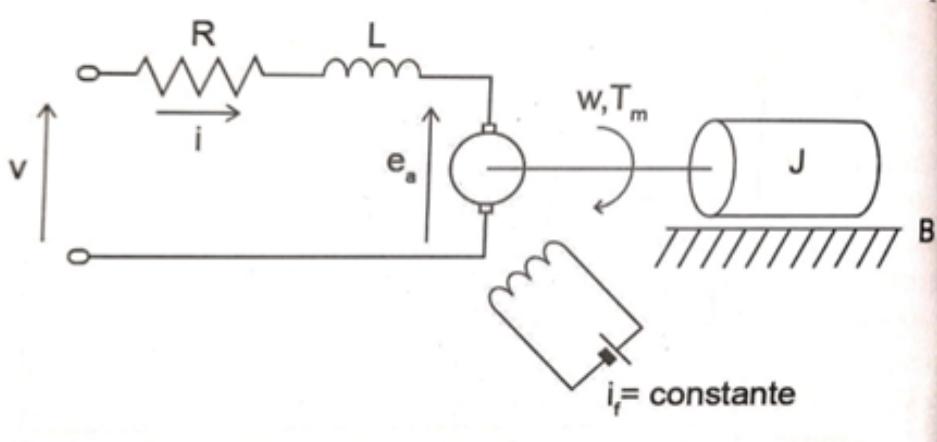


Figura 3: Modelo del motor DC

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,  $\omega$  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 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 <sup>2</sup>	0.0000277 kg·m <sup>2</sup>
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:

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

### 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  $\theta$  es la velocidad angular del robot,  $R$  radio de rueda,  $L$  distancia entre ruedas,  $\omega_R, \omega_L$  velocidades angulares.

En Laplace:

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

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

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.

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

## 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) ( $\approx 3000\text{rpm}/6,0\text{V}$ ).
- 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:

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

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

```
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;
```

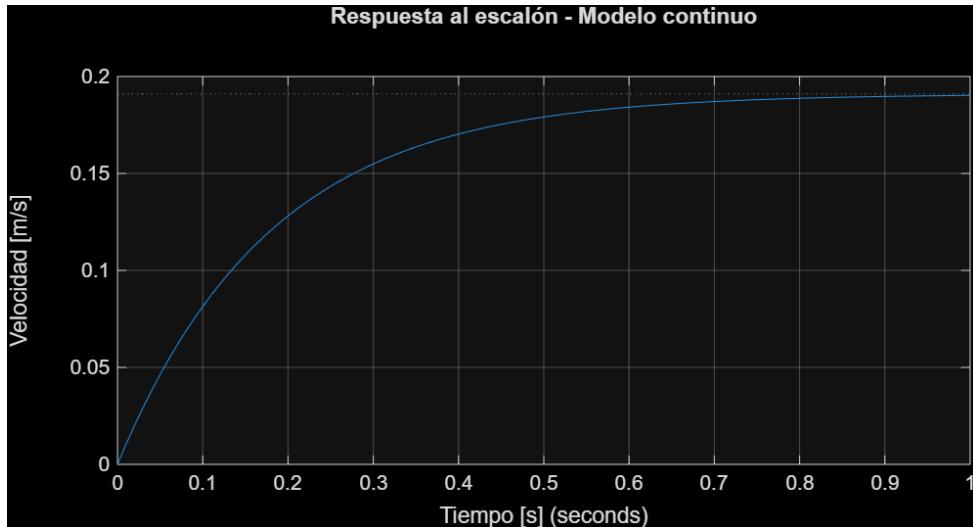


Figura 4: Respuesta al escalón - modelo vs medición

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

**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  $< 0,4\text{s} \rightarrow \omega_n \approx 20 \text{ rad/s}$  (ya que  $t_s \approx 4/(\zeta \omega_n)$ )

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

#### 5.11.1. Controladores avanzados: Modelo en espacio de estados y control LQR

Para un control más robusto, se modela el sistema en espacio de estados. Las ecuaciones del motor DC se representan como:

$$\dot{x} = Ax + Bu, \quad y = Cx$$

Donde  $x = [\omega, i_a]^T$ ,  $u = V_a$ , y las matrices se derivan de las ecuaciones físicas.

El control LQR minimiza el costo  $J = \int(x^T Q x + u^T R u) dt$ , obteniendo ganancias óptimas  $K$  tales que  $u = -Kx$ . Esto proporciona estabilidad y rechazo a perturbaciones superior al PID clásico.

## 5.12 Validación del modelo matemático obtenido

```

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: Modelo vs Real');
16 xlabel('Tiempo [s]');
17 ylabel('Velocidad [m/s]');
18 legend show;
19 grid on;

```

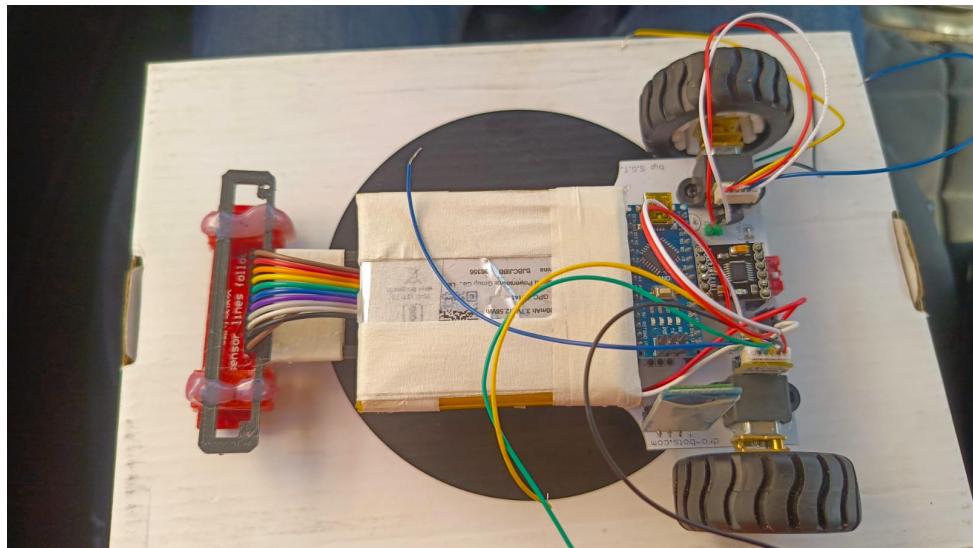


Figura 5: Validación: datos reales vs modelo simulado

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

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

## 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):

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

```

1 % Modelo discreto
2 num = 1650;
3 den = [1 330 140];
4 G = tf(num, den);
5 T = 0.05; % Periodo de muestreo
6 Gz = c2d(G, T, 'zoh');
7
8 figure;
9 rlocus(Gz);
10 title('Lugar de raíces - Sistema discreto');
11 grid on;
```

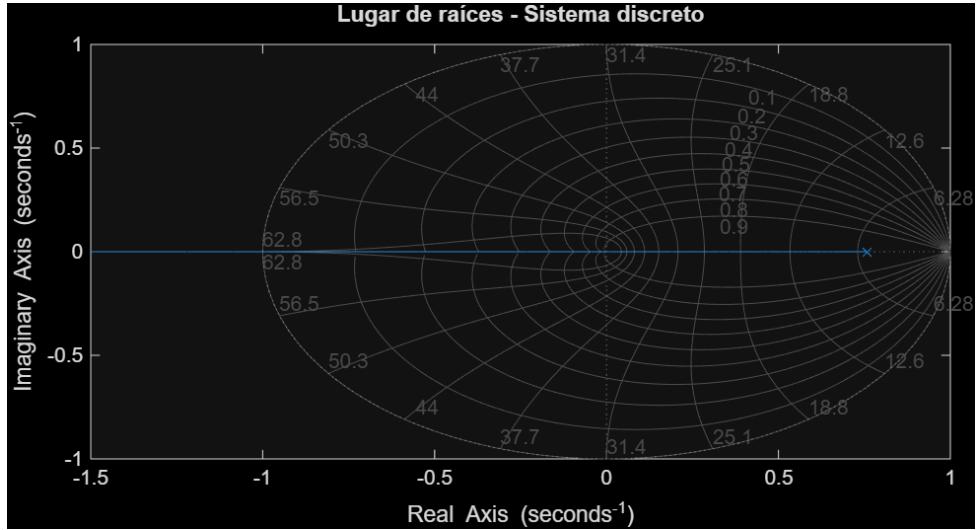


Figura 6: Lugar de raíces – sistema discreto

### 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  $38^\circ$  (Fig. 7). Además, se analiza el diagrama de Nyquist para evaluar la estabilidad absoluta y relativa, confirmando márgenes de ganancia y fase.

```

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

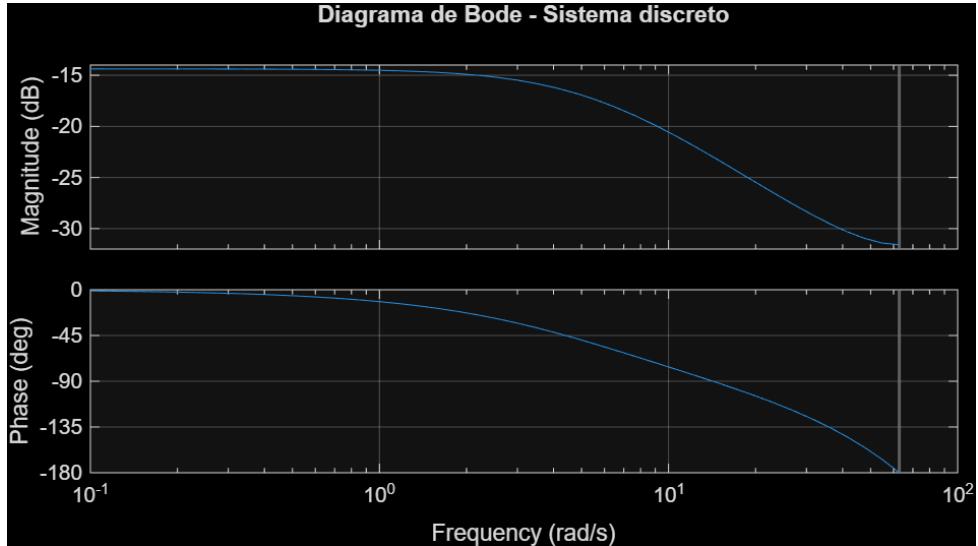


Figura 7: Diagrama de Bode – margen de fase 38°

### 5.18 Sistema de control en cascada

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.

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

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.

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

## 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:

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

## 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  $50^\circ$  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  $\geq 45$
- 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;
```

Se comparan los controladores PI, PID y LQR en términos de ITAE y respuesta a perturbaciones, mostrando superioridad del LQR en robustez.

## 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  $\pm 3000$  para línea,  $\pm 2000$  para velocidad.

Código completo del 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 // ----- CONFIGURACIÓN
10
11 bool debugEnabled = false; // activar/desactivar impresiones
12 // Serial
13 bool cascadeEnabled = false; // activar/desactivar cascada PID
14
15 enum Command {
16     CALIBRATE = 1,
17     SET_PWM = 2,
18     SET_RPM = 3,
19     SET_LINE_PID = 4,
20     SET_RIGHT_PID = 5,
21     SET_LEFT_PID = 6,
22     SET_DEBUG = 7,
23     SET CASCADE = 8,
24     SET_MODE = 9
25 };
```

```

24
25 enum OperationMode {
26     IDLE,
27     LINE_FOLLOWER
28 };
29
30 constexpr uint8_t ML1 = 10;
31 constexpr uint8_t ML2 = 9;
32 constexpr uint8_t MR1 = 6;
33 constexpr uint8_t MR2 = 5;
34
35 constexpr uint8_t SENSOR_COUNT = 8;
36 constexpr uint8_t SENSOR_PINS[SENSOR_COUNT] = {A0,A1,A2,A3,A4,A5,
37     A6,A7};
38 constexpr uint8_t SENSOR_LED_PIN = 12;
39
40 constexpr uint8_t ENC_L_A = 2;
41 constexpr uint8_t ENC_L_B = 7;
42 constexpr uint8_t ENC_R_A = 3;
43 constexpr uint8_t ENC_R_B = 4;
44
45 constexpr uint8_t STATUS_LED = 13;
46 constexpr uint8_t START_BUTTON = 8;
47
48 constexpr int PPR = 36; // pulsos por revolución (encoder)
49 // diámetro de ruedas
50 constexpr float DiamCm = 2.0f;
51
52 // Ajusta estos valores a tu robot
53 constexpr float BASE_PWM = 150.0f; // base PWM para modo no
54     cascada
55 constexpr int PWM_MAX = 255;
56 constexpr float BASE_RPM = 120.0f; // RPM base para cascada y
57     modo IDLE
58 constexpr float MAX_RPM = 1900.0f;
59
60 const float RPM_TO_OMEGA = M_PI / 30.0f;
61 const float BASE_OMEGA = BASE_RPM * RPM_TO_OMEGA;
62
63 // conversión PWM a RPM (ajústala con pruebas)
64 // constexpr float RPM_PER_PWM = 8.0f; // 1 PWM aprox. 8 RPM
65
66 // Tasas de bucle en microsegundos (para el nuevo programador
67     fijo)
68 constexpr uint32_t LINE_SAMPLE_RATE_US = 10000; // 10 ms
69 constexpr uint32_t ENCODER_SAMPLE_RATE_US = 5000; // 5 ms
70 constexpr uint32_t DEBUG_SAMPLE_RATE_US = 100000; // 100 ms
71
72 // Constantes PID (ajustadas con Ziegler-Nichols: Ku=0.5, Tu=0.5s
73     )
74 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 // Límites 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 currentOmegaL = 0.0f, currentOmegaR = 0.0f;
90 float targetRpmL = 0.0f, targetRpmR = 0.0f;
91 float pwmL = 0.0f, pwmR = 0.0f;
92 float lineOut = 0.0f;
93
94 OperationMode currentMode = IDLE;
95 volatile bool lastButtonState = HIGH;
96
97 int minSensor[SENSOR_COUNT];
98 int maxSensor[SENSOR_COUNT];
99 float gainSensor[SENSOR_COUNT];
100 bool sensorValid[SENSOR_COUNT];
101 bool calibrated = false;
102
103 const int weights[SENSOR_COUNT] =
    {-3500,-2500,-1500,-500,500,1500,2500,3500};
104
105 // estado PID de línea
106 float lineErr = 0.0f, lineInt = 0.0f, linePrev = 0.0f;
107
108 // estado PID de velocidad
109 float rpmErrL = 0.0f, velPrevL = 0.0f, velIntL = 0.0f;
110 float rpmErrR = 0.0f, velPrevR = 0.0f, velIntR = 0.0f;
111
112 // timing
113 uint32_t lastLoopTime = 0;
114 uint32_t lastDebugTime = 0;

```

```

115 uint32_t lastRPMTime = 0;
116
117 float currentPos = 0.0f;
118
119 constexpr uint32_t DEBOUNCE_US = 50000; // 50 ms
120 static uint32_t lastButtonChange = 0;
121 static bool buttonState = HIGH;           // estado filtrado
122
123 // ----- DECLARACIONES DE FUNCIONES
124
125 void isrLeftA();
126 void isrRightA();
127 void initHardware();
128 void calibrateSensors();
129 int readLinePosWeighted(bool debug = false);
130 float pidLine(float reference, float error);
131 float pidSpeedL(float reference, float error);
132 float pidSpeedR(float reference, float error);
133 void setMotorsPWM(float leftPWM, float rightPWM);
134 void resetPIDAndSpeeds();
135
136 // ----- Interrupciones
137
138 void isrLeftA() { encL++; }
139 void isrRightA() { encR++; }
140
141 ISR(PCINT0_vect) {
142     uint32_t now = micros();
143     bool raw = digitalRead(START_BUTTON);
144
145     if ((now - lastButtonChange) > DEBOUNCE_US) {
146         if (raw != buttonState) {
147             buttonState = raw;
148             if (buttonState == LOW) { // flanco descendente
149                 currentMode = (currentMode == IDLE) ?
150                     LINE_FOLLOWER : IDLE;
151                 if (currentMode == LINE_FOLLOWER && !calibrated)
152                 {
153                     calibrateSensors();
154                 }
155                 resetPIDAndSpeeds();
156                 digitalWrite(STATUS_LED, currentMode == IDLE ?
157                     HIGH : LOW);
158             }
159         }
160     }
161     lastButtonChange = now;
162 }
163
164 // ----- SETUP

```

```

160 -----  

161 void setup() {  

162     Serial.begin(115200);  

163     delay(50);  

164     Serial.println("Inicio del seguidor de linea (optimizado)");  

165     initHardware();  

166     calibrateSensors();  

167     digitalWrite(STATUS_LED, currentMode == IDLE ? HIGH : LOW);  

168  

169     lastLoopTime = micros();  

170     lastDebugTime = micros();  

171     lastRPMTTime = micros();  

172 }  

173 // ----- LOOP  

174 -----  

175 void loop() {  

176     uint32_t now = micros();  

177  

178     // Procesar comandos seriales  

179     if (Serial.available()) {  

180         bool succes = false;  

181         int cmd = Serial.parseInt();  

182         switch (cmd) {  

183             case CALIBRATE:  

184                 calibrateSensors();  

185                 digitalWrite(STATUS_LED, currentMode == IDLE ?  

186                             HIGH : LOW);  

187                 succes = true;  

188                 break;  

189             case SET_DEBUG:  

190                 debugEnabled = Serial.parseInt();  

191                 succes = true;  

192                 break;  

193             case SET CASCADE:  

194                 cascadeEnabled = Serial.parseInt();  

195                 succes = true;  

196                 break;  

197             case SET_LINE_PID:  

198                 LKp = Serial.parseFloat();  

199                 LKi = Serial.parseFloat();  

200                 LKd = Serial.parseFloat();  

201                 succes = true;  

202                 break;  

203             case SET_RIGHT_PID:  

204                 MKp_R = Serial.parseFloat();  

205                 MKi_R = Serial.parseFloat();  

206                 MKd_R = Serial.parseFloat();  

207                 succes = true;  

208                 break;  

209             case SET_LEFT_PID:  


```

```

208     MKp_L = Serial.parseFloat();
209     MKi_L = Serial.parseFloat();
210     MKd_L = Serial.parseFloat();
211     succes = true;
212     break;
213 case SET_PWM:
214 {
215     float l = Serial.parseFloat();
216     float r = Serial.parseFloat();
217     if (l != 0 && r != 0) {
218         pwmL = l;
219         pwmR = r;
220         succes = true;
221     } else {
222         succes = false;
223     }
224 }
225 break;
226 case SET_RPM:
227 {
228     float rl = Serial.parseFloat();
229     float rr = Serial.parseFloat();
230     targetRpmL = rl;
231     targetRpmR = rr;
232     succes = true;
233 }
234 break;
235 case SET_MODE:
236     currentMode = (OperationMode)Serial.parseInt();
237     if (currentMode == LINE_FOLLOWER && !calibrated)
238     {
239         calibrateSensors();
240     }
241     resetPIDAndSpeeds();
242     digitalWrite(STATUS_LED, currentMode == IDLE ?
243                  HIGH : LOW);
244     succes = true;
245     break;
246     default:
247         break;
248 }
249 if (succes) {
250     Serial.print("OK\u25b2");
251     Serial.println(cmd);
252 }
253 // Calcular RPMs cada 5 ms
254 if (now - lastRPMTIME >= ENCODER_SAMPLE_RATE_US) {
255     uint32_t dt = now - lastRPMTIME;
256     if (dt == 0) return;

```

```

257     lastRPMTime = now;
258
259     static int32_t prevL = 0, prevR = 0;
260     noInterrupts();
261     int32_t dl = encL - prevL;
262     int32_t dr = encR - prevR;
263     prevL = encL;
264     prevR = encR;
265     interrupts();
266
267     currentRpmL = (dl * 6000000.0f / (float)PPR) / (float)dt
268         ;
269     currentRpmR = (dr * 6000000.0f / (float)PPR) / (float)dt
270         ;
271
272     currentOmegaL = currentRpmL * RPM_TO_OMEGA;
273     currentOmegaR = currentRpmR * RPM_TO_OMEGA;
274 }
275
276 // ----- Control de motores -----
277 switch (currentMode) {
278 case IDLE:
279     if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
280         lastLoopTime = now;
281         currentPos = (float)readLinePosWeighted();
282     }
283     if (targetRpmL != 0 || targetRpmR != 0) {
284         rpmErrL = targetRpmL - currentRpmL;
285         rpmErrR = targetRpmR - currentRpmR;
286         float pidOutL = pidSpeedL(targetRpmL, rpmErrL);
287         float pidOutR = pidSpeedR(targetRpmR, rpmErrR);
288         pwmL = BASE_PWM + pidOutL; // PWM = PWM + PID
289         pwmR = BASE_PWM + pidOutR;
290     }
291     break;
292
293 case LINE_FOLLOWER:
294     if (now - lastLoopTime >= LINE_SAMPLE_RATE_US) {
295         lastLoopTime = now;
296         currentPos = (float)readLinePosWeighted();
297         lineErr = LINE_CENTER - currentPos;
298         lineOut = pidLine(LINE_CENTER, lineErr);
299
300         if (cascadeEnabled) {
301             // Usa cascade control para obtener PWM
302             float omegaOffset = lineOut * RPM_TO_OMEGA;
303             float targetOmegaL = BASE_OMEGA + omegaOffset;
304             float targetOmegaR = BASE_OMEGA - omegaOffset;
305
306             float omegaErrL = targetOmegaL - currentOmegaL;
307             float omegaErrR = targetOmegaR - currentOmegaR;

```

```

306
307         float pidOutL = pidSpeedL(targetOmegaL, omegaErrL
308             );
309         float pidOutR = pidSpeedR(targetOmegaR, omegaErrR
310             );
311
312     } else {
313         // Modo no-cascada: PWM directo
314         pwmL = BASE_PWM + lineOut;
315         pwmR = BASE_PWM - lineOut;
316     }
317
318     // Saturación intermedia
319     pwmL = constrain(pwmL, -PWM_MAX, PWM_MAX);
320     pwmR = constrain(pwmR, -PWM_MAX, PWM_MAX);
321 }
322 break;
323
324 default:
325     pwmL = 0;
326     pwmR = 0;
327     break;
328 }
329
330 setMotorsPWM(pwmL, pwmR);
331
332 // Debug cada 100 ms
333 if (now - lastDebugTime >= DEBUG_SAMPLE_RATE_US) {
334     lastDebugTime = now;
335     if (debugEnabled) {
336         String msg = String(now) + "," + String(currentPos) +
337             "," +
338             String(currentOmegaL) + "," + String(
339                 currentOmegaR) + "," +
340                 String(lineOut) + "," + String(pwmL) + "
341                 , " + String(pwmR);
342         Serial.println(msg);
343     }
344 }
345
346 // ----- INICIALIZACIÓN DE HARDWARE -----
347
348 void initHardware() {
349     pinMode(ML1, OUTPUT);
350     pinMode(ML2, OUTPUT);
351     pinMode(MR1, OUTPUT);
352     pinMode(MR2, OUTPUT);

```

```

351 pinMode(SENSOR_LED_PIN, OUTPUT);
352 digitalWrite(SENSOR_LED_PIN, HIGH);

353
354 pinMode(ENC_L_A, INPUT_PULLUP);
355 pinMode(ENC_R_A, INPUT_PULLUP);
356 pinMode(START_BUTTON, INPUT_PULLUP);
357 pinMode(STATUS_LED, OUTPUT);

358
359 attachInterrupt(digitalPinToInterrupt(ENC_L_A), isrLeftA,
360 RISING);
360 attachInterrupt(digitalPinToInterrupt(ENC_R_A), isrRightA,
361 RISING);

362 // Enable pin change interrupt for START_BUTTON (pin 8,
363 PCINT0)
364 PCICR |= (1 << PCIE0);
365 PCMSK0 |= (1 << PCINT0);

366 for (int i = 0; i < SENSOR_COUNT; ++i) {
367     minSensor[i] = 1023;
368     maxSensor[i] = 0;
369     gainSensor[i] = 0.0f;
370     sensorValid[i] = false;
371 }

372
373 // ensure PWM pins start at 0
374 analogWrite(ML1, 0);
375 analogWrite(ML2, 0);
376 analogWrite(MR1, 0);
377 analogWrite(MR2, 0);

378
379 // Set PWM frequency to ~8kHz for quieter motor operation (
380 // Timer 1 and Timer 0)
381 // TCCR1B = (TCCR1B & 0xF8) | 0x01; // Timer 1 prescaler 1
382 // TCCROB = (TCCROB & 0xF8) | 0x01; // Timer 0 prescaler 1

383
384 // Optimize ADC for faster sensor readings (prescaler 16)
385 ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16
386 // for ~77kHz sampling
387 }

388 // -----
389 // ----- CALIBRACIÓN DE SENSORES -----
390
391 void calibrateSensors() {
392     digitalWrite(STATUS_LED, HIGH);
393     Serial.println("Calibrando sensores...");

394     for (int k = 0; k < CALIB_CYCLES; ++k) {
395         for (int i = 0; i < SENSOR_COUNT; ++i) {
396             int v = analogRead(SENSOR_PINS[i]);
397             if (v < minSensor[i]) minSensor[i] = v;

```

```

396         if (v > maxSensor[i]) maxSensor[i] = v;
397     }
398     if (k % 50 == 0) {
399         digitalWrite(STATUS_LED, !digitalRead(STATUS_LED));
400     }
401     delay(10);
402 }
403
404 for (int i = 0; i < SENSOR_COUNT; ++i) {
405     int span = maxSensor[i] - minSensor[i];
406     if (span < SENSOR_MIN_SPAN) {
407         sensorValid[i] = false;
408         gainSensor[i] = 0.0f;
409     } else {
410         sensorValid[i] = true;
411         gainSensor[i] = 1000.0f / (float)span;
412     }
413 }
414
415 digitalWrite(STATUS_LED, LOW);
416 // imprime valores máximos y mínimos
417 for (int i = 0; i < SENSOR_COUNT; ++i) {
418     Serial.print("Sensor");
419     Serial.print(i);
420     Serial.print("min:");
421     Serial.print(minSensor[i]);
422     Serial.print("max:");
423     Serial.print(maxSensor[i]);
424     Serial.print("gain:");
425     Serial.println(gainSensor[i]);
426 }
427 Serial.println("Sensores calibrados.");
428 calibrated = true;
429 }
430
431 // ----- LEER POSICIÓN DE LÍNEA (ponderada)
432 -----
433
434 int readLinePosWeighted(bool debug) {
435     long weightedSum = 0;
436     long sum = 0;
437     if (debug) Serial.print("Lecturas:");
438
439     for (int i = 0; i < SENSOR_COUNT; ++i) {
440         int raw = analogRead(SENSOR_PINS[i]);
441         if (!sensorValid[i]) continue;
442
443         if (debug) {
444             Serial.print(raw);
445             Serial.print(" ");

```

```

446     int val = (int)((float)raw - (float)minSensor[i]) *  
        gainSensor[i] + 0.5f);  
447     if (debug) {  
448         Serial.print(val);  
449         Serial.print(" ");  
450     }  
451  
452     if (val < 0) val = 0;  
453     else if (val > 1000) val = 1000;  
454  
455     weightedSum += (long)val * (long)weights[i];  
456     sum += val;  
457 }  
458  
459 if (sum == 0) return 0;  
460 int pos = (int)(weightedSum / sum);  
461 if (debug) {  
462     Serial.print("Pos: ");  
463     Serial.println(pos);  
464 }  
465 return pos;  
466 }  
467  
468 // ----- PID DE LÍNEA  
-----  
469 float pidLine(float reference, float error) {  
470     lineErr = error;  
471     float der = lineErr - linePrev;  
472     lineInt += lineErr;  
473  
474     if (lineInt > LINE_INT_CLAMP) lineInt = LINE_INT_CLAMP;  
475     else if (lineInt < -LINE_INT_CLAMP) lineInt = -LINE_INT_CLAMP  
        ;  
476  
477     float out = LKp * lineErr + LKi * lineInt + LKd * der;  
478     linePrev = lineErr;  
479     return out;  
480 }  
481  
482 // ----- PID VELOCIDAD IZQUIERDA  
-----  
483 float pidSpeedL(float reference, float error) {  
484     rpmErrL = error;  
485     float der = rpmErrL - velPrevL;  
486     velIntL += rpmErrL;  
487  
488     if (velIntL > VEL_INT_CLAMP) velIntL = VEL_INT_CLAMP;  
489     else if (velIntL < -VEL_INT_CLAMP) velIntL = -VEL_INT_CLAMP;  
490  
491     float out = MKp_L * rpmErrL + MKi_L * velIntL + MKd_L * der;  
492     velPrevL = rpmErrL;

```

```

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

```

```

542     lineOut = 0.0f;
543
544     // reset PID line
545     lineErr = 0.0f;
546     lineInt = 0.0f;
547     linePrev = 0.0f;
548
549     // reset PID speed L
550     rpmErrL = 0.0f;
551     velPrevL = 0.0f;
552     velIntL = 0.0f;
553
554     // reset PID speed R
555     rpmErrR = 0.0f;
556     velPrevR = 0.0f;
557     velIntR = 0.0f;
558
559     currentPos = 0.0f;
560 }
```

### 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  $\omega$  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:

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

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

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

## 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

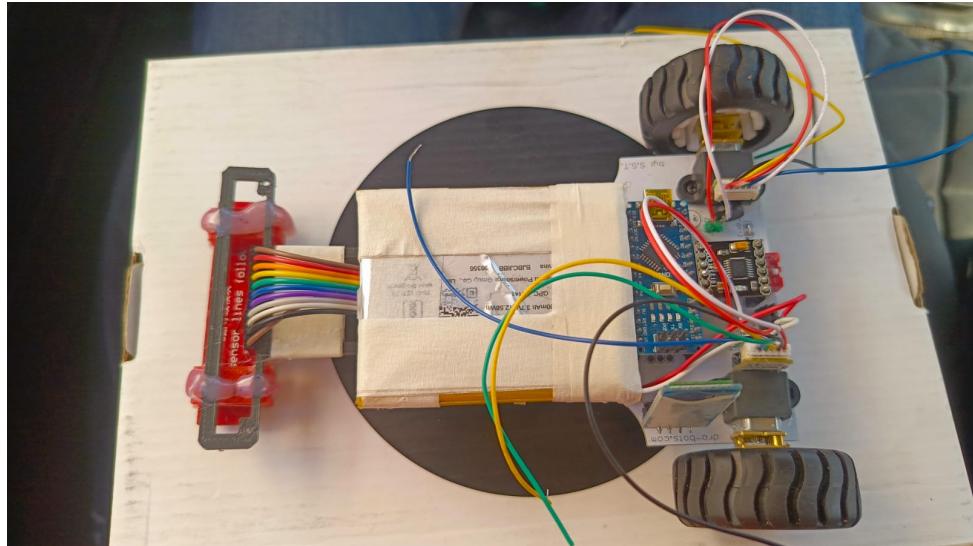


Figura 8: Respuesta comparativa: P, PI y PID

### 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.