Práctica Profesional Supervisada

Julio, Emmanuel Carlos

Plataforma autónoma de movilidad y procesamiento de análisis Pampa

Instituto de Ingeniería y Agronomía

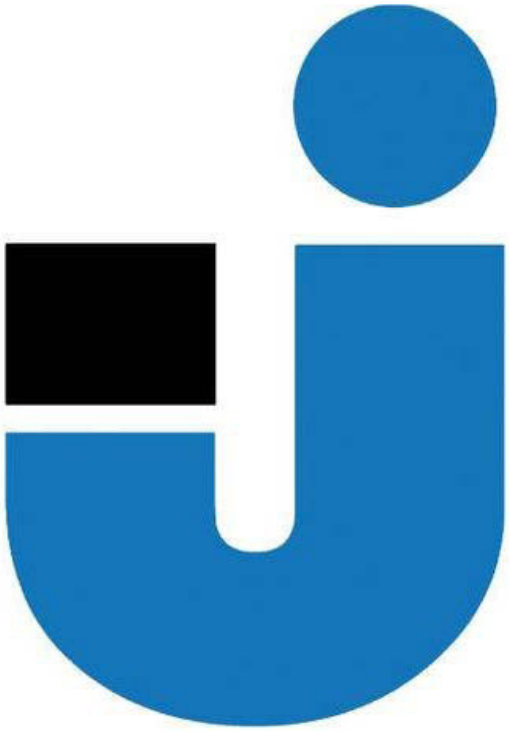
Carrera: Ingeniería en Informática

2025

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Carrera de Ingeniería en Informática



PRÁCTICA PROFESIONAL SUPERVISADA

Informe final

Plataforma autónoma de movilidad y procesamiento de análisis Pampa

Emmanuel Carlos Julio

Florencio varela Julio, 2025

ESTUDIANTE

Apellido y Nombres: *Emmanuel Carlos Julio*

Correo electrónico: Emajulio.ej@gmail.com

ORGANIZACIÓN DONDE SE REALIZA LA PRÁCTICA PROFESIONAL SUPERVISADA

Nombre de la institución: *Universidad Nacional Arturo Jauretche*

Dirección: *Av. Calchaquí 6200, Florencio Varela, (1888) Buenos Aires, Argentina*

Teléfono: *+54 11 4275-6100*

Sector: *Programa Tecnologías de la Información y la Comunicación (TIC) en aplicaciones de interés social, Instituto de Ingeniería y Agronomía*

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: *Dr. Ing. Martín Morales*

Correo electrónico: *martin.morales@unaj.edu.ar*

DOCENTE SUPERVISOR

Apellido y Nombres: *Prof. Mg. OSIO, Jorge*

Correo electrónico: *josio@unaj.edu.ar*

COORDINADOR DE LA CARRERA DE INGENIERÍA INFORMÁTICA

Apellido y Nombres: *Dr. Ing. Morales, Martin*

Correo electrónico: *martin.morales@unaj.edu.ar*

Resumen

El proyecto *Pampa* nace con el objetivo de construir un sistema terrestre autónomo altamente versátil y modular, capaz de adaptarse a distintos escenarios de navegación y control sin depender exclusivamente de una estrategia fija. Su arquitectura permite cambiar dinámicamente entre modos de operación como control directo, evasión de obstáculos o desplazamiento planificado, lo que lo convierte en una plataforma ideal para experimentar con distintos enfoques de movimiento. Esta flexibilidad no solo potencia su utilidad en campo, sino que abre la puerta a investigar a fondo la integración y coordinación entre sensores, motores y lógica de control. Cada componente fue elegido y dispuesto para maximizar la interoperabilidad futura, permitiendo ampliaciones como nuevas estrategias, sensores adicionales o protocolos de comunicación sin rediseñar todo el sistema. La decisión de investigar esta integración no responde solo a un objetivo funcional, sino a una visión a largo plazo: crear una base tecnológica robusta, capaz de evolucionar hacia soluciones autónomas más complejas con mínimo esfuerzo de reconfiguración.

Palabras Clave: Robótica autónoma, Versatilidad operativa, Estrategias de navegación dinámicas, Plataforma modular, Control adaptable, Integración de sensores y motores, Arquitectura escalable, Robótica experimental, Expansión futura, Control inteligente

Abstract

The *Pampa* project was created with the goal of building a highly versatile and modular autonomous ground system capable of adapting to different navigation and control scenarios without relying on a fixed strategy. Its architecture enables seamless switching between operational modes such as direct control, obstacle avoidance, or planned movement, making it an ideal platform for experimenting with various motion approaches. This flexibility not only enhances its field utility but also opens the door to in-depth research on the integration and coordination of sensors, motors, and control logic. Each component was selected and arranged to maximize future interoperability, allowing for the addition of new strategies, sensors, or communication protocols without the need for a complete system redesign. The decision to explore this integration is not only functional but also rooted in a long-term vision: to build a robust technological foundation capable of evolving into more complex autonomous solutions with minimal reconfiguration effort.

Keywords: Autonomous robotics, Operational versatility, Dynamic navigation strategies, Modular platform, Adaptive control, Sensor and motor integration, Scalable architecture, Experimental robotics, Future expansion, Intelligent control

Dedicatorias y Agradecimientos

A mis profesores de la Universidad Nacional Arturo Jauretche y a quienes tuve el honor de acompañar como alumno en la ETT N.º 1 “Martín Miguel de Güemes”, donde me formé como Técnico en Electrónica. Sin el acompañamiento, la dedicación y los valiosos consejos de todos ellos, no contaría con los conocimientos ni las herramientas necesarias para llevar adelante este proyecto en su totalidad, desde el inicio hasta su finalización.

1. Índice General

Índice de figuras

2. Marco teórico

El proyecto es la integración de varias áreas de conocimiento que deben tratarse por separado y en los próximos pasos serán detallados. En conjunto brindan la posibilidad de mejorar la información que se tiene a la hora de tomar decisiones en el área agrícola.
El estudio de los conceptos como “Métricas” establece que nada que no sea medido puede mejorarse, y tomando esto como premisa podemos ahondar que si se tuvieran más datos sobre los niveles de los componentes que definen al suelo como fértil podríamos encontrar mejores correlaciones entre diferentes factores y ver como estos, en conjunto , pueden ser indicativos de determinadas producciones pudiendo ,en caso de ser necesario, realizar determinadas correcciones para cambiar un resultado esperado.
El campo de la automatización se volvió muy útil y el hecho de que que una tarea repetitiva pueda hacerse siempre exactamente de la misma manera puede favorecer ampliamente el resultado de los análisis.
El objetivo del proyecto es la creación de un rover capas de traquear determinada área de forma autómata con configuraciones mínimas (que serán establecidas a lo largo del documento) y a medida que este avance cada x cantidad de metros sea capaz de tomar una muestra del material orgánico de interés (medir las propiedades del suelo) junto con otros datos como temperatura y coordenadas. Estos datos deben ser enviados juntos con la fecha y hora mediante una comunicación inalámbrica y deben ser cifrados por el propio dispositivo para que sean almacenados en un servidor.
Se espera poder analizar a largo plazo la correlación de los niveles de nitrato potasio ,sodio ,humedad y temperatura con la producción y realizar inferencias mediante el estudio de los datos recolectados.

2.2. Interesados

Los interesados en el desarrollo del proyecto Pampa abarcan un espectro amplio, tanto del ámbito académico como del productivo. En primer lugar, instituciones educativas y docentes comprometidos con la innovación tecnológica pueden ver en este proyecto una oportunidad concreta para fomentar el aprendizaje basado en proyectos reales, con integración multidisciplinaria entre electrónica, software y mecánica. Además, productores agrícolas, especialmente aquellos con parcelas medianas o pequeñas, podrían beneficiarse directamente de un sistema autónomo adaptable como Pampa para tareas de recolección de datos, diagnóstico de cultivos o incluso supervisión remota.

Su arquitectura modular y su capacidad para operar con diversas estrategias de movimiento le otorgan al proyecto una gran proyección futura. Puede evolucionar hacia aplicaciones más complejas, como patrullaje en seguridad rural, inspección de instalaciones, o incluso convertirse en una plataforma de prueba para nuevas tecnologías de sensores, algoritmos de IA embarcada o redes de IoT colaborativas. Esta versatilidad lo hace especialmente atractivo para

empresas de tecnología, desarrolladores de soluciones agrícolas inteligentes y centros de investigación que buscan una base flexible para probar sus desarrollos en condiciones reales.

2.3. Proyectos similares en otras partes del mundo

El uso de robots móviles y sistemas de Internet de las Cosas (IoT) ha transformado múltiples industrias, más allá del ámbito agrícola. Estas tecnologías permiten mejorar la eficiencia operativa, reducir costos y recopilar datos de forma precisa en entornos que antes requerían intervención humana directa. A continuación, se describen desarrollos relevantes tanto en agricultura como en otros sectores industriales:

2.3.1. Agricultura inteligente

- Robots autodirigidos: Tractores y plataformas móviles equipadas con módulos de IA y visión artificial capaces de realizar labores como siembra, detección de plagas, desmalezado y cosecha automatizada.
- Cosechadores autónomos: Utilizan visión computacional y actuadores de precisión para recolectar frutos (como fresas o manzanas) sin dañarlos, optimizando la producción en invernaderos y cultivos extensivos.
- Robots pulverizadores y drones de supervisión: Automatizan la aplicación de agroquímicos y permiten monitorear el estado del cultivo en tiempo real a través de imágenes aéreas y sensores climáticos.

2.3.2. Minería y exploración

- Rovers terrestres autónomos: Utilizados en minería para explorar terrenos hostiles, recolectar muestras y mapear con precisión sin exponer personas a ambientes de riesgo.
- IoT subterráneo: Equipos con sensores que reportan vibraciones, gases tóxicos y humedad para prevenir accidentes y optimizar los procesos de excavación.

2.3.3. Logística y manufactura

- Vehículos Autónomos Móviles (AMR): Robots capaces de moverse de forma independiente en almacenes, adaptando rutas en tiempo real y evitando obstáculos. Son ampliamente utilizados por empresas como Amazon, DHL y fabricantes automotrices.
- Sistemas de inspección autónomos: Robots que recorren instalaciones industriales para controlar temperatura, vibraciones o fugas en tiempo real, reduciendo el tiempo de parada de líneas de producción.

2.3.4. Salud y asistencia

- Robots de monitoreo en hospitales: Equipados con cámaras, sensores de temperatura y comunicación inalámbrica, permiten vigilar pacientes, entregar medicamentos o desinfectar áreas sin intervención humana.

- Plataformas de asistencia en rehabilitación: Robots móviles que asisten en ejercicios terapéuticos controlados o en el traslado autónomo de pacientes dentro de instituciones médicas.

2.4. Enfoque del proyecto Pampa

El proyecto se centra dentro de la categoría de robots terrestres autodirigidos, específicamente en contextos agrícolas, pero con una arquitectura flexible que permite su extensión a otros usos como supervisión de instalaciones, vigilancia ambiental o exploración en entornos cerrados. Al diseñarse como una plataforma modular, resiliente y con capacidades de navegación autónoma, el proyecto Pampa no solo contribuye al sector agropecuario, sino que también sienta bases para futuras aplicaciones en otras industrias, como minería de superficie, monitoreo ambiental y control logístico en terrenos complejos.

3. Diseño del hardware

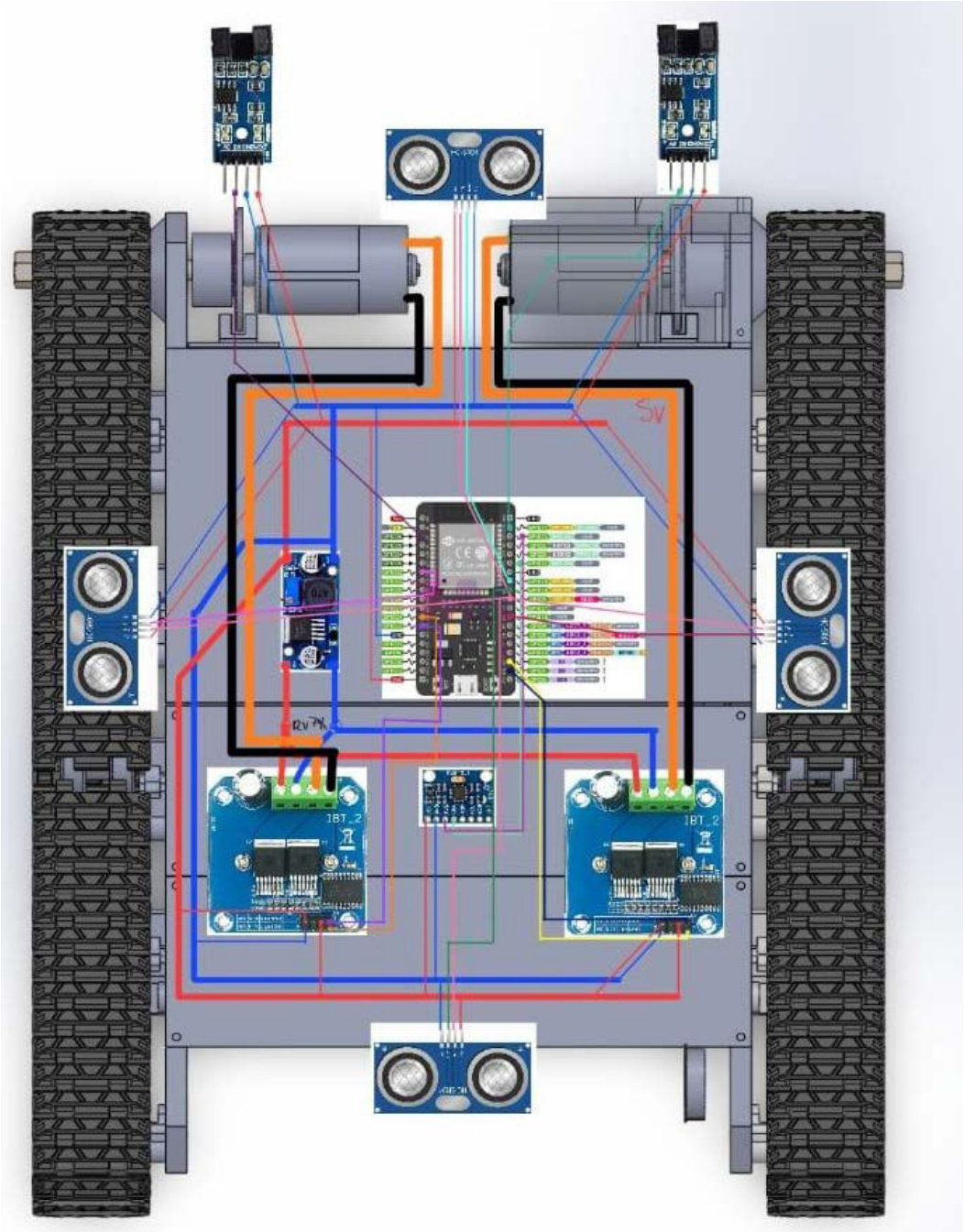


Figura 3 - Diagrama de conexión electrónica del sistema (vista superior).
Fuente: Propia

El proceso de diseño del sistema Pampa implicó múltiples iteraciones y ajustes sobre los componentes físicos, motivados por la necesidad de alcanzar parámetros de desempeño mínimos en cuanto a velocidad, robustez y eficiencia energética. A diferencia de enfoques

lineales, el desarrollo del hardware en este proyecto adoptó una lógica incremental, basada en pruebas reales y retroalimentación directa de campo.

Inicialmente, se emplearon moto-reductores genéricos ampliamente utilizados en proyectos de robótica educativa con Arduino, combinados con un puente H **L298N**. Si bien esta configuración resultó funcional en una etapa temprana, rápidamente se detectó que no alcanzaba la velocidad mínima requerida para garantizar un desplazamiento eficiente en terrenos irregulares.

Ante esta limitación, se procedió a rediseñar el chasis del sistema para incorporar **motores de corriente continua de mayor tamaño y par**, lo que derivó en una nueva problemática: el L298N no era capaz de entregar la corriente necesaria para su funcionamiento continuo sin riesgo de sobrecalentamiento o caída de tensión. Esto motivó la migración hacia un **driver BTS7960**, con capacidad de manejo de mayor corriente y eficiencia superior.

No obstante, al realizar las primeras pruebas con esta nueva configuración, se observó que el sistema no lograba avanzar adecuadamente, lo cual llevó a una última revisión crítica de la selección de motores. Finalmente, se optó por motores de **engranaje helicoidal DC 12V/470 RPM**, cuya relación entre torque, velocidad y eficiencia permitió cumplir con los requisitos operativos previstos en el diseño.

3.1 Selección de componentes

Si bien el proyecto atravesó diversas iteraciones durante su desarrollo, con pruebas de múltiples configuraciones y reemplazo progresivo de componentes, en esta sección se presentarán exclusivamente aquellos elementos que conforman la **versión final funcional del sistema**. Esta delimitación responde a la necesidad de centrar el análisis técnico en los resultados obtenidos a partir de decisiones validadas experimentalmente, dejando registro únicamente de los componentes que fueron incorporados en la arquitectura definitiva del robot Pampa.

3.1.1. ESP32 DOIT DevKit v1

Descripción: Microcontrolador de arquitectura dual-core con conectividad WiFi y Bluetooth integrada, con soporte para FreeRTOS y múltiples periféricos.
Criterio de selección: Se seleccionó por su capacidad de procesamiento paralelo, conectividad inalámbrica nativa y compatibilidad con bibliotecas modernas como ESPAsyncWebServer, esenciales para tareas de control en tiempo real y comunicación web.

3.1.2. Motores DC con engranaje helicoidal 12V / 470 RPM

Descripción: Motores de corriente continua con caja reductora helicoidal integrada, diseñados para entregar torque elevado a velocidades moderadas.
Criterio de selección: Fueron seleccionados tras varias iteraciones, por cumplir con los

requerimientos de velocidad mínima, eficiencia mecánica y capacidad de carga, que no se lograban con moto-reductores genéricos utilizados en robótica educativa.

3.1.3. Driver BTS7960

Descripción: Módulo puente H de alta potencia basado en MOSFETs, capaz de manejar corrientes de hasta 43A con protección térmica y por sobrecarga.
Criterio de selección: Se adoptó en reemplazo del L298N debido a su superior capacidad de corriente, menor pérdida de potencia y mayor confiabilidad al trabajar con motores de mayor demanda.

3.1.4. Tacómetros ópticos (encoders)

Descripción: Sensores que detectan pulsos generados por un disco ranurado, utilizados para calcular velocidad y distancia de desplazamiento.
Criterio de selección: Permiten implementar retroalimentación para control de velocidad y estimación de distancia recorrida, elementos clave en la navegación autónoma.

3.1.5. MPU6050 (Giroscopio + Acelerómetro)

Descripción: Sensor de seis ejes con comunicación I2C, utilizado para medir orientación angular y aceleraciones.
Criterio de selección: Elegido por su bajo consumo, amplia adopción en sistemas embebidos y compatibilidad con el ESP32, lo cual facilita la integración para estrategias de navegación con corrección angular.

3.1.6. Sensores ultrasónicos HC-SR04

Descripción: Dispositivos que miden distancia a través de pulsos de sonido, con un rango efectivo de 2 a 400 cm.
Criterio de selección: Incorporados por su bajo costo, simplicidad de implementación y efectividad para detección de obstáculos en entornos no estructurados.

3.1.7. JoystickPS3 (Bluetooth)

Descripción: Controlador inalámbrico utilizado para la modalidad de control manual del robot.
Criterio de selección: Seleccionado por su estabilidad en la comunicación Bluetooth con el ESP32, ergonomía y capacidad de lectura analógica de ejes para control de velocidad variable.

3.2 Criterio de elección de motores, controladores, sensores y microcontroladores.

El proceso de selección de componentes en el sistema Pampa no respondió a un único esquema predefinido, sino que se construyó progresivamente a partir de ensayos, validaciones empíricas y limitaciones impuestas por el entorno de uso. Esta sección detalla cronológicamente las

decisiones técnicas adoptadas en cuanto a los actuadores, sensores y controladores, explicando los fundamentos que llevaron a la arquitectura final.

3.2.1. Fase inicial: motores paso a paso y descartes energéticos

El primer enfoque consistió en utilizar motores **NEMA 17** controlados mediante **drivers Pololu A4988**, dada su disponibilidad y precisión. Sin embargo, rápidamente se detectó que el consumo energético era incompatible con el perfil de autonomía requerido por el robot. Alimentado por una batería de 12V y apenas 2Ah, el sistema sufría caídas de tensión y una autonomía reducida. Esto llevó a reconsiderar el tipo de motor, priorizando eficiencia energética por sobre precisión posicional.

3.2.2. Transición a motores DC y necesidad de retroalimentación

Se optó por motores de corriente continua, más eficientes desde el punto de vista energético y con mejor respuesta para tareas de tracción en el terreno. Esta transición exigió incorporar un sistema de medición de velocidad para mantener control sobre la locomoción. Se implementó entonces un tacómetro óptico con un disco de ranuras (inicialmente básico), que fue refinado hasta alcanzar un modelo de 24 ranuras, permitiendo una resolución adecuada para estimar la velocidad angular.

La velocidad angular **ω** se relaciona con la cantidad de pulsos por segundo **N** mediante la siguiente fórmula:

$$\omega = \frac{2\pi \cdot N}{n}$$

donde nnn es el número de ranuras del disco (en este caso, 24). Esta velocidad angular puede ser convertida a lineal según el radio de las ruedas.

3.2.3. Limitaciones de torque y rediseño de motorización

A pesar de esta mejora, los motorreductores genéricos inicialmente utilizados eran incapaces de alcanzar la velocidad mínima deseada de aproximadamente 2 cm/s, y carecían del torque necesario para movilizar la estructura de 2 kg en superficies irregulares. Se intentó entonces emplear motores de alto par, pero la corriente demandada supera la capacidad del L298N, el cual mostraba sobrecalentamiento y caídas de rendimiento. Esto llevó al reemplazo por el BTS7960, capaz de suministrar corrientes mayores con menor disipación de potencia.

No obstante, incluso con el nuevo puente H, los motores aún no lograban impulsar el sistema con fluidez. La solución definitiva consistió en incorporar motores de engranaje helicoidal DC 12V / 470 RPM, los cuales ofrecieron la combinación adecuada entre torque sostenido, velocidad lineal y consumo eficiente.

3.2.4. Sensado de orientación: incorporación del giroscopio

A fin de otorgar al sistema una referencia confiable de orientación, especialmente en estrategias autónomas, se integró un sensor MPU6050, el cual proporciona datos sobre la rotación en el eje Z (yaw) y, eventualmente, información sobre inclinaciones en pitch y roll. Esto posibilitó mejorar la navegación y abrir la puerta a futuras implementaciones de control en entornos con pendiente o inestabilidad.

3.2.5. Mejoras en alimentación

Como parte de las adaptaciones al consumo creciente de los motores seleccionados, se reemplazó la batería inicial por una de **12V y 7Ah**, garantizando así una mayor autonomía y estabilidad energética durante ciclos prolongados de operación.

3.2.6. Selección del microcontrolador: ESP32 DOIT DevKit v1

El microcontrolador seleccionado para el sistema Pampa fue el ESP32 DOIT DevKit v1, una placa de desarrollo basada en el SoC ESP32-WROOM-32 fabricado por Espressif. La elección de este microcontrolador se fundamentó en su equilibrio entre potencia de procesamiento, bajo consumo energético y conectividad inalámbrica integrada, características esenciales para un sistema autónomo distribuido como el planteado.

Desde el punto de vista técnico, el ESP32 ofrece las siguientes prestaciones:

Procesador:	Dual-core Tensilica LX6 de 32 bits, con frecuencia de hasta **240 MHz**, permitiendo procesamiento paralelo real a través de la asignación de tareas a cada núcleo de manera explícita
Memoria:	**RAM:** ~520 KB de SRAM interna. **Flash:** típicamente 4 MB de memoria flash externa. **Particiones configurables:** en Pampa se utiliza el esquema `default_4MB.csv`, que permite dividir la memoria en secciones como `app`, `spiffs`, `ota`, etc., facilitando actualizaciones OTA y almacenamiento persistente de archivos HTML/CSS/JS. Aunque posteriormente se perdió la capacidad de actualización por una decisión técnica.

Conectividad:	WiFi 802.11 b/g/n integrada. Bluetooth clásico y BLE (utilizado para el control con joystick PS3).
Puertos de Entrada/Salida (GPIO):	El ESP32 dispone de hasta **34 pines GPIO**, muchos de ellos multifunción. En el sistema Pampa, se asignaron pines para: ● Control de motores mediante PWM. ● Lectura de encoders (interrupciones). ● Sensores ultrasónicos (ECHO/TRIGGER). ● Comunicación I2C para el MPU6050. ● Interfaz con SPIFFS y control OTA.
Multitarea y asignación de núcleos:	Gracias a su compatibilidad con FreeRTOS, el ESP32 permite ejecutar múltiples tareas concurrentes (por ejemplo, lectura del joystick, control de motores, adquisición de datos de sensores) sin bloqueo, pudiendo incluso asignarse tareas críticas a núcleos separados para evitar latencias indeseadas. En el proyecto Pampa, por ejemplo, la tarea de movimiento se ejecuta en el core 1, mientras que la del tacómetro se ejecuta en el core 0, logrando así un balance eficiente del procesamiento.

Esta arquitectura de procesamiento paralelo, sumada a su bajo consumo, lo convierte en una plataforma ideal para robótica móvil, donde se requiere control preciso en tiempo real, reactividad a estímulos y conectividad constante sin sacrificar autonomía.

3.3.2 Aplicación específica de PWM en el sistema Pampa

En el contexto del proyecto Pampa, la señal PWM se utiliza para regular la velocidad de los motores de tracción, permitiendo maniobras suaves, corrección de rumbo y respuesta a estrategias de navegación.

El microcontrolador ESP32 dispone de múltiples canales PWM (LEDC) configurables mediante la API de Arduino. En este sistema se configuraron:

- Resolución: 8 bits (valores de 0 a 255)
- Frecuencia: 5 kHz

- Canales separados para cada dirección de giro de cada motor (izquierdo y derecho)

Estos canales alimentan las entradas del driver BTS7960, el cual traduce la señal PWM en energía motriz para los motores DC de engranaje helicoidal.
El PWM también es central en la implementación de estrategias avanzadas de control, como el uso de algoritmos de retroalimentación (PID y lógica difusa), que ajustan dinámicamente el duty cycle para compensar errores de velocidad o trayectoria.
Finalmente, la elección de una frecuencia de 5 kHz permitió evitar fenómenos de vibración o ruido audible, manteniendo la eficiencia del sistema sin afectar la percepción mecánica del movimiento.

3.3 Diseño de la arquitectura electrónica

El diseño electrónico del sistema Pampa fue concebido para integrar de forma eficiente todos los módulos funcionales del robot, garantizando la estabilidad operativa, la distribución adecuada de energía y la posibilidad de mantenimiento o expansión. A continuación se detallan los principales aspectos técnicos del diseño.

3.3.1 Conectividad y modularidad

Para facilitar el montaje, diagnóstico y modificación durante la etapa de prototipo, se optó por conectores tipo Dupont en la mayoría de los módulos de señal y sensores. Aunque este tipo de conector no ofrece alta resistencia a vibraciones, su uso permite rápida intervención y ajuste de conexiones. Como mejora futura, se contempla la migración hacia conectores asegurables o soldaduras definitivas en versiones posteriores del sistema.

En cambio, los motores de tracción y los conversores step-down fueron conectados mediante soldadura directa, priorizando la robustez eléctrica en los elementos con mayor consumo de corriente.

3.3.2 Distribución de energía y protección

El sistema utiliza una batería principal de **12V y 7Ah**, desde la cual se alimenta:

- El sistema de potencia (motores), directamente a través de los drivers BTS7960.
- El sistema lógico, mediante conversores step-down que regulan a 5 V para el ESP 32 y sensores.

Se incorporó un diodo de protección para evitar retorno de corriente desde el pin Vin del ESP32 hacia el step-down en el momento de carga o flasheo por USB. Además, se integró un interruptor físico general y una ficha de carga externa, facilitando la operación segura del sistema en campo.

3.3.3 Separación de señales y líneas de potencia

En el diseño final del cableado se implementó una **separación física y lógica** entre las líneas de señal (PWM, I2C, entradas digitales de sensores) y las líneas de potencia (12 V y 5V de fuerza). Esta separación permite minimizar el acoplamiento de ruido electromagnético, lo cual es especialmente relevante para asegurar la precisión en la lectura de sensores ultrasónicos y giroscopios durante condiciones dinámicas.

3.3.4 Montaje estructural de los módulos

El microcontrolador ESP32 fue ubicado centralmente en el chasis y fijado sobre una montura personalizada diseñada en impresión 3D, lo que facilita tanto el acceso a los pines como el ordenamiento simétrico del cableado. Por su parte, los drivers BTS7960 fueron montados mediante tornillería, garantizando firmeza estructural y disipación térmica pasiva por contacto con la superficie del chasis.

3.3.5 Creación de PCB a medida para facilitar las conexiones

A medida que la cantidad de sensores y componentes aumentaba, fue necesario encontrar una estrategia de conexiones, ya que al desmontar el robot y volverlo a montar los errores de conecciones dada la cantidad de cables se fueron incrementando ocasionando en algunos casos, roturas en algunos componentes por conectar mal algun pin, Para esto se utilizó el programa **Proteus** para implementar una placa electronica a medida con algunas consideraciones.

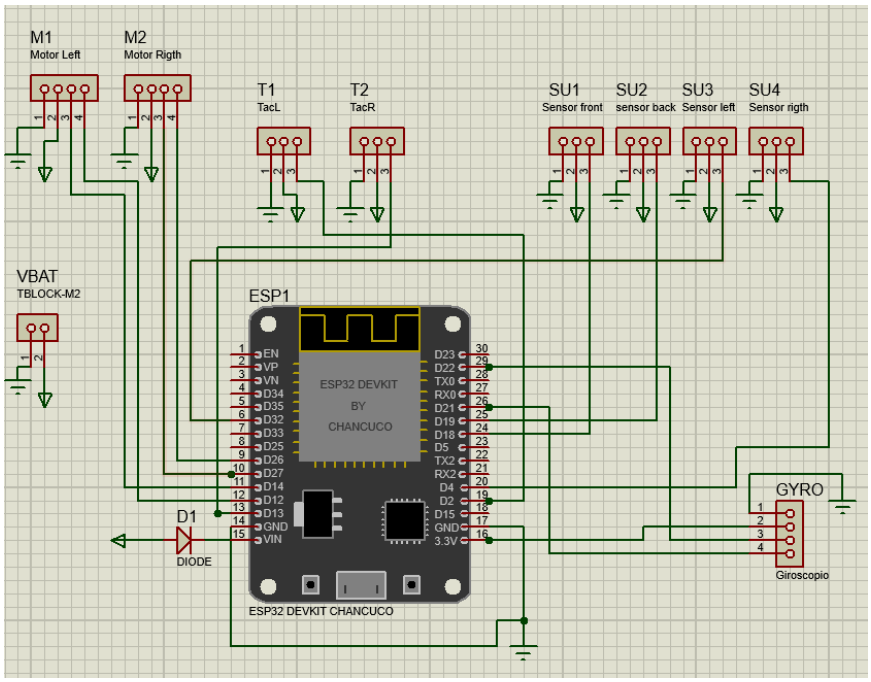


Figura 4.3.5 - Esquema electrónico de conexiones
Fuente: Propia

Apoyándonos en el archivo de **configuraciones** del proyecto generamos un esquemático del proyecto que después se emplea en el software para realizar el ruteado de las pistas

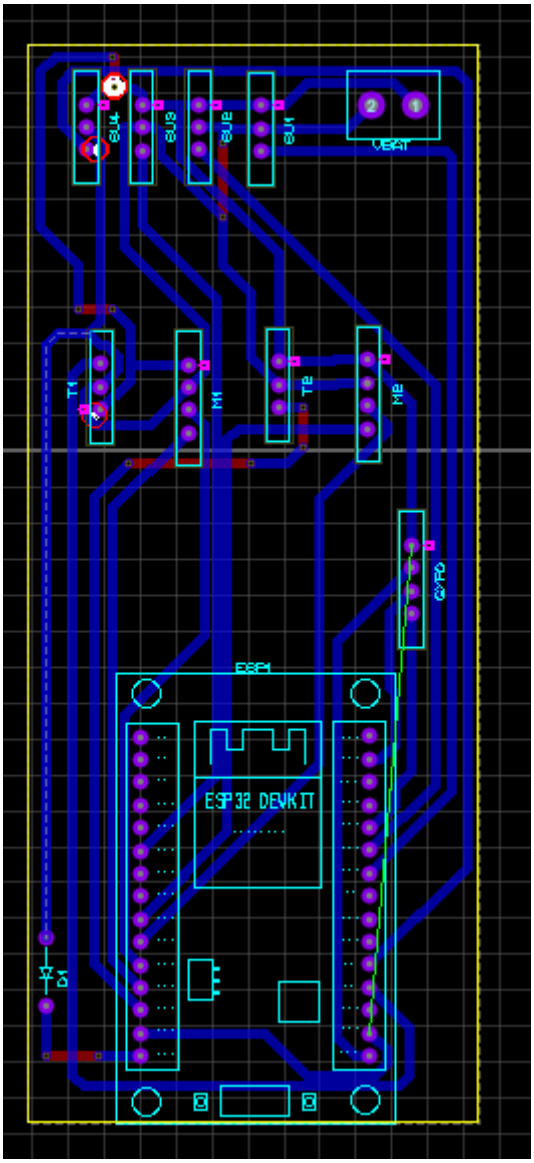


Figura 3.3.5a: vista superior de pcb con componentes
Fuente: Propia

Posteriormente se grabó en un pcb mediante la técnica de transferencia y se diluyó el cobre con percloruro férrico

Figura 3.3.5b: Circuito ya impreso en placa
Fuente: Propia

4. Diseño mecánico

El diseño mecánico del sistema Pampa fue concebido desde una perspectiva funcional y adaptable, orientado a garantizar la **movilidad estable del robot en entornos irregulares** y al mismo tiempo proporcionar espacio estructurado para la integración de los distintos módulos electrónicos, sensores, batería y cables de alimentación.

A lo largo del desarrollo, se adoptó un enfoque iterativo mediante el uso de herramientas CAD (específicamente SolidWorks), lo cual permitió analizar interferencias, ajustar tolerancias y **optimizar la relación entre la distribución de masa, tracción y espacio disponible**. El uso de un modelo digital completo también facilitó la planificación del montaje físico y la validación previa al ensamblado real.

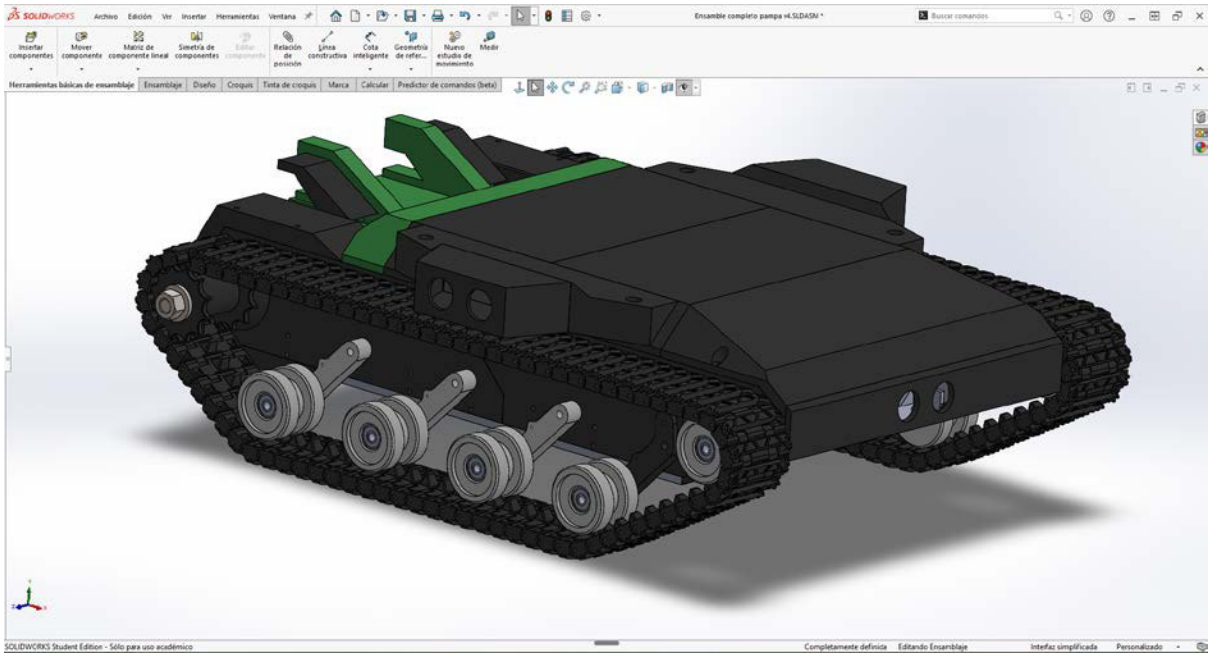


Figura 4 *- Ensamblaje completo del sistema Pampa en SolidWorks.*
Fuente: Propia

En la Figura 5 se observa el conjunto mecánico del robot con sus orugas montadas, cinco ruedas de apoyo por lateral, y el sistema de carcasa superior que aloja los módulos electrónicos. La cubierta central resaltada en azul representa la **tapa desmontable**, diseñada para facilitar el acceso a los componentes internos (como el ESP32, drivers BTS7960 y baterías). Los motores se encuentran posicionados en la parte frontal del sistema, acoplados directamente al eje principal de tracción mediante un sistema de perno. El diseño fue modelado íntegramente para producción mediante impresión 3D y considera la simetría de ensamblado para reducir la cantidad de piezas únicas.

4.1 Diseño general de la estructura física

La estructura principal está conformada por un **chasis rectangular compacto**, con un ancho suficiente para alojar la electrónica en la parte superior y sistemas de tracción en ambos laterales. Se definió una geometría simétrica para facilitar la programación de movimientos y equilibrar el peso.

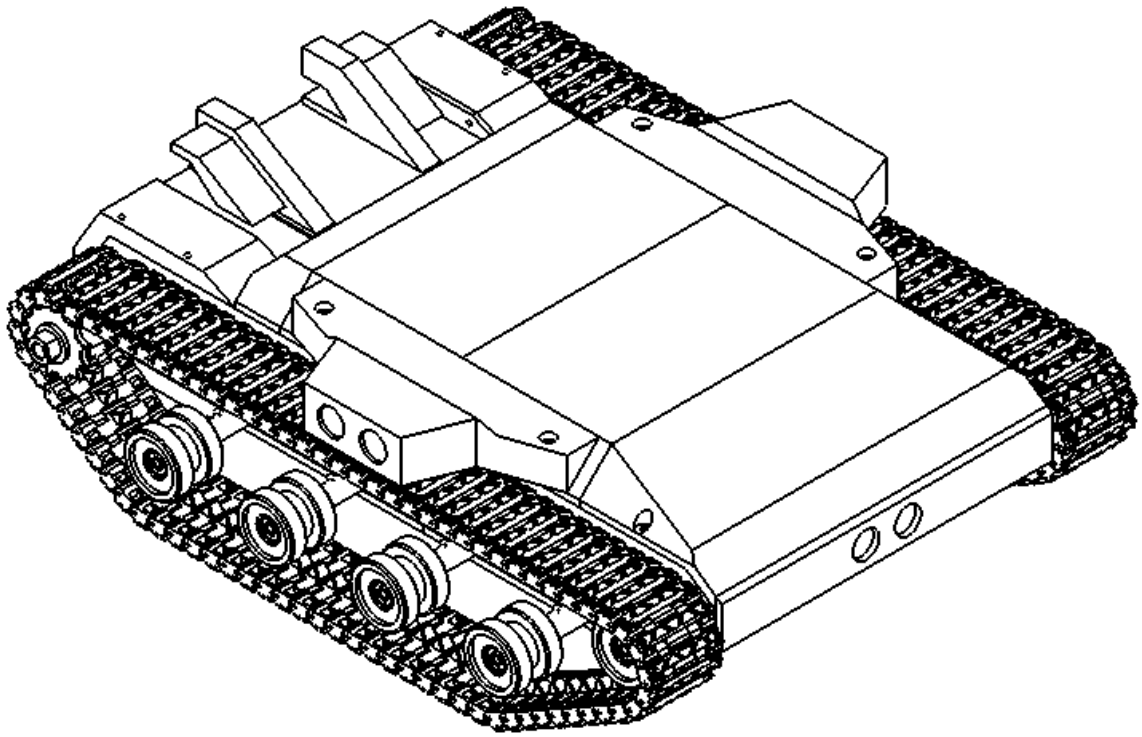


Figura 4.1 - Vista isométrica del modelo 3D de Pampa
Fuente: Propia

4.2 Sistema de tracción

Se optó por un sistema de **orugas montadas sobre cinco ruedas de soporte**, una rueda tractora y una guía tensora en cada lateral. Este diseño garantiza una **tracción continua** incluso en superficies irregulares, mejora el reparto del peso y minimiza el deslizamiento.

El sistema posee una longitud total de **401,37 mm** y una altura de **72 mm**, lo que permite suficiente espacio

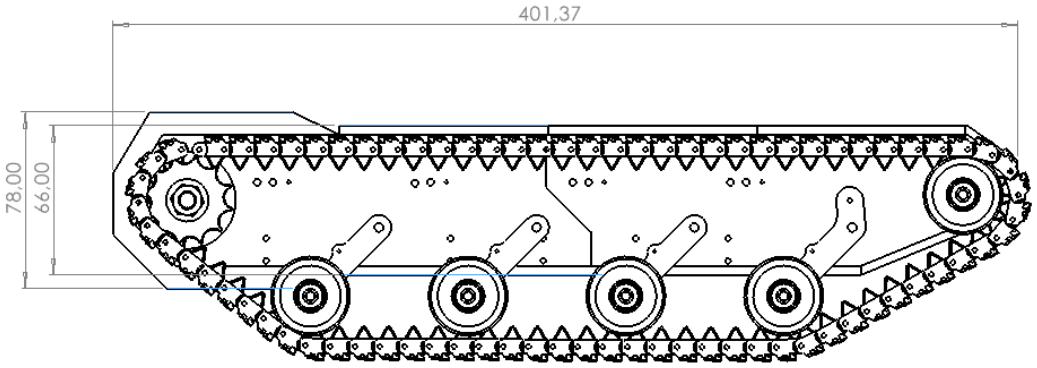


Figura 4.2 - Vista lateral acotada del sistema de tracción por orugas.
Fuente: Propia

4.3 Diseño funcional del disco ranurado y sistema de acople al motor

El disco ranurado cumple un rol fundamental dentro del sistema de medición del desplazamiento en el robot Pampa. Este componente fue diseñado específicamente para trabajar en conjunto con un sensor óptico, permitiendo estimar la **velocidad angular del eje** y, a partir de ella, la **velocidad lineal del vehículo**.

El disco cuenta con **24 ranuras equiespaciadas**, elegidas estratégicamente por dos razones principales:

- Permiten un **divisor exacto de 360°**, lo que simplifica el procesamiento de datos y facilita la conversión entre rotación y avance.
- Garantizan una resolución angular suficiente para el tipo de movimiento y precisión buscados, sin saturar el procesamiento del ESP32.

La fórmula utilizada para transformar los pulsos en distancia lineal recorrida es:

$$v = \omega \times r$$

donde:

- $\omega = \frac{2\pi.N}{n.T}$ = Velocidad Angular (rad/s)
  - N = número de pulsos registrados en el tiempo T
  - n = número total de ranuras (24)
- r = radio efectivo de la rueda motriz (en metros)

Así, conociendo el radio r de la rueda y la cantidad de pulsos detectados, es posible estimar el **avance lineal** con buena precisión. En el código del sistema, esta conversión se realiza mediante una constante empírica calibrada, **CM_POR_PULSO = 0.693**, que ya incluye el radio de la rueda y la densidad de ranuras.

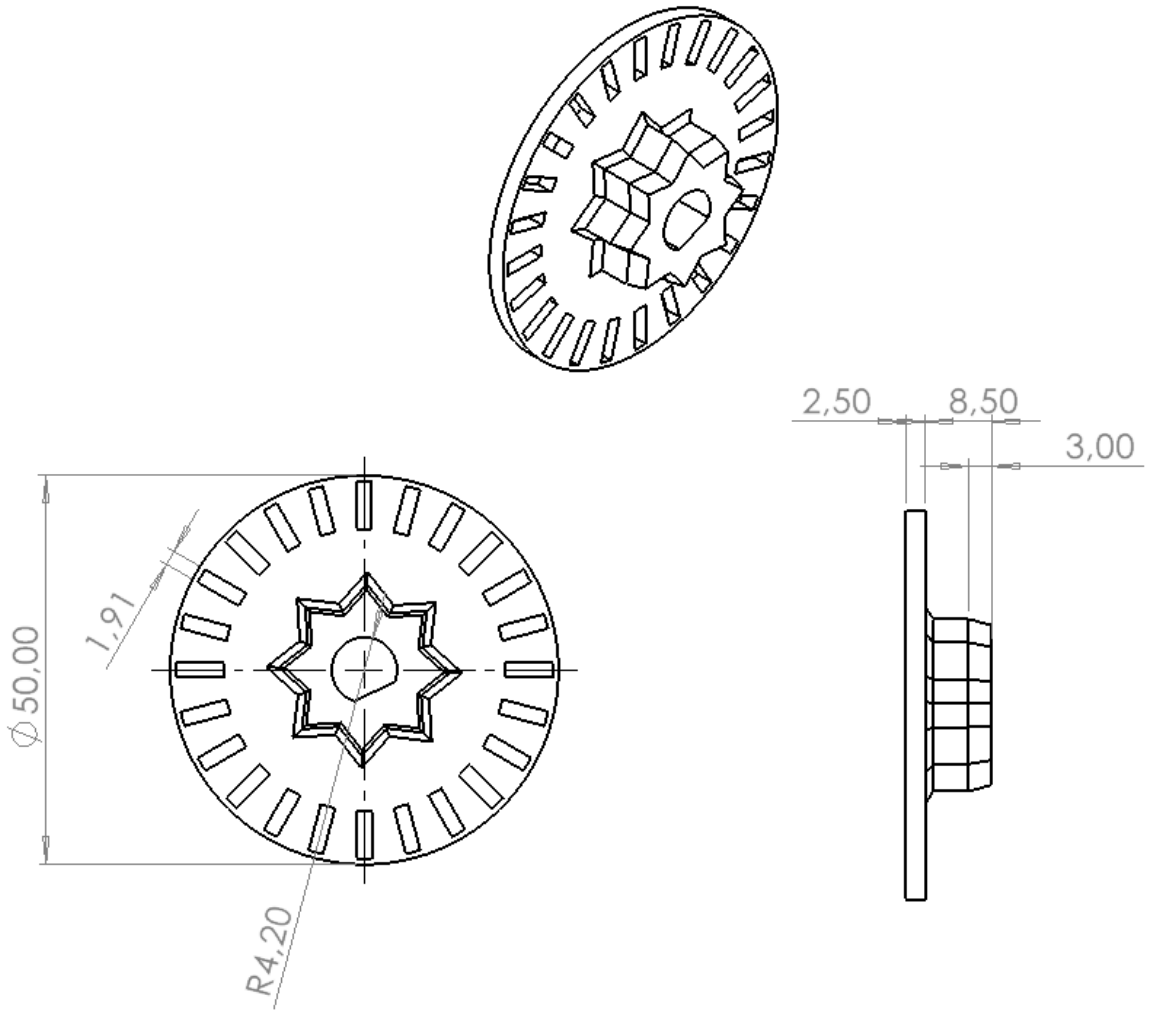


Figura 4.3 – Disco ranurado con 24 divisiones. Vista frontal, isométrica y de sección. El anillo central está diseñado para acoplarse directamente al eje del motor mediante perno. Fuente: Propia

Una característica destacada del diseño es la protuberancia central (ver Figura 7.2.3), que cumple una doble función:

1. Permite el encastre firme del disco al eje del motor, sin necesidad de adhesivos ni fijaciones externas.
2. Hace posible retirar y reemplazar fácilmente el motor o el disco sin desmontar el eje completo.

Este sistema de perno permite acoplar el disco a distintos ejes motrices estandarizados, lo que facilita las pruebas experimentales, el reemplazo de motores en laboratorio y la iteración del diseño mecánico sin generar desechos o reconstruir el módulo completo.

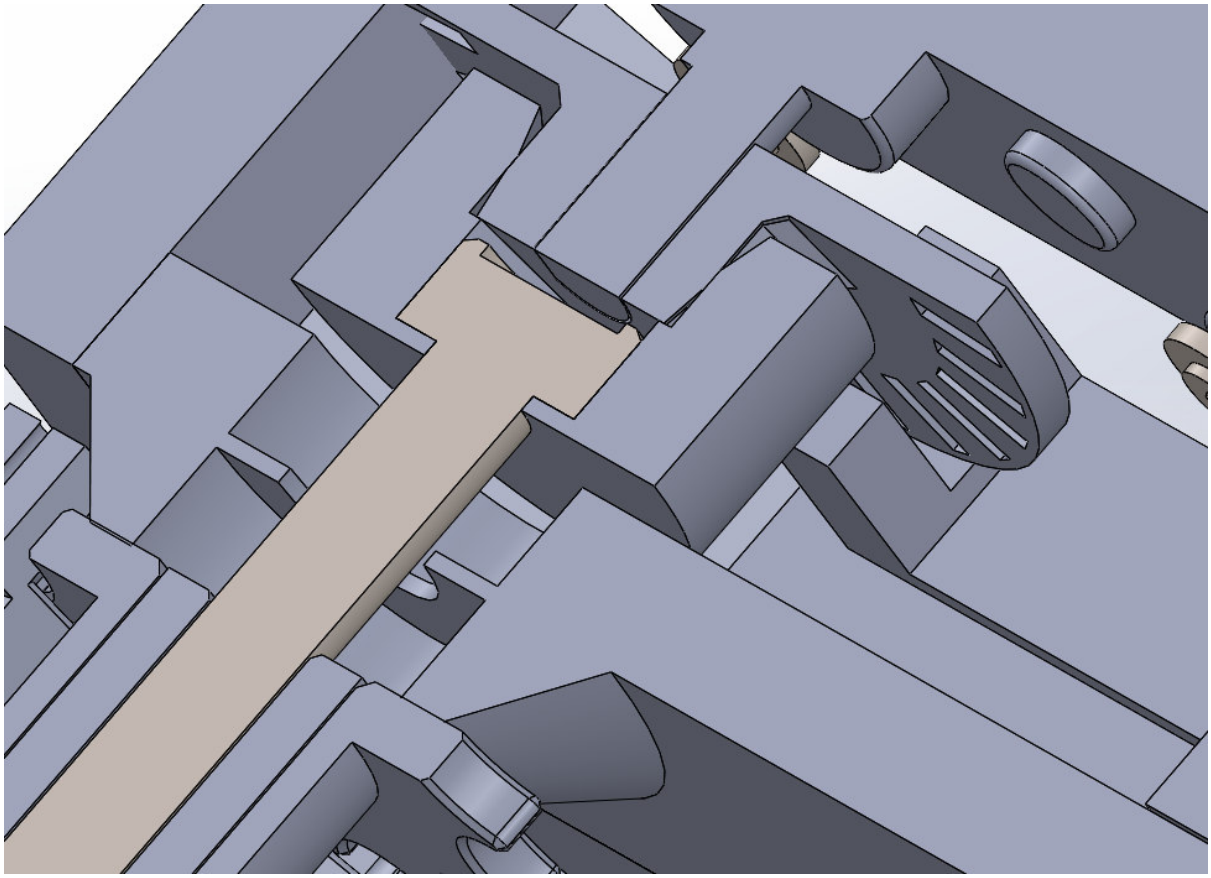


Figura 4.3b - Detalle de la carcasa mostrando el sistema de acople entre el motor y el disco. Fuente: Propia

4.4 Soportes estructurales para sensores y módulos electrónicos

Los módulos como el **ESP32**, los **drivers BTS7960**, el giroscopio y los sensores ultrasónicos fueron montados sobre **soportes diseñados a medida** mediante impresión 3D. Esto permitió alinear los sensores correctamente respecto al frente del robot, asegurar la refrigeración pasiva de los controladores, y mantener el centro de masa bajo.

4.5 Ajuste de la integración mecánica-electrónica

Durante las pruebas en campo, se realizaron ajustes para evitar interferencias entre los cables de potencia y las líneas de señal. También se evaluó la resistencia mecánica de las uniones impresas, reforzando con tornillería en las zonas críticas. El diseño final logra mantener **rigidez estructural**, modularidad para mantenimiento, y espacio adecuado para futuras expansiones.

4.6 Diseño e impresión 3D

El recorrido por el diseño e impresión 3D del sistema Pampa comenzó en el entorno digital de SolidWorks, donde cada pieza (desde el disco ranurado hasta los eslabones de las orugas) se modeló con precisión milimétrica. Gracias a esta etapa, detectamos colisiones tan sutiles como el paso del disco por el sensor óptico o la interferencia entre los eslabones y los soportes del chasis, evitando múltiples impresiones fallidas y optimizando el flujo de trabajo desde el CAD hasta la impresora.
Inicialmente utilizamos Marlin en la Creality Ender 3 SE, pero pronto enfrentamos pérdidas de pasos al imprimir múltiples piezas pequeñas (como los eslabones) en una sola pasada. La migración a Klipper no solo estabilizó el movimiento, sino que redujo el tiempo de impresión y mejoró la fiabilidad en esquinas y cambios de dirección. Por último, encontramos en OrcaSlicer la solución definitiva: perfiles preajustados de relleno, corte por partes, tests automáticos de flujo y retracción, y herramientas para recortar directamente desde el CAD. Este conjunto de capacidades aceleró enormemente el ciclo "modelar–imprimir–evaluar–ajustar", permitiéndonos iterar con agilidad y alcanzar la versión final que describe esta sección.

4.6.1. Software utilizado

El flujo de trabajo de diseño e impresión 3D del sistema Pampa se basó en una combinación de herramientas de modelado CAD y de laminado (slicing), cada una elegida por sus fortalezas en la etapa correspondiente.

SolidWorks 2025 Student Edition

- Versión completa con licencia académica (USD 75), instalada en un equipo con AMD Ryzen 7 5700G (8 núcleos, 16 hilos, 3,8 GHz base, 4,4 GHz turbo), 32 GB de RAM DDR4 y almacenamiento en SSD NVMe.
- Uso intensivo de la función de ensamblajes y de detección de interferencias para validar clearance antes de imprimir: por ejemplo, gracias a esta comprobación se ajustó el paso de los eslabones entre sí y su interacción con el sensor óptico, minimizando soportes y repeticiones de impresión.
- Empleo de extensiones de archivo:
    - .sldprt para el desarrollo individual de piezas (discos, anclajes, soportes).
    - .sldasm para crear el ensamblaje completo del chasis y generar vistas de montaje.
- Organización de versiones mediante una convención de nombres tipo PV1, PV2_motor, etc., en un único archivo de ensamblaje para evitar problemas de vinculación.

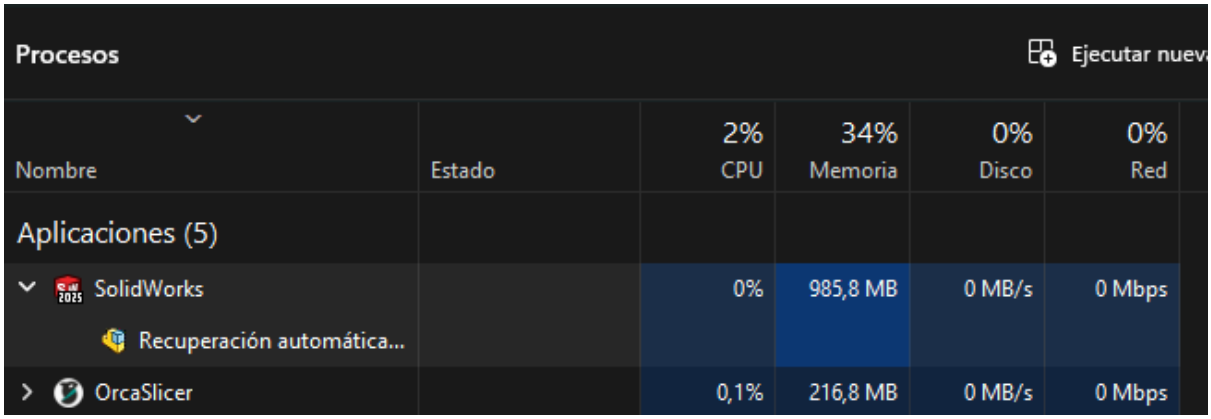


Figura 4.6.1 – Panel de rendimiento de SolidWorks mostrando el uso de CPU y memoria al cargar el ensamblaje completo Pampa.
Fuente: Propia

OrcaSlicer (perfil Grillon PLA)

- Se eligió como slicer principal por sus tests automáticos de calibración (flujo, retracción, temperatura) y por permitir corte por regiones directamente en CAD.
- Ajustes personalizados: diámetro de boquilla 0,4 mm, filamento Grillon PLA, velocidad de 60 mm/s, infill variable según geometría.

Ultimaker Cura y Creality Slicer

- Complementarios para primera calibración y validación cruzada de perfiles. Cura sirvió para iterar rápidamente parámetros básicos; Creality Slicer se empleó para calibrar la Ender 3 SE tras migrar el firmware a Klipper, mejorando la estabilidad de los eslabones pequeños durante impresiones múltiples.

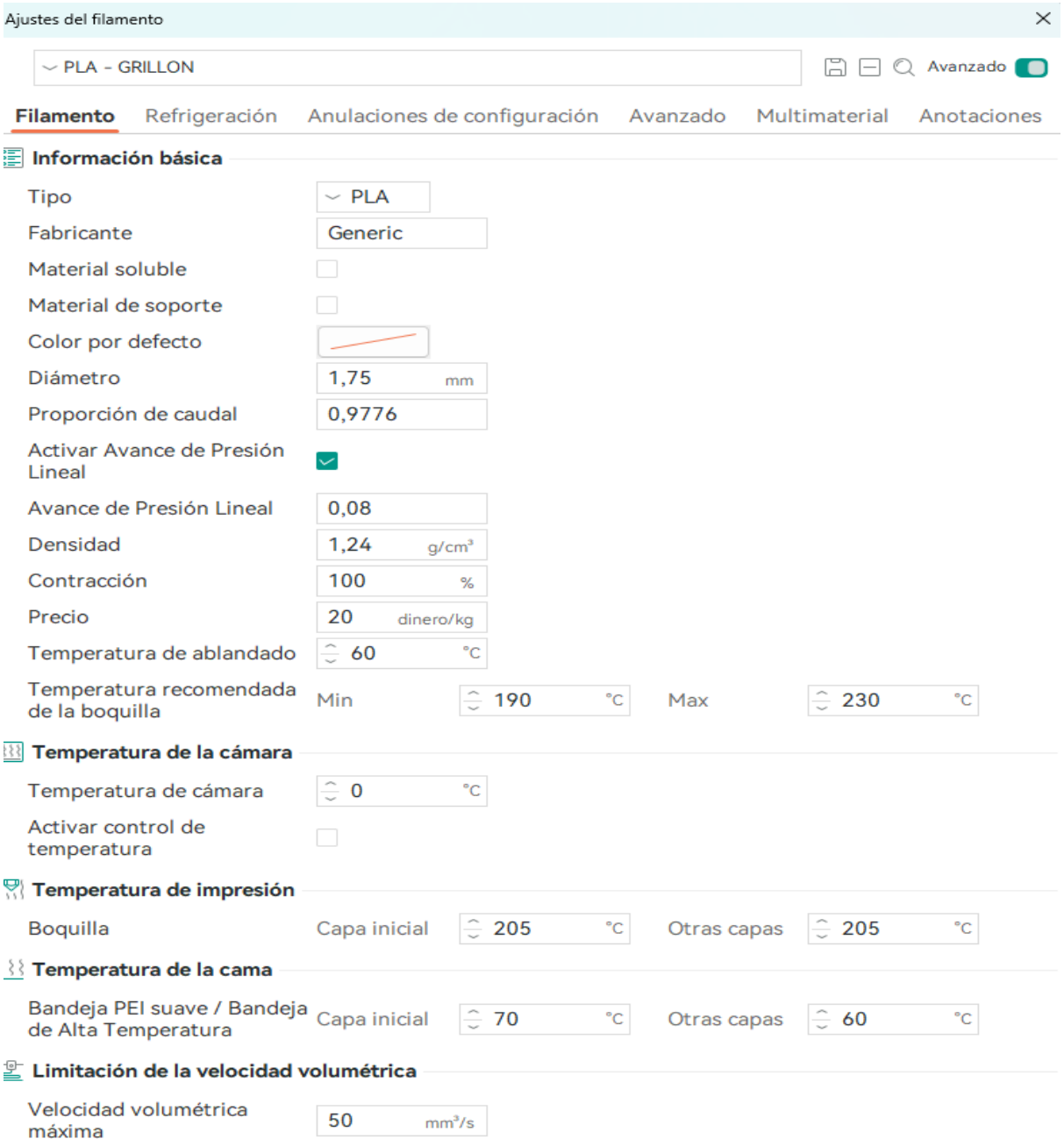


Figura 4.6.1b: – Captura de OrcaSlicer con perfil optimizado para filamento Grillon PLA, mostrando parámetros de retracción y test de flujo.
Fuente: Propia

4.6.2. Modelado de componentes personalizados

Durante el desarrollo del sistema, fue necesario modelar una serie de componentes personalizados para garantizar una integración precisa entre los módulos electrónicos (como el tacómetro, los motores y el sistema de tracción) y la estructura mecánica general del prototipo.

El modelado se realizó utilizando el software **SolidWorks**, lo cual permitió diseñar piezas paramétricas y ensamblajes virtuales complejos. Se trabajó inicialmente con un lateral del chasis, al que luego se aplicaron simetrías para obtener la geometría opuesta, optimizando tiempo y asegurando coherencia estructural.

Un aspecto destacable del proceso fue que **algunas piezas se modelaron anticipadamente, incluso antes de recibir el componente real**, basándose únicamente en planos o medidas proporcionadas por el fabricante. Un claro ejemplo de esto fue el motorreductor metálico, cuyo modelo CAD fue diseñado y posteriormente impreso en 3D para comprobar su ajuste. La imagen a continuación muestra el modelo impreso (izquierda) junto al motor real (derecha), confirmando la exactitud del diseño:

Figura 4.6.1 - Prototipo impreso en 3D del motorreductor junto al componente original. Fuente: Propia

Además, se diseñaron piezas específicas como:

- Discos ranurados para el sistema tacométrico óptico.
- Soportes de sensores con guías y posicionamiento preciso.
- Adaptadores para acoplar ejes, ruedas dentadas y soportes estructurales.
- Ensambles funcionales que se validaron virtualmente antes de pasar a impresión.

La siguiente figura muestra un detalle del conjunto de tracción, incluyendo el ajuste entre el **disco ranurado** y el **sensor tacómetro**:

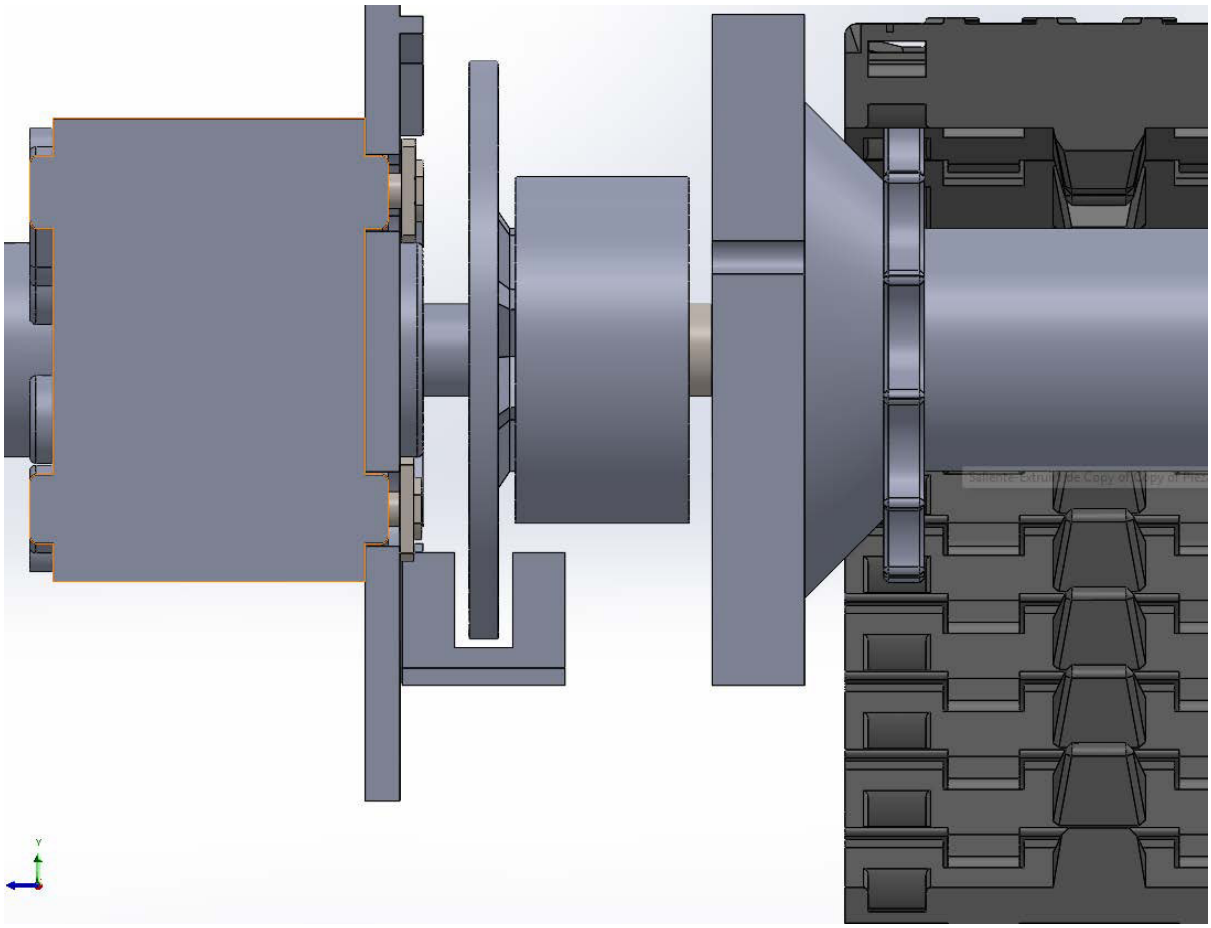


Figura 4.6.1a - Ensamblaje parcial donde se observa el posicionamiento del sensor respecto al disco ranurado y a la oruga.
Fuente: Propia

Finalmente, se diseñó una réplica virtual completa del motor (basado en mediciones y geometría visual), lo que permitió generar piezas de acople sin necesidad de esperar a tener el componente físico:

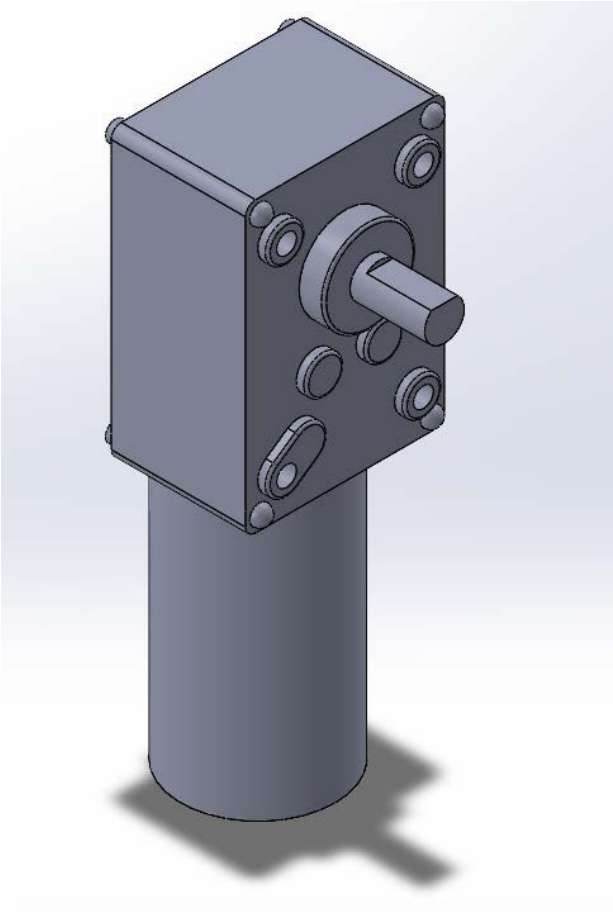


Figura 4.6.3b: Modelo 3D del motorreductor metálico creado en SolidWorks para pruebas virtuales y diseño de soportes.
Fuente: Propia

Este enfoque, basado en modelado anticipado y verificación posterior, fue clave para acelerar el proceso de desarrollo y reducir iteraciones físicas innecesarias.

4.6.3. Rol del prototipado rápido en el desarrollo interdisciplinario

El prototipado rápido fue necesario para articular mecánica, electrónica y software en el proyecto Pampa. Gracias a la capacidad de imprimir piezas en cuestión de horas, pudimos avanzar de manera iterativa en el **sistema de acoplamiento de los motores**, ajustando a la vez el diseño mecánico y la lógica de control. A continuación se resumen las cuatro versiones principales del acople:

4.6.3.1. Versión 1 – Acople por presión simple motor paso a paso nema 17

En esta primera iteración, el acople consistió en un manguito impreso que se ajustaba a presión sobre el eje de un motor paso a paso NEMA 17. Aunque la pieza se diseñaba más fácilmente, pronto surgieron dos problemas críticos:

- Ineficiencia energética: los motores NEMA 17 requieren un suministro de corriente constante; al acoplarlos de esta forma se detectaron picos de consumo elevados que reducían notablemente la autonomía del sistema.
- Holguras y dimensionado incorrecto: al comenzar a usar la impresora 3D no se había calibrado aún la correspondencia entre el modelo CAD y las dimensiones reales de la impresión, lo que provocó acoples con excesiva holgura o deformaciones en el manguito, impidiendo un ajuste firme y requiriendo repetidas revisiones.

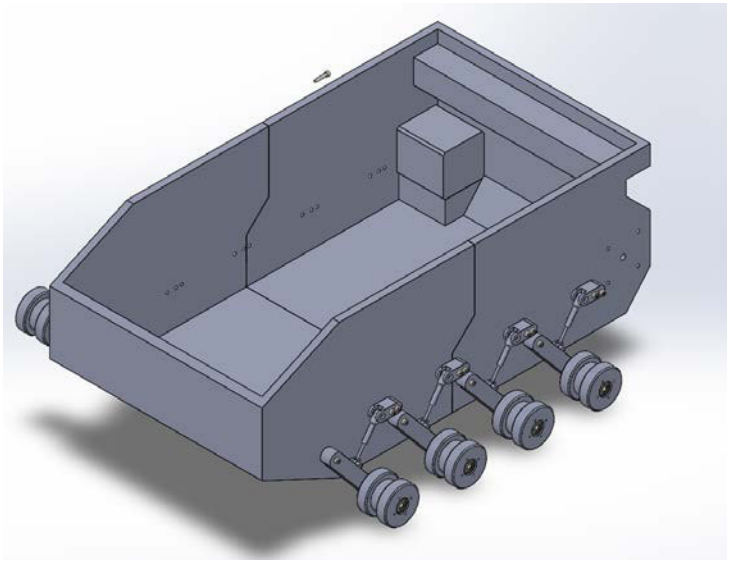


Figura 4.6.3.1 - Chasis con montura para motor nema 17 con corte para facilitar la impresión 3d (Pampa V1)
Fuente: Propia

4.6.3.2. Versión 2 – Moto reductores comerciales

En la segunda iteración se optó por motores con caja reductora comercial, acoplados de modo que un extremo del eje aloja el **disco ranurado** para el encoder y el otro sostiene el **piñón** de transmisión a la oruga.
También para poder modularizar el diseño se dividen los soportes de las orugas en dos, de manera que si se requiriera más ancho solamente fuera necesario imprimir la pieza que actuaría como separador de estas.
Esta configuración aportó un elevado torque inicial, pero pronto se identificaron dos deficiencias críticas:

- **Velocidad insuficiente:** los reductores comerciales redujeron la velocidad de salida a un nivel tan bajo que, aunque el par era adecuado, el avance resultante quedaba limitado a muy pocas revoluciones por minuto.
  - **Rango efectivo de PWM acotado:** debido a la baja velocidad nominal, solo un intervalo muy estrecho (aprox. 90 %–100 % del duty cycle) producía movimiento

real, impidiendo el control fino necesario para maniobras de precisión y haciendo errático el ajuste de la velocidad.

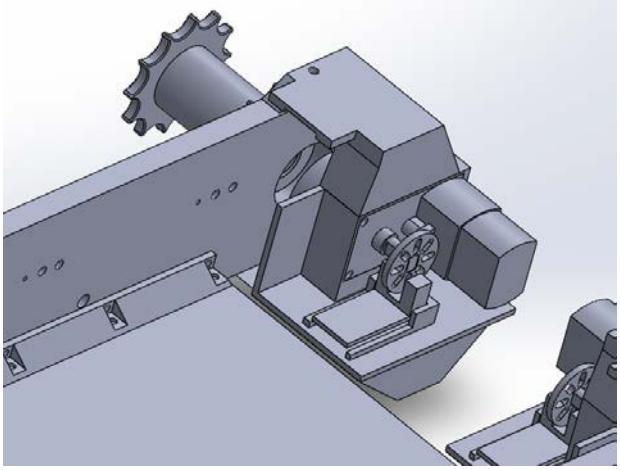


Figura 4.6.3.2: Acople de moto-reductor comercial con disco ranurado y piñón
Fuente: Propia

Esta experiencia puso de relieve la necesidad de encontrar un compromiso mejor entre torque y velocidad, así como de validar empíricamente el rango de PWM utilizable antes de fijar el diseño mecánico.

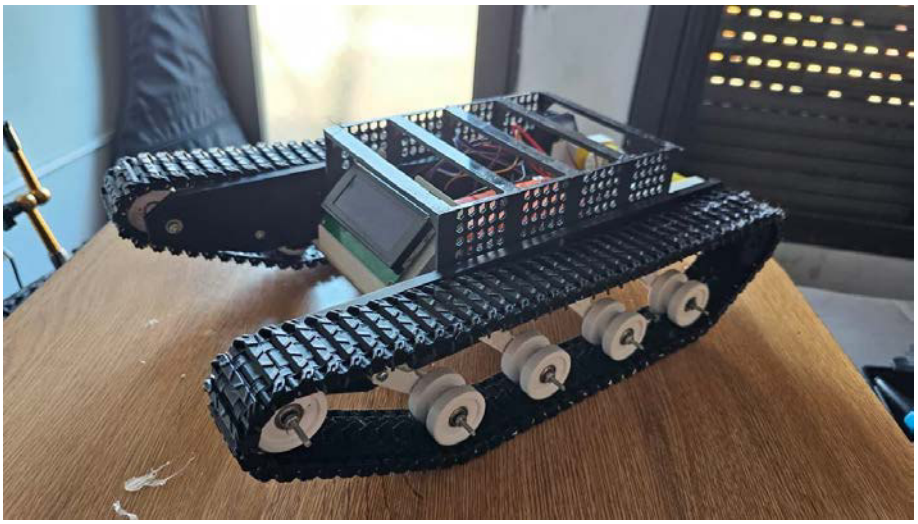


Figura 4.6.3.2 - Pampa V2 prototipo con pantalla incluida
Fuente: Propia

4.6.3.3. Versión 3 – Acople con motores de corriente continua

En esta tercera iteración se sustituyó el motoreductor por un **motor DC de 20 000 rpm** directo al eje, con la intención de maximizar la velocidad de salida. Sin embargo, al carecer de caja reductora, el conjunto adoleció de **torque suficiente** para mover el chasis de 5 kg en terrenos

reales. Este desfase entre velocidad nominal y par entregado quedó patente en las pruebas de campo, donde el robot apenas alcanzaba a superar el rozamiento estático. Como consecuencia, fue necesario rediseñar nuevamente el acople, incorporando una etapa reductora que permitiera equilibrar torque y velocidad para un control más fino.

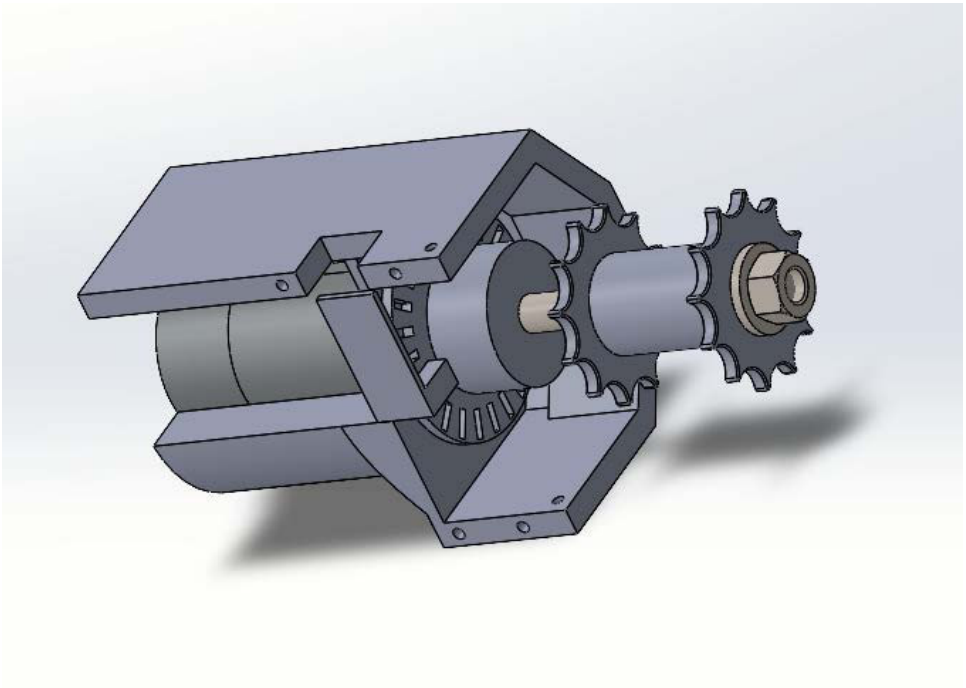


Figura 4.6.3.3 - Acople provisional con motor DC de alta velocidad (20 000 rpm)
Fuente: Propia

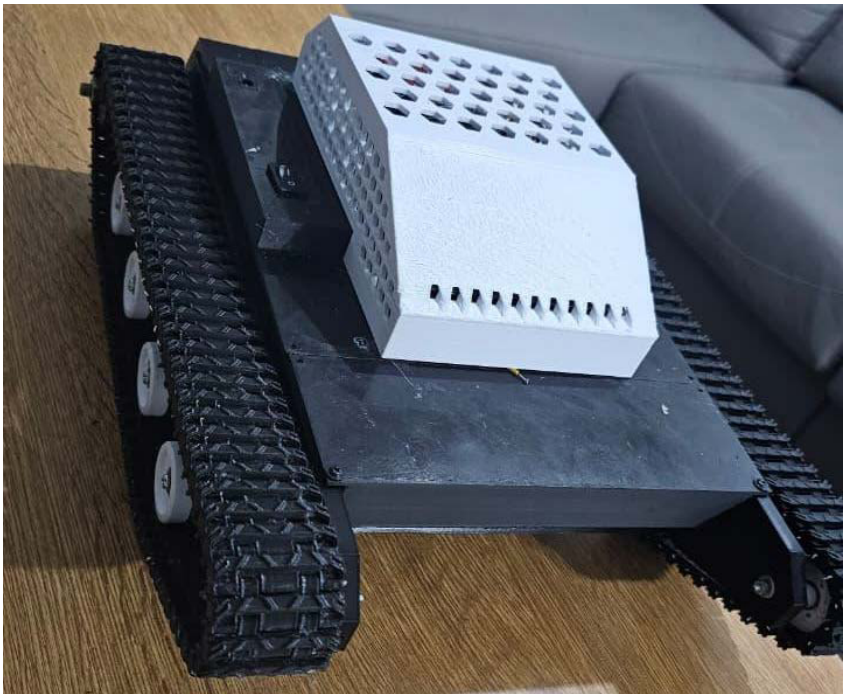


Figura 4.6.3.3a - Pampa V3
Fuente: Propia

4.6.3.4 Versión final – Acople para 460 rpm

El prototipo definitivo fue diseñado para ejes más gruesos y motoreductores de alto par. Presenta un collarín reforzado y nervaduras radiales que resistieron pruebas de tracción de hasta 5 Nm. Gracias al prototipado rápido, imprimir esta versión llevó menos de 2 horas, y

permitió validar en el campo el comportamiento dinámico sin necesidad de reconstruir todo el chasis.

Impacto interdisciplinario:

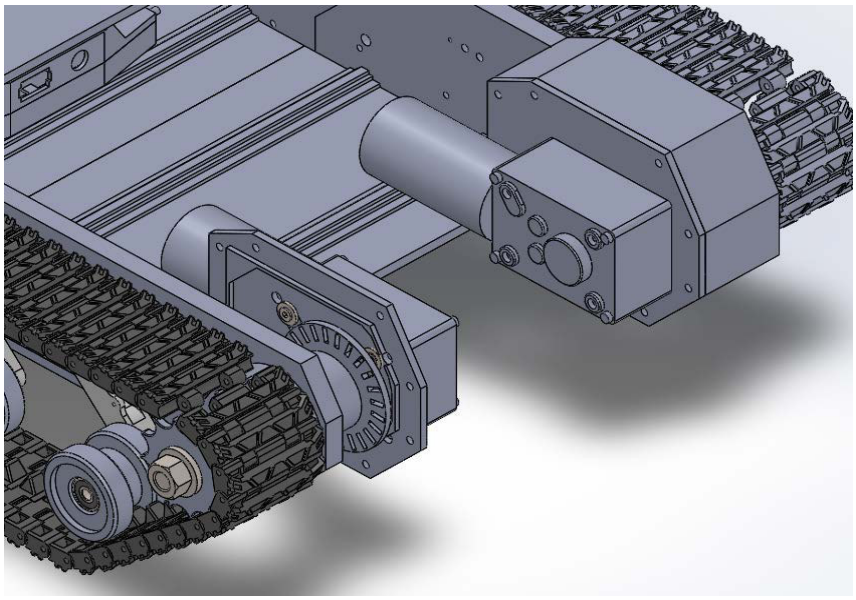
- **Mecánica:** Cada iteración se basó en mediciones reales de par y geometría del eje, lo que reforzó la precisión del modelado CAD.

- **Electrónica/Software:** Las variaciones físicas llevaron a recalibrar el conteo de pulsos y los algoritmos de control de velocidad (`CM_POR_PULSO`, ajustes de PWM), asegurando que la lógica de navegación funcionara de forma consistente con cada nuevo acople.
  La inclusión de un archivo de configuración global fue una decisión técnica muy acertada en esta área.

- **Ciclo ágil "modelar–imprimir–evaluar–ajustar":** Gracias a este flujo, pudimos integrar en cuestión de días mejoras que de otro modo habrían tardado semanas,

demostrando que el prototipado rápido es imprescindible en proyectos de grado con plazos ajustados y componentes multifuncionales.

Figura 4.6.3.4 - Acople final con motorreductor de 460 rpm de alto par.



Fuente: Propia

Figura 4.6.3.4a - Pampa V4 version final
Fuente: Propia

5. Programación

La implementación del software del Robot Autónomo Pampa se basó en una arquitectura modular, diseñada para maximizar la cohesión interna de cada componente y minimizar el acoplamiento entre ellos. Para ello se aplicaron de forma rigurosa los principios SOLID, garantizando que clases como **TankController** y **PS3ControllerManager** cumplieran el Principio de Responsabilidad Única (SRP) y la Inversión de Dependencias (DIP), mientras que el patrón Strategy permitió intercambiar dinámicamente algoritmos de navegación sin modificar el código cliente (Open/Closed, OCP).
Este enfoque facilitó:

- Extensibilidad: Añadir nuevas estrategias de movimiento o sensores sin tocar el núcleo de la aplicación.
- Testabilidad: Aislar módulos para pruebas unitarias y de integración.
- Mantenibilidad: Localizar y corregir errores con un impacto mínimo en otras partes del sistema.

El flujo de trabajo siguió una metodología ágil coordinada en Jira, con sprints centrados en entregables concretos (módulo de comunicación WebSocket, control de motores, navegación inteligente, etc.). El control de versiones se gestionó con Git, manteniendo ramas de características y releases estables para cada iteración, sin embargo este aspecto solo se menciona brevemente como soporte al proceso ágil.

5.1. Arquitectura del software y estructura modular

En la optimización del firmware para ESP32, uno de los objetivos principales fue minimizar el consumo de Flash sin sacrificar claridad ni mantenibilidad del código. Para ello, apostamos por una arquitectura modular y orientada a objetos que eliminara al máximo la duplicidad:

1. **Reutilización de componentes genéricos**
   En lugar de duplicar la lógica de control de motor para cada vía, creamos una única clase Motor que encapsula la configuración de LEDC (frecuencia = 5 kHz, resolución = 8 bit) y los métodos begin(), move(direction, speed) y stop(). Luego simplemente instanciamos dos objetos (**motorFL** y **motorFR**), lo que reduce drásticamente el código generado frente a tener dos implementaciones separadas.
2. **Sensado ultrasonido con un solo tipo**
   De forma similar, el sensor de proximidad se abstrajo en una clase **UltrasonicSensor** que gestiona el trigger, la espera de *pulseIn*() y el cálculo de distancia. En tiempo de ejecución, creamos cuatro instancias (frontal, trasero, izquierdo y derecho). Gracias a esta unificación, no sólo ganamos coherencia, sino que el compilador puede compartir el mismo fragmento de código en Flash, en lugar de inyectar copias por cada sensor.
3. ***CaterpillarController* como wrapper**
   El controlador de oruga combina motor y tacómetro. De nuevo, en vez de escribir dos

controladores casi idénticos, definimos **CaterpillarController** que recibe un **Motor*** y un *Tacometer**. Creamos dos objetos (uno por cada oruga) sin replicar ni una línea de lógica, y el binario sólo aloja el código de la clase una sola vez.

4. **Estrategias de navegación mediante uso de interfaz Strategy Pattern**
   Gracias a una interfaz base **MotionStrategy**, cada algoritmo ("manual", "evitación", "grilla", "Vector".) se implementa en su propia clase derivada. Esto evita copiar y pegar bloques de control en cada variante y permite al linker descartar las estrategias no usadas, reduciendo aún más el tamaño final.
5. Eliminación de la partición OTA para ganar espacio
   Al crecer el cómputo y las dependencias, el firmware llegó a ocupar más del 70 % de la Flash (aprox. 900 KB). Para recuperar espacio, se decidió eliminar la partición OTA (de ~128 KB), pasando de un layout con ota_0 y ota_1 a uno donde sólo existe app0. Así liberamos espacio inmediato y mantuvimos una sola partición de aplicación.

Gracias a este enfoque (una combinación de diseño SOLID, interfaces limpias y configuraciones de compilador agresivas) logramos reducir el footprint en Flash, mantener un código claro y modular, y ahorrar tanto bytes como esfuerzo de mantenimiento.

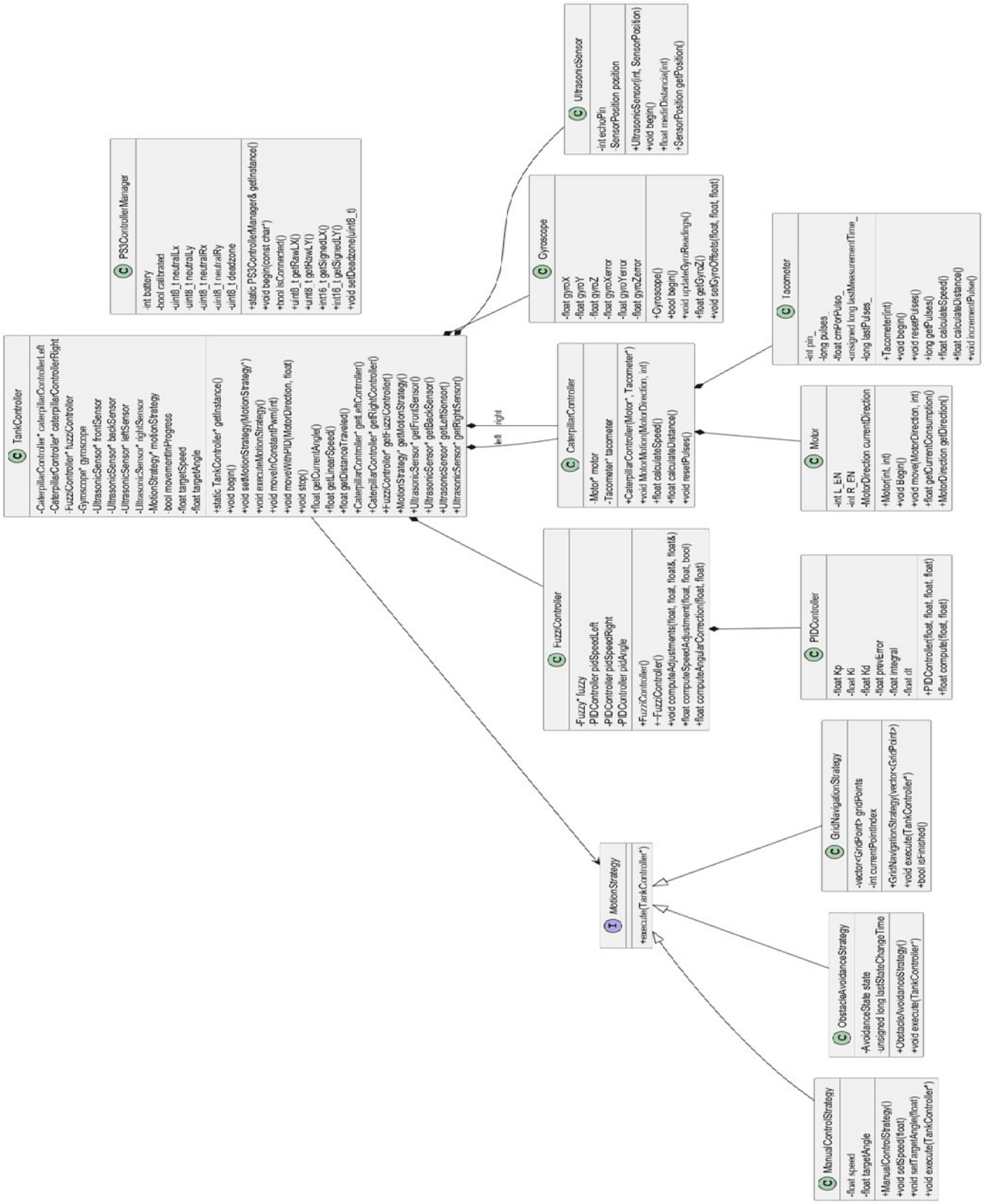


Figura 5.1 - Diagrama uml de composición de clases
Fuente: Propia

5.2. Modelado orientado a objetos y patrones de diseño

La clase **TankController** está implementada como un singleton:

- Su constructor es privado y se expone un método estático *getInstance()* que garantiza una única instancia en todo el firmware.
- Centraliza la orquestación de todos los subsistemas (orugas, sensores, giroscopio, estrategia de movimiento), evitando múltiples inicializaciones y conflictos de acceso al hardware.
- **PS3ControllerManager** utiliza la misma lógica: un solo objeto gestionando conexión, calibración y lectura del mando PS3.

Este enfoque asegura que, independientemente de cuántas veces invocamos getInstance(), siempre trabajamos sobre el mismo estado compartido, reduciendo el uso de memoria y simplificando la sincronización entre tareas de FreeRTOS.

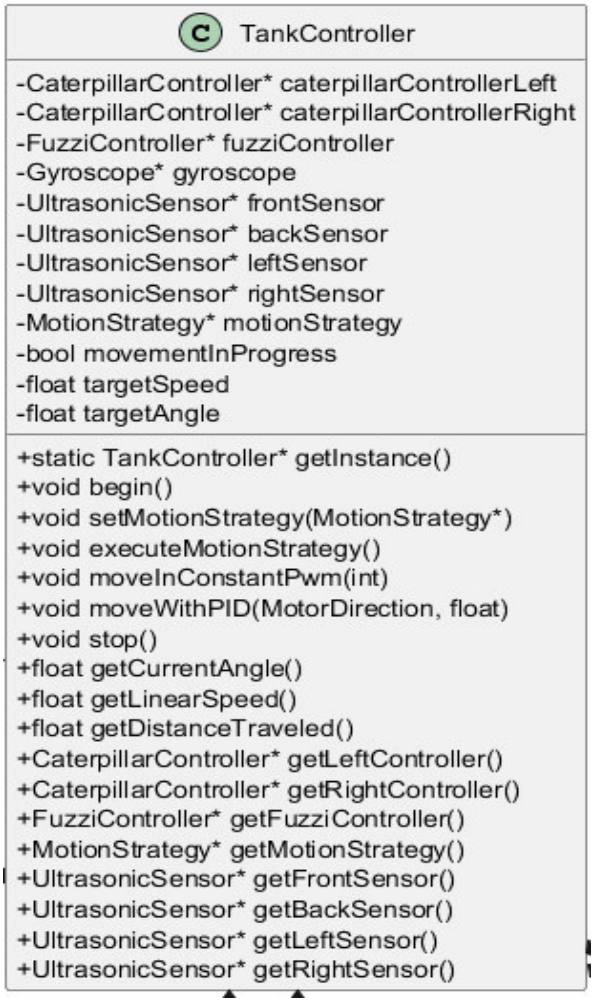


Figura 5.2 - Diagrama uml de entidad principal.
Fuente: Propia

Para la navegación empleamos el patrón **Strategy**, basado en una interfaz común:

```
class MotionStrategy {
public:
  virtual void execute(TankController* tank) = 0;
  virtual ~MotionStrategy(){}
};
```

De ella derivan tres implementaciones concretas:

- **ManualControlStrategy**: mapea el input del joystick PS3 a movimientos de orugas.
- **ObstacleAvoidanceStrategy**: máquina de estados que responde a lecturas de sensores ultrasónicos y **SAFE_DISTANCE**.
- **GridNavigationStrategy**: recorre una serie de puntos predefinidos, calculando errores de posición y corrigiendo con PID.

Gracias a esta abstracción, TankController puede cambiar de estrategia en tiempo de ejecución llamando simplemente a:

```
tank->setMotionStrategy(new ObstacleAvoidanceStrategy());
```

Sin necesidad de modificar su propia lógica interna ni recompilar, facilitando la extensibilidad y la limpieza del código.

5.3. Organización de código y dependencias

Adoptamos una estructura clara y modular que facilita tanto el desarrollo como la optimización para ESP32. Nuestro platformio.ini centraliza la configuración de la placa, el framework y las bibliotecas externas:

- lib_deps agrupa todas las dependencias externas (ESP-Async, Adafruit, PS3 Controller, Fuzzy, etc.), de modo que PlatformIO las descarga e integra automáticamente.
- extra_scripts nos permite inyectar pasos personalizados (por ejemplo el script print_tree.py) antes de la generación del ELF, para inspeccionar la estructura de carpetas o medir el tamaño de la memoria.

```
[env:usb]
platform          = espressif32@>=3.0.0,<4.0.0
board             = esp32doit-devkit-v1
framework         = arduino
lib_deps          =
  WiFi
  ESPmDNS
```

```
  ArduinoOTA
  Update
  SPI
  AsyncTCP
  jvpernis/PS3 Controller Host@^1.1.0
  zerokol/eFLL@^1.4.1
  mathieucarbou/ESPAsyncWebServer @ 3.3.7
  adafruit/Adafruit MPU6050@^2.2.6
  adafruit/Adafruit Unified Sensor@^1.1.15
  bblanchon/ArduinoJson@^7.3.0
  arduino-libraries/Arduino_JSON@^0.2.0
monitor_speed     = 115200
extra_scripts     = pre:print_tree.py
```

Por su parte, la organización en disco sigue este patrón:

```
/
├─ platformio.ini
├─ print_tree.py      ← hook que imprime la estructura antes de
linkear
├─ src/
│   └─ main.cpp       ← punto de entrada y creación de tareas
FreeRTOS
├─ include/
│   ├─ includes.h     ← todas las cabeceras globales
│   └─ MotionStrategy/...  ← interfaz y estrategias de navegación
├─ lib/
│   ├─ TankController/...   ← orquestación principal (singleton)
│   ├─ PS3ControllerManager/...
│   ├─ CaterpillarController/...
│   └─ ...                  ← resto de módulos (Fuzzi, PID,
sensores...)
└─ data/
    ├─ index.html
    ├─ script.js
    └─ styles.css         ← UI web servida por SPIFFS y
AsyncWebServer
```

Esta disposición aporta varias ventajas:

1. **Modularidad y reutilización**: cada componente (motores, sensores, estrategias) vive en su propia carpeta bajo lib/ o include/, permitiendo pruebas y compilaciones aisladas.
2. **Evitar duplicación:** gracias al uso de interfaces (MotionStrategy) y clases auxiliares (CaterpillarController), podemos instanciar múltiples motores/sensores sin replicar código, reduciendo el footprint.
3. **Escalabilidad:** la lista de dependencias en platformio.ini es fácilmente extensible y versionable, y PlatformIO gestiona internamente las descargas y la resolución de conflictos.
4. **Transparencia y optimización:** el script de inspección de carpetas (print_tree.py) y la inspección de memoria al final del build facilitan la detección de crecimientos inesperados del firmware (por ejemplo, al añadir nuevas estrategias o eliminar la partición OTA).

5.4. Integración de FreeRTOS: Tareas, Prioridades, Núcleos e Interrupciones

FreeRTOS permite dividir el programa en tareas concurrentes, cada una con su propia prioridad. En nuestro código, por ejemplo, tenemos tareas como **movementTask** (para gestionar el controlador PS3 y el movimiento del tanque) y **sensorBroadcastTask**/**tacometerTask** (para lectura de sensores periódica). Cada tarea se ejecuta en un bucle infinito (for(;;)) y debe ceder el CPU periódicamente usando funciones como vTaskDelay() para no bloquear la ejecución de las demás. Las prioridades se asignan numéricamente (por ejemplo, prioridad 1, 2, 3, etc.): 0 es la más baja y el valor máximo viene dado por config MAX_PRIORITIES-1. El planificador de FreeRTOS siempre elegirá ejecutar la tarea de mayor prioridad que esté lista (es decir, que no esté bloqueada esperando algo). Si dos o más tareas del mismo nivel de prioridad están listas, el planificador les asigna tiempo de CPU en orden **round-robin** (rotatorio) para repartir el tiempo equitativamente. En otras palabras, tareas de igual prioridad se alternarán su ejecución por porciones de tiempo iguales (time-slicing), típicamente controladas por la interrupción de tick del sistema.
FreeRTOS es un sistema operativo de tiempo real preemptivo, lo que significa que el planificador puede interrumpir (preemptar) la tarea en curso para dar paso a una tarea de mayor prioridad en cuanto ésta esté lista, sin necesidad de que la tarea en ejecución “coopere”. Esto garantiza que las tareas críticas (con alta prioridad) respondan rápidamente. Por ejemplo, podríamos asignar a **movementTask** una prioridad más alta que la de **sensorBroadcastTask** si consideramos que reaccionar al movimiento del joystick es más importante que enviar datos de sensores. Así, cuando el PS3 está conectado y mueve el joystick (lo que eventualmente podría desencadenar una acción inmediata), **movementTask** tendrá preferencia de ejecución sobre la tarea de difusión de sensores. En cambio, la tarea de tacómetro (si tiene prioridad menor) sólo se ejecutará cuando ninguna otra más prioritaria esté lista, o en sus intervalos de espera. De este modo, cada tarea realiza su función a la frecuencia necesaria (en nuestro caso, **movementTask** verifica entrada ~50Hz y tacometerTask corre a ~20Hz según los vTaskDelay) sin bloquear a las demás.

5.4.1 Afinidad de núcleos y balanceo de carga

El ESP32 es un microcontrolador dual-core (dos núcleos de CPU) y la implementación de FreeRTOS en ESP-IDF soporta SMP (multiproceso simétrico). Esto significa que, por defecto, las tareas “libres” (sin afinidad fija) pueden ejecutarse en cualquiera de los dos núcleos, lo que incrementa el throughput y utilización de CPU disponibles. FreeRTOS se encarga de distribuir las tareas entre los diferentes núcleos para acelerar la ejecución en paralelo. En una tarea creada con xTaskCreate normal (sin especificar núcleo), le indicamos al planificador que puede colocarla en el núcleo que quiera (en sistemas multicore como ESP32). El planificador entonces decide en tiempo de ejecución qué núcleo ejecuta cada tarea libre, pudiendo moverla de un núcleo a otro entre contextos de planificador para balancear la carga. Este reparto dinámico (balanceo de carga) ocurre de forma transparente: si un núcleo está libre o menos cargado, puede ejecutar tareas pendientes que no están aferradas a otro núcleo.
Sin embargo, FreeRTOS también permite fijar la afinidad de núcleo de una tarea. Podemos crear tareas “pinneadas” a un núcleo específico usando xTaskCreatePinnedToCore() (indicando 0, 1 o tskNO_AFFINITY). Si una tarea está vinculada a un núcleo dado, solo podrá ejecutarse en ese núcleo; en cambio, una tarea sin afinidad (o con tskNO_AFFINITY) puede migrar y ejecutarse en cualquiera de los dos. Esto es útil en ciertas situaciones: por ejemplo, en ESP32 el núcleo 0 (PRO_CPU) suele usarse para tareas del sistema (WiFi, Bluetooth, radio) y el núcleo 1 (APP_CPU) suele reservarse para la lógica de la aplicación. De hecho, es una buena práctica reservar el núcleo 0 para WiFi/Bluetooth y ejecutar nuestras tareas de usuario en el núcleo 1 para no interferir con las tareas de radio, las cuales son críticas. Nuestro código probablemente sigue esta recomendación: es común crear las tareas del controlador PS3, tacómetro y difusión de sensores en el núcleo 1, dejando el núcleo 0 libre para la pila inalámbrica (especialmente si usamos WiFi o BT para el mando PS3). Fijar las tareas al núcleo 1 garantiza que incluso si el núcleo 0 se ocupa manejando WiFi/Bluetooth, nuestras tareas de control y sensores seguirán ejecutándose en paralelo en el otro núcleo.

El balanceo de carga en FreeRTOS (ESP32) ocurre cuando las tareas no tienen afinidad fija. El scheduler SMP evaluará qué tareas están listas y podrá asignarlas al núcleo que esté disponible. En general, cada núcleo corre una instancia del planificador y toma decisiones independientes. Esto permite, por ejemplo, que mientras el núcleo 0 está ocupado en una tarea, el núcleo 1 pueda ejecutar otra tarea lista, aprovechando los dos CPUs. En nuestro sistema, si dejamos alguna tarea sin fijar afinidad (tskNO_AFFINITY), FreeRTOS podría ejecutarla en cualquiera de los núcleos según convenga. No obstante, como mencionamos, en aplicaciones con WiFi/BT es común fijar afinidades manualmente para evitar conflictos. En resumen, la afinidad de núcleo nos da control sobre en qué CPU corre cada tarea, y el planificador SMP de FreeRTOS se encarga de repartir la carga cuando las tareas son libres, asegurando que ambas CPUs estén ocupadas ejecutando las tareas de mayor prioridad disponibles.

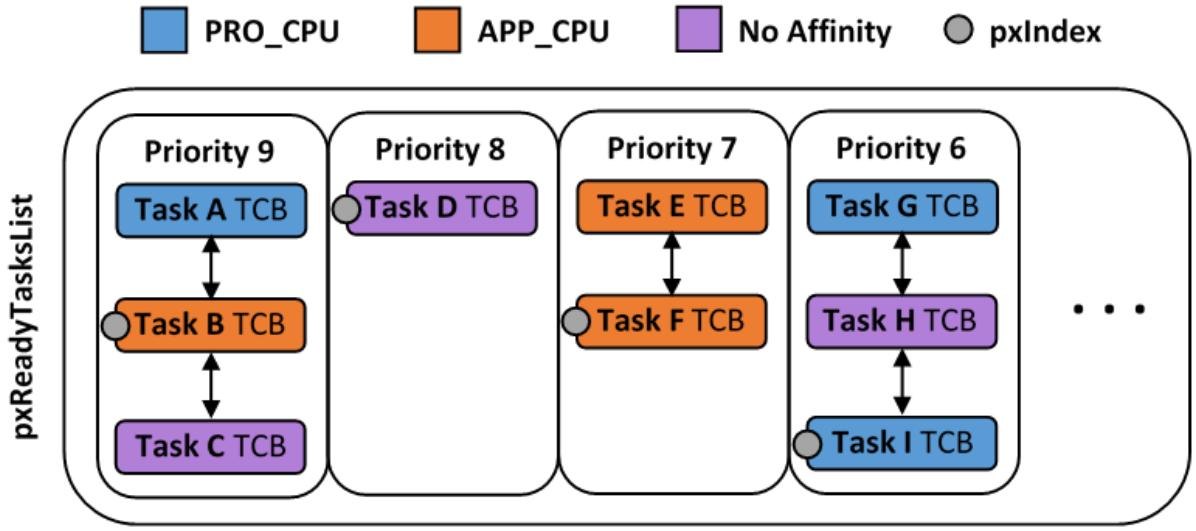


Figura 5.4.1 - Listas de tareas Ready (pxReadyTasksList) en FreeRTOS SMP sobre ESP32. Fuente: https://embarcados.com.br/esp32-lidando-com-multiprocessamento-parte-ii/

5.4.2 Gestión de interrupciones y reasignación de tareas

Las interrupciones (ISR, por sus siglas en inglés) juegan un papel importante en FreeRTOS y en la arquitectura de ESP32 de dos núcleos. FreeRTOS utiliza una interrupción de tick periódica (un temporizador de sistema) para llevar la cuenta del tiempo y para efectuar la planificación temporal. En el caso del ESP32 cada núcleo recibe una interrupción de tick cada 1 ms, que invoca al planificador en ese núcleo. Esto significa que cada milisegundo se fuerza una pequeña interrupción que "presta atención" al estado de las tareas en cada CPU: la interrupción de tick interrumpirá la tarea que esté ejecutándose en ese momento en cada núcleo y el sistema comprobará si debe cambiar de tarea. Si tras ese tick la tarea que estaba corriendo sigue siendo la de mayor prioridad disponible (es decir, ninguna tarea de mayor o igual prioridad se volvió lista mientras tanto), entonces continúa la misma tarea; de lo contrario (por ejemplo, si otra tarea de mayor prioridad despertó o se des-bloqueó), el planificador cambiará el contexto para que comience a ejecutarse la tarea más prioritaria. Este mecanismo garantiza la preempción temporal y que ninguna tarea acapara la CPU más allá del quantum configurado sin que el sistema reevalúe quién debe correr. En este fragmento de código se ve la inicialización de las tareas en el setup del programa así como la instanciación de **TankController**.

```
void setup() {
    Serial.begin(115200);
    delay(100);
    Serial.println("Iniciando sistema...");

    TankController::getInstance()->begin();
```

```
    TankController::getInstance()->setMotionStrategy(new
ManualControlStrategy());

    PS3ControllerManager::getInstance().begin();

    xTaskCreatePinnedToCore(
        movementTask, "MovementTask",
        MOVEMENT_TASK_STACK, nullptr,
        MOVEMENT_TASK_PRIO, &movementTaskHandle,
        1
    );

    xTaskCreatePinnedToCore(
        tacometerTask, "TacometerTask",
        TACOMETER_TASK_STACK, nullptr,
        TACOMETER_TASK_PRIO, &tacometerTaskHandle,
        0
    );
}
```

Además del tick, existen las interrupciones externas o de hardware (por ejemplo, interrupciones de GPIO, timers hardware, UART, etc.) que pueden ocurrir asíncronamente. En el ESP32, las interrupciones de hardware pueden asignarse a uno u otro núcleo. De hecho, según la documentación de ESP-IDF, al reservar/instalar una interrupción externa mediante las APIs del ESP32, ésta se asignará en el mismo núcleo desde donde se hace la asignación. Por ejemplo, si en nuestro código una tarea (ejecutándose en el núcleo 1) llama a attachInterrupt() para el tacómetro o algún sensor, el manejador ISR de ese periférico quedaría ligado al núcleo 1. (Las interrupciones internas del Xtensa, como las de su timer interno, están fijas a su propio core y no se pueden mapear al otro). Es importante notar que la desasignación (liberación) de una interrupción también debe hacerse desde el mismo núcleo al que fue asociada, aunque se permite habilitar/deshabilitar la interrupción desde el otro núcleo si fuera necesario. Debido a estas reglas, se recomienda fijar la afinidad (usar xTaskCreatePinnedToCore) de cualquier tarea que vaya a configurar interrupciones, para saber con certeza en qué CPU se instaló el servicio de interrupción. Si una tarea sin afinidad asignará una ISR, podría “migrar” a otro núcleo posteriormente, dejando la ISR en el núcleo original – una situación confusa para depurar o al intentar eliminarla más tarde. En resumen, FreeRTOS/ESP32 asigna las interrupciones de hardware al núcleo en el que se registran, por lo que nuestro código debe manejar cuidadosamente la afinidad cuando usamos ISR.

En cuanto a la interacción entre interrupciones e hilos (tareas), FreeRTOS provee mecanismos para sincronizar ambos mundos. Una ISR de hardware puede comunicarse con las tareas a

través de queues, semáforos, o notificaciones directas, usando las funciones especiales "FromISR" de FreeRTOS (por ejemplo, xSemaphoreGiveFromISR, xQueueSendFromISR, etc.). Cuando una interrupción despierta o desbloquea a una tarea de mayor prioridad, el kernel de FreeRTOS puede realizar un cambio de contexto inmediato al salir de la ISR – esto se conoce como yield from ISR. Así, la tarea importante se ejecuta justo después de atender la interrupción, minimizando la latencia. Por ejemplo, supongamos que nuestro tacómetro utiliza una ISR de hardware (cada vez que pasa un haz de luz infrarroja por el sensor, una interrupción incrementa un contador de pulsos). Esa ISR podría dar una notificación a una tarea de procesamiento de velocidad. Si dicha tarea tiene alta prioridad y estaba bloqueada esperando, en cuanto la ISR la libere, FreeRTOS planificará que se ejecute lo antes posible. En un sistema de dos núcleos, si la tarea despertada está afín al mismo núcleo donde ocurrió la interrupción, entonces ese núcleo podrá conmutar contexto a la tarea inmediatamente. Si la tarea es unpinned (sin núcleo fijo), podría ejecutarse en cualquiera de los núcleos disponibles. De hecho, el ESP32 soporta interrupciones inter-núcleo: un núcleo puede mandar una señal de interrupción al otro para notificar eventos, como por ejemplo para indicarle que reevalúe el planificador si una tarea de su afinidad se volvió lista en medio de una ISR en el núcleo opuesto.
En nuestro código, las llamadas a *vTaskDelay(pdMS_TO_TICKS(x))* dentro de las tareas utilizan internamente el tick de FreeRTOS para poner la tarea en estado "Blocked" por el número de ticks especificado. El núcleo 0 del ESP32 es el encargado de llevar la cuenta maestra de los ticks del sistema, de modo que cada vez que pasan, digamos, 20ms, la tarea **movementTask** será despertada (puesta en Ready) por el kernel para otra iteración. Mientras una tarea está bloqueada en delay, el planificador puede asignar ese CPU a ejecutar otras tareas que estén Ready. Así conseguimos que **movementTask** se ejecute a ~50 Hz y tacometerTask a 20 Hz de manera estable, sincronizadas por el tick. Si durante ese tiempo ocurriera una interrupción importante (por ejemplo, una interrupción del módulo Bluetooth que indique nueva información del control PS3), esta podría despertar a alguna tarea del stack BT o actualizar datos que **movementTask** leerá en su siguiente ciclo. Gracias a la prioridad alta que podríamos darle a **movementTask**, tan pronto como **movementTask** esté lista (tras su delay o un evento) se asegurará su ejecución prioritaria.

```
void movementTask(void* pvParameters) {
    auto& ps3 = PS3ControllerManager::getInstance();
    bool prevConnected = false;
    for (;;) {
        bool connected = ps3.isConnected();
        if (connected && !prevConnected) {
            Serial.println("PS3 Controller connected (detected in task)");
        } else if (!connected && prevConnected) {
            Serial.println("PS3 Controller disconnected");
        }
        prevConnected = connected;
```

```
        if (!connected) {
            vTaskDelay(pdMS_TO_TICKS(200));
            continue;
        }

        int16_t ly = Ps3.data.analog.stick.ly - 128;
        int16_t ry = Ps3.data.analog.stick.ry - 128;

        if (abs(ly) > 5) {
            TankController::getInstance()->executeMotionStrategy();
        }

        vTaskDelay(pdMS_TO_TICKS(20));
    }
}

void tacometerTask(void* pvParameters) {
    const TickType_t interval = pdMS_TO_TICKS(50);
    for (;;) {
        vTaskDelay(interval);
    }
}
```

En resumen, FreeRTOS maneja las interrupciones asignándose a los núcleos de forma controlada y cooperando con el planificador. El scheduler de FreeRTOS se activa periódicamente por una interrupción de tick en cada núcleo para re-evaluar qué tarea debe correr a continuación. Las interrupciones de hardware, por su parte, interrumpen temporalmente las tareas para atender eventos urgentes, pero al finalizar pueden dar lugar a un cambio de tarea si han despertado algo más prioritario. Nuestro código aprovecha este comportamiento: al usar FreeRTOS, podemos atender eventos rápidos (interrupciones) con baja latencia y, al mismo tiempo, mantener múltiples tareas (lectura de control, control de motores, medición de velocidad, envío de datos) ejecutándose quasi-paralelamente en los dos núcleos sin interferirse. Esto resulta en un sistema más responsivo y equilibrado, donde las interrupciones garantizan respuesta inmediata y el planificador se encarga de repartir el tiempo de CPU según las prioridades de nuestras tareas en el ESP32.

5.5. Locomoción: motores y controladores

Esta sección describe la cadena completa de tracción de Pampa, desde la generación de consignas de velocidad y giro hasta su ejecución segura sobre los puentes BTS7960. El control de motores se apoya en LEDC (PWM por hardware del ESP32) con un timer único para todas las salidas de tracción, garantizando frecuencia y fase comunes; cada motor utiliza dos canales

(RPWM/LPWM) y una lógica de dead-time e inversión segura para evitar corrientes de cruce. La dinámica de aplicación de par se modula mediante rampas de aceleración/frenado y límites de rate, reduciendo picos de corriente y vibraciones. Sobre esta base actúan los controladores de alto nivel: CaterpillarController, que encapsula motor + tacómetro y aporta estimación de velocidad/distancia con realimentación, y TankController, que orquesta ambos lados, coordina sensores y aplica consignas compuestas (avance, giro en sitio, trayectos con rampa). El lazo de control combina PWM abierto con correcciones cerradas (PID/Fuzzy según el caso) para igualar velocidades entre lados y sostener rumbo pese a tolerancias, carga y terreno.
La implementación está desacoplada en FreeRTOS: las tareas de locomoción ejecutan el ciclo de control con prioridad sobre telemetría y red, mientras que las interrupciones de tacómetro permanecen mínimas (contadores y marcas de tiempo). La odometría resultante alimenta a las estrategias de navegación y habilita funciones de mayor nivel como moveWithPID, rotateToAngle y moveDistanceWithRamp. Finalmente, se integran mecanismos de seguridad y robustez que garantizan que cualquier consigna de software se traduzca en un comportamiento físico predecible y seguro.

5.5.1 Control de motores mediante PWM (LEDC)

El término **PWM** (Pulse Width Modulation, o Modulación por Ancho de Pulso) hace referencia a una técnica de control ampliamente utilizada en sistemas embebidos para regular la potencia entregada a una carga eléctrica, particularmente motores, LEDs y otros actuadores. Esta técnica consiste en generar una señal cuadrada cuyo ciclo de trabajo (duty cycle) varía en función de la señal de control deseada.

En términos técnicos, el PWM se basa en una señal periódica que alterna entre estados lógicos alto (1) y bajo (0). El parámetro clave es el ciclo de trabajo (duty cycle), definido como el porcentaje del tiempo total del ciclo en que la señal permanece en estado alto:

$$Duty\ Cycle(\%) = \frac{t_{ON}}{T} \times 100$$

Donde:

- t_{ON}= duración del pulso en estado alto
- T = duración total del ciclo

Esta modulación permite controlar la potencia promedio entregada a una carga. Por ejemplo:

- Un duty cycle del 25% entrega un 25% de la energía posible por ciclo.
- Un duty cycle del 100% equivale a un encendido constante.

En sistemas con motores DC, este principio se traduce en una regulación efectiva de la velocidad de rotación.

5.5.2 Clase Motor (BTS7960): configuración de LEDC y canales PWM

Con el **BTS7960** (módulo IBT-2) cada motor requiere **dos entradas PWM**: `RPWM` y `LPWM`. La dirección se define activando una de ellas y dejando la otra en 0; la **velocidad** es el *duty* aplicado sobre la entrada activa. Para invertir el sentido, la clase reduce primero el duty a **cero**, inserta un **dead-time** de unos microsegundos y recién entonces conmuta el canal activo, evitando corrientes de cruce. En esta arquitectura, Pampa usa **LEDC** con un **único timer** compartido por **todos los canales de tracción** para garantizar misma frecuencia y fase base; se asignan **dos canales por motor** (uno para `RPWM`, otro para `LPWM`). La frecuencia efectiva recomendada está en el rango **15–20 kHz** (elimina zumbido audible sin castigar térmicamente al puente), con **8–10 bits** de resolución según el compromiso buscado entre granularidad a baja velocidad y estabilidad de conmutación. El mapeo desde la consigna de 0–255 se hace escalando al ancho de bits configurado, manteniendo coherencia con el resto del software.

La clase Motor encapsula la secuencia segura de cambio de estado: bajar duty del canal activo → esperar dead-time → actualizar líneas (RPWM/LPWM) → aplicar rampa de aceleración al nuevo duty. El frenado puede ser "a vela" (ambos PWM en 0, el motor queda en coast) o dinámico (ambos en 1, el puente cortocircuita el motor y disipa energía); en Pampa se reserva el frenado dinámico para emergencias por la corriente que puede circular. Si el módulo expone R_EN/L_EN, se mantienen habilitados y se usan solo RPWM/LPWM para mando; en caso de integrarlos a lógica de seguridad, la clase puede deshabilitarlos para corte rápido ante sobrecarga.

Operativamente, el **actualizador** corre en una tarea FreeRTOS dedicada: aplica rampas, límites de duty y *rate limiting*, y empuja los nuevos valores LEDC de manera no bloqueante. Se asume **alimentación** y **retornos** de potencia robustos, con desacoples cerca del módulo y cables a motor trenzados para reducir EMI. El BTS7960 incorpora **protección térmica y por sobrecorriente**; aun así, la configuración de PWM y rampas debe evitar pulsos largos a alto duty con el motor trabado. Respecto a niveles lógicos, la mayoría de módulos IBT-2 aceptan **3.3 V** del ESP32, pero se valida en banco (o se usa *level shifting* si el lote requiere 5 V). En síntesis: dos canales PWM por motor, un **solo timer LEDC** para toda tracción, control cuidadoso de inversión con **dead-time**, y elección de frecuencia/resolución pensada para el perfil térmico y de ruido del **BTS7960**.

$$V_{efectiva} = V_{suministrada} \times \frac{Duty\ Cycle}{100}$$

5.5.3 CaterpillarController : encapsulación motor + tacómetro

El **CaterpillarController** es una entidad que unifica en una sola interfaz el control de un motor de tracción y su sensor de retroalimentación (tacómetro). Esta encapsulación simplifica la lógica de control y permite al resto del sistema manipular la locomoción y medir su rendimiento sin interactuar directamente con los componentes de bajo nivel.

5.5.3.1 Propósito y rol en la arquitectura

Su propósito principal es servir como **punto de acceso único** para las operaciones críticas de locomoción, combinando la capacidad de:

- Ordenar un movimiento (dirección y velocidad).
- Mantener un movimiento constante (persistente).
- Obtener retroalimentación en tiempo real de velocidad y distancia.
- Gestionar el estado interno del motor y el contador de pulsos.

Dentro de la arquitectura, cada oruga (lado izquierdo o derecho del robot) tiene su propio **CaterpillarController**, lo que permite un control independiente y balanceado en estrategias de movimiento avanzadas.

5.5.3.2 Integración con Motor y Tacometer

El **CaterpillarController** recibe referencias a un objeto **Motor** y un objeto **Tacometer** en su constructor:

- **Motor**: se encarga de la modulación PWM y dirección, soportando modos de ejecución normal y persistente.
- **Tacometer**: mide los pulsos provenientes del encoder, calculando velocidad lineal (cm/s) y distancia recorrida.

Ambos objetos son administrados por el **TankController**, que los crea y enlaza al inicializarse. Esto asegura que **CaterpillarController** no gestione la memoria de estos componentes, evitando dobles liberaciones.

5.5.3.3 Estrategias de persistencia de movimiento

El motor implementa dos tareas en **FreeRTOS**:

1. **motorTask**: procesa comandos puntuales recibidos por cola (`xQueueOverwrite`), ejecutándose cuando se requiere un cambio de estado inmediato.

```
1. void Motor::motorTask(void* param) {
2.  Motor* self = static_cast<Motor*>(param);
3.     MotorCommand cmd;
4.     for (;;) {
5.         if (xQueueReceive(self->commandQueue, &cmd, portMAX_DELAY) ==
   pdTRUE) {
6.             self->setDirection(cmd.dir);
7.             self->setSpeed(cmd.speed);
8.             ledcWrite(self->fwdCh, 0);
9.             ledcWrite(self->revCh, 0);
10.            if (cmd.dir == FORWARD) {
11.                ledcWrite(self->fwdCh, cmd.speed);
12.            } else if (cmd.dir == BACKWARD) {
13.                ledcWrite(self->revCh, cmd.speed);
14.            }
15.            xEventGroupSetBits(self->movementEventGroup,
   MOVEMENT_DONE_BIT);
16.        }
17.    }
18.}
```

2. **motorPersistentMovementTask**: mantiene activo el movimiento persistente mediante un bucle que refresca la señal PWM cada 50 ms mientras persistentMovementActive esté en true.

```
3. void Motor::motorPersistentMovementTask(void* param) {
4.     Motor* self = static_cast<Motor*>(param);
5.     for (;;) {
6.         while (self->persistentMovementActive) {
7.             MotorDirection dir = self->getDirection();
8.             int speed = self->getSpeed();
9.             ledcWrite(self->fwdCh, 0);
10.            ledcWrite(self->revCh, 0);
11.            if (dir == FORWARD) {
12.                ledcWrite(self->fwdCh, speed);
13.            } else if (dir == BACKWARD) {
14.                ledcWrite(self->revCh, speed);
15.            }
16.            vTaskDelay(pdMS_TO_TICKS(50));
17.        }
18.        vTaskDelay(pdMS_TO_TICKS(50));
19.    }
20.}
```

5.5.3.4 Medición de velocidad y distancia

La medición se realiza a través del tacómetro asociado:

- **calculateSpeed()**: devuelve la velocidad lineal en cm/s, calculada a partir de las RPM del eje y el perímetro de la rueda.

```
float CaterpillarController::calculateSpeed() {
    return tacometer->calculateLinearSpeedCMs();
}
```

Este método se encuentra incluido en la clase **Tacometer()** de manera que se puede acceder a él mediante la composición de objetos

```
float Tacometer::calculateLinearSpeedCMs() const {
    return (currentRPM_ * WHEEL_CIRCUMFERENCE_CM) / 60.0f;
}
```

WHEEL_CIRCUMFERENCE_CM es una variable introducida en la clase **Configuration.h** lo que permite modificarla en caso de que sea necesario.

- **calculateDistance()**: devuelve la distancia recorrida acumulada en cm, basada en el conteo total de pulsos y la constante CM_PER_PULSE.

```
float CaterpillarController::calculateDistance() {
    return tacometer->getCurrentDistance();
}
```

Estos valores son esenciales para el control PID/Fuzzy y para estrategias que dependen de precisión en desplazamientos.

Esta separación permite que el control manual (rápido y responsivo) y el control autónomo (fluido y constante) coexistan en la misma arquitectura.

5.5.3.5 Medidas de protección y sincronización con mutex

En la clase Motor, el acceso y modificación de variables críticas (**currentDirection**, **currentSpeed**) se realiza mediante:

- portENTER_CRITICAL(&stateMutex) y portEXIT_CRITICAL(&stateMutex), asegurando exclusión mutua a nivel de interrupciones.

```
1. MotorDirection Motor::getDirection() const {
2.     portENTER_CRITICAL(&stateMutex);
3.     MotorDirection dir = currentDirection;
4.     portEXIT_CRITICAL(&stateMutex);
5.     return dir;
6. }
7.
8. int Motor::getSpeed() const {
9.     portENTER_CRITICAL(&stateMutex);
```

```
    int sp = currentSpeed;
    portEXIT_CRITICAL(&stateMutex);
    return sp;
}

void Motor::setSpeed(int speed) {
    portENTER_CRITICAL(&stateMutex);
    currentSpeed = speed;
    portEXIT_CRITICAL(&stateMutex);
}

void Motor::setDirection(MotorDirection dir) {
    portENTER_CRITICAL(&stateMutex);
    currentDirection = dir;
    portEXIT_CRITICAL(&stateMutex);
}
```

Esto protege la integridad de los datos en un entorno con múltiples tareas y posibles ISR (por ejemplo, actualizaciones desde tacómetros o controladores externos) sobre todo cuando se ajusta rápidamente por ejemplo cuando se está implementado la estrategia de PID para la compensación de velocidades en los motores.

5.5.4. TankController: orquestación y singleton

El **TankController** es la pieza central del sistema de locomoción, el encargado de orquestar y coordinar todos los elementos que intervienen en el movimiento del robot. A través de esta clase se integran los motores, tacómetros, controladores auxiliares como el giroscopio y el controlador difuso, así como las estrategias de movimiento que definen el comportamiento general. Su diseño sigue el patrón Singleton, lo que garantiza que en todo el sistema exista una única instancia coherente y sincronizada, evitando problemas de control simultáneo. Este módulo no solo gestiona la ejecución de las órdenes de movimiento, sino que también valida la disponibilidad de los componentes, asegura una inicialización segura y proporciona herramientas para depuración y diagnóstico en tiempo real.

5.5.4.1. Rol central en la arquitectura

El **TankController** actúa como el cerebro que toma las decisiones de locomoción y las traduce en comandos específicos para cada lado del robot. No se limita a transmitir órdenes, sino que interpreta las estrategias activas, mide el estado de los sensores y aplica ajustes para lograr movimientos precisos y estables. Esto le permite coordinar elementos tan diversos como el control por PWM persistente, el mantenimiento de velocidad en línea recta o la corrección de trayectoria con el giroscopio. En la práctica, cualquier acción que implique desplazamiento físico pasa por este punto de control.

5.5.4.2. Patrón Singleton y control de instancia

La implementación de TankController sigue un patrón Singleton seguro para entornos concurrentes. Esto significa que solo puede existir una instancia activa en todo el sistema y que su creación está protegida por un mutex para evitar condiciones de carrera. El patrón se refuerza con un mecanismo de double-checked locking, que reduce el costo de sincronización una vez que la instancia ha sido creada. Además, se incluyen banderas como **isFullyReady** e **initializationFailed** para controlar el estado de inicialización y evitar el uso de un controlador incompleto o fallido.

5.5.4.3. Inicialización y validación de componentes

Durante su construcción, el TankController se encarga de instanciar y poner en marcha todos los módulos críticos para la locomoción: motores, tacómetros, controladores de oruga, giroscopio y controlador difuso. Después, mediante el método begin(), verifica que cada uno de estos elementos esté correctamente inicializado antes de declararse listo. Este proceso incluye mensajes detallados por consola, que ayudan a identificar rápidamente cualquier componente que no responda, y una función **validateComponents()** que ofrece un diagnóstico estructurado del estado de cada subsistema.

```
1.  bool TankController::validateComponents() {
2.      bool allValid = true;
3.
4.      Serial.println("=== COMPONENT VALIDATION ===");
5.
6.      if (!leftMotor) {
7.          Serial.println(" Left Motor: NULL");
8.          allValid = false;
9.      } else {
10.         Serial.println(" Left Motor: OK");
11.     }
12.
13.     if (!rightMotor) {
14.         Serial.println(" Right Motor: NULL");
15.         allValid = false;
16.
17. // resto de sistemas ///
18.     if (!fuzziController) {
19.         Serial.println(" FuzziController: NULL");
20.         allValid = false;
21.     } else {
22.         Serial.println(" FuzziController: OK");
23.     }
24.
25.     Serial.println("ℹ  Ultrasonic Sensors: DISABLED");
26.
27.     Serial.printf("Motion Strategy: %s\n", motionStrategy ? "ASSIGNED" : "NULL");
28.
```

```
29.     Serial.println("============================");
30.
31.     return allValid;
32. }
```

5.5.4.4. Gestión de estrategias de movimiento

El TankController no codifica directamente cómo debe moverse el robot, sino que delega esa lógica en un objeto de tipo **MotionStrategy**. Esto le permite cambiar dinámicamente el comportamiento del vehículo, pasando de un control manual a una navegación autónoma sin alterar el código base. La asignación de estrategias está protegida por validaciones de estado y por registros de cambios, de manera que cada transición quede documentada en la consola y se eviten asignaciones erróneas cuando el sistema no está listo.

5.5.4.5. Coordinación de CaterpillarControllers y sensores

Cada lado del robot está gobernado por un **CaterpillarController**, que a su vez encapsula un motor y un tacómetro. El TankController coordina ambas unidades para lograr desplazamientos simétricos o diferenciados según la maniobra. Al mismo tiempo, integra datos de sensores como el giroscopio y, en versiones completas, los sensores ultrasónicos, para adaptar el movimiento a las condiciones del entorno. Esta coordinación asegura que las órdenes de giro, avance o retroceso se ejecuten de forma coherente y sincronizada en ambos lados del chasis.

5.5.4.6. Control avanzado de locomoción

El **TankController** incorpora tres métodos principales para control avanzado del desplazamiento: **moveWithPID**, **rotateToAngle** y **moveDistanceWithRamp**. Cada uno responde a necesidades distintas, desde mantener velocidad constante con retroalimentación, hasta realizar giros precisos o desplazamientos con perfiles de aceleración y frenado optimizados. Estas funciones combinan el uso de los atributos internos (como referencias a **CaterpillarController** izquierdo y derecho, el **FuzziController**, el **Gyroscope** y los parámetros de configuración de velocidad) con cálculos matemáticos y temporización precisa para garantizar un comportamiento estable y repetible.

5.5.4.6.1. moveWithPID

Este método utiliza el controlador difuso (**fuzziController**) para calcular ajustes de PWM que mantengan la velocidad objetivo (**targetSpeed**) de forma simétrica en ambos lados del robot. Se apoya en la lectura de velocidad real de cada CaterpillarController mediante el método **calculateSpeed**() y en el ajuste individual de cada motor a través de **MotorMotion**. La lógica se centra en equilibrar las velocidades izquierda y derecha, compensando automáticamente diferencias por desgaste o carga. Aunque la base de cálculo es PID, la implementación actual está encapsulada en el FuzziController, lo que permite modularidad y una futura sustitución de la técnica de control sin alterar la estructura de TankController.

5.5.4.6.2. rotateToAngle

En este modo, el objetivo es alinear el robot a un ángulo específico definido por el parámetro degrees. Se reinicia el giroscopio **(gyroscope->resetAngle())** para tener un punto de referencia y se calcula en cada ciclo la diferencia angular entre el objetivo y la orientación actual **(getAngleDeg)**. Esa diferencia se normaliza con **AngleUtils::normalizeAngleDiff** para evitar saltos abruptos en el rango ±180°. Si el error supera un umbral definido **(THRESHOLD_DEG)**, se aplica un **PWM** calculado con un **PIDController** específico para giros, enviando comandos opuestos a las orugas izquierda y derecha (una avanza y la otra retrocede) para girar sobre el eje central. El ciclo se repite hasta que el error angular está dentro del margen de tolerancia, momento en el que se detienen los motores y se restablece la referencia del giroscopio.

5.5.4.6.3. moveDistanceWithRamp

Este método desplaza el robot una distancia determinada (**distanceCm**) aplicando un control de aceleración y frenado progresivo que evita cambios bruscos en la tracción. La clave está en el cálculo del PWM dinámico a lo largo del recorrido mediante una función cuadrática, que permite que la aceleración inicial y la desaceleración final sean suaves y controladas.
El cálculo se basa en dos fases:

1. Aceleración inicial: cuando la distancia recorrida xxx es menor que **accelEnd** (10% del recorrido total), el PWM se incrementa según:

$$PWM(x) = Vmin + (Vmax - Vmin) \cdot (\frac{x}{accelEnd})^2$$

2. Desaceleración final: cuando x supera **decelStart** (90% del recorrido total), el PWM disminuye con:

$$PWM(x) = V_{min} + (V_{max} - V_{min}) \cdot (\frac{D\ - x}{D - \ decelStart})^2$$

donde:

- V_{min} y V_{max} son la velocidad mínima y máxima permitida.
- X es la distancia recorrida en cm.
- D: es la distancia total (distanceCm).

En la zona intermedia, cuando $\boldsymbol{accelEnd \leq x \leq decelStart}$, el PWM se mantiene constante en V_{max}

Esta formulación matemática hace evidente la complejidad del control: no es un simple incremento lineal, sino una curva parabólica que optimiza la entrega de potencia y el comportamiento dinámico del robot. De esta forma, el sistema reduce el riesgo de

deslizamiento al arrancar, evita sobrecargas mecánicas y mejora la precisión al frenar cerca del objetivo.

5.5.4.7 Protección, depuración y manejo de memoria

El TankController incluye utilidades de diagnóstico que permiten monitorear el estado de la memoria del sistema y detectar posibles fugas o fragmentación excesiva. Funciones como **printMemoryStatus()** y **logHeapUsage()** muestran información detallada del uso de la RAM y de la PSRAM, emitiendo advertencias cuando el nivel disponible cae por debajo de un umbral seguro. En cuanto a protección, además del uso de mutex en la creación de la instancia, el controlador detiene los motores antes de su destrucción para evitar comportamientos indeseados si la instancia se libera en medio de un movimiento.

5.5.4.8 Importancia en la escalabilidad y mantenimiento

La existencia de un punto único de control como el **TankController** simplifica enormemente la evolución del sistema. Al centralizar la lógica de orquestación y delegar el comportamiento en estrategias intercambiables, el robot puede adquirir nuevas capacidades (como navegación por visión o evasión de obstáculos) sin que eso implique reescribir el código de control de motores. Además, el patrón Singleton y la validación exhaustiva de componentes hacen que el sistema sea más robusto y fácil de mantener, reduciendo la probabilidad de fallos críticos en campo.

5.6. Gestión y procesamiento de sensores

En Pampa, la capa de sensores se diseña como un **pipeline asíncrono y desacoplado** que prioriza la coherencia temporal y la trazabilidad de los datos frente a las restricciones del ESP32. La captura se realiza cerca del hardware (ISR para tacómetros y temporización precisa en ultrasonidos; lectura periódica por I²C para el giróscopo), mientras que la **normalización, filtrado y fusión** se resuelven en tareas de FreeRTOS con **colas y buffers circulares** para aislar jitter y picos de carga. Cada muestra se **sella con tiempo** (tick monotónico) y se estandariza en **unidades físicas** antes de publicarse al resto del sistema, aplicando técnicas ligeras (promedio móvil, mediana u outlier rejection) que preservan eventos rápidos sin bloquear la locomoción. Este enfoque reduce la latencia perceptible de control, evita trabajo pesado dentro de interrupciones y facilita la **consistencia entre módulos** (motores, navegación y web). En las subsecciones se detalla:

- La clase **Tacometer** y sus cálculos de **velocidad y distancia** a partir de pulsos (incluido el factor CM_POR_PULSO);
- La clase **UltrasonicSensor**, con su temporización de trigger/echo y estrategias de reintento/filtrado sin bloqueo; y
- El **giróscopo** (lectura I²C, calibración de offsets y esquema de actualización periódica) como referencia de orientación para la navegación.

5.6.1 Tacómetro

El tacómetro mide el avance a partir de **eventos digitales** generados por un disco ranurado y un sensor óptico. En Pampa se capturan esos eventos con **ISR muy cortas** (solo incrementan contadores y registran tiempo) y se procesa la cinemática en una tarea periódica para no bloquear la locomoción. La magnitud base es la **distancia por pulso**, `que depende del diámetro efectivo de la rueda D, del número de ranuras N, de si contamos uno o dos flancos por ranura E∈{1,2} y de la` **relación de reducción** G (vueltas de motor por vuelta de rueda). Como el disco está en el eje del motor, G es la relación del tren de engranajes; si está en la rueda, G=1.

$$CM\_POR\_PULSO = \frac{\pi .D}{N .E .G} \quad \text{(Con D en cm)}$$

A partir de esa constante, la **distancia acumulada** se obtiene como $S = n_{pulsos} \times CM\_POR\_PULSO$. La **velocidad lineal** puede estimarse por dos vías:

- **frecuencia instantánea**, usando el intervalo entre pulsos Δt_k

$$V_k = \frac{CM\_POR\_PULSO}{\Delta t_k}$$

que ofrece baja latencia pero es ruidosa a bajas RPM;

- **Conteo en ventana** de duración T

$$v = \frac{\Delta n \cdot CM\_POR\_PULSO}{T}$$

más estable pero con mayor retardo. En práctica combinamos ambos: la tarea de procesamiento aplica un **suavizado exponencial** o un **promedio móvil** sobre Vk y valida con la estimación en ventana para mitigar outliers.

El uso de interrupciones es clave frente a pulseIn(). pulseIn() bloquea la CPU esperando flancos, degrada la concurrencia y pierde eventos si se atienden otros sensores; en cambio, con ISR registramos marcas de tiempo (p. ej., micros()) y un contador atómico, manteniendo el trabajo pesado fuera de la ruta de interrupción. Para robustez:

1. rechazo de rebotes por tiempo mínimo entre pulsos (Δt_{min}), coherente con la velocidad máxima mecánica.
2. manejo de wrap-around de contadores y del temporizador usando aritmética modular.
3. dirección tomada de las señales del H-bridge (o de un encoder cuadratura si estuviera disponible) para sumar o restar pulsos con sentido físico.

En el plano de **calibración**, el D efectivo debe medirse **bajo carga** y sobre el terreno real, ya que el aplastamiento del neumático y el **deslizamiento** introducen sesgos. Asimismo, si el disco está en el motor, G debe ser el **ratio real** (no solo el nominal), ya que holguras y

desalineaciones pueden alterar el recuento efectivo. Finalmente, empleamos dos tacómetros (uno por cada oruga/rueda motriz) y usamos su feedback para **igualar velocidades** y compensar tolerancias entre motores: el controlador de velocidad ajusta el **PWM** objetivo de cada lado para minimizar el error $e_v = V_{izq} - V_{dere}$sin saturar ni introducir oscilaciones.

5.6.2 UltrasonicSensor

El módulo UltrasonicSensor mide distancia por **tiempo de vuelo** (ToF): se emite un pulso de 10 µs en el pin *trigger* y se mide en el pin *echo* cuánto tarda en volver el frente reflejado. La conversión a distancia parte de t_{echo} (en µs) y de la **velocidad del sonido** v(T) , aproximada por $v(T) \approx 331.3 + 0.6T\ m/s$ (con T en °C). En centímetros: $v_{cm/\mu s} = 10^{-4}\, v(T)$. Por tanto,

$$d_{cm} = \frac{v_{cm/\mu s} \cdot t_{echo}}{2}$$

(el factor 2 compensa ida y vuelta). Para lecturas coherentes se aplica además un offset de montaje (distancia desde el plano del transductor al borde del chasis), que se resta para expresar la medida en el marco del robot.
Desde el punto de vista temporal, una lectura puede hacerse con pulseIn() (bloqueante con timeout) o mediante una máquina de estados no bloqueante que programa el trigger, espera el flanco de echo y cierra la medición con micros() en una tarea de FreeRTOS. En Pampa, el sensor se integra al pipeline de sensores: el disparo y la captura se encapsulan en la clase y el cálculo/normalización se realiza en una tarea periódica, preservando la latencia del control de motores. El uso de timeouts acota el peor caso y evita colgar el hilo si no hay eco.

5.6.3 Gyroscope (MPU6050)

El MPU6050 aporta **tasa de giro** (ω, en °/s) y **aceleración** (a, en g) sobre I²C, que en el ESP32 viaja por **SDA/SCL** con *pull-ups* (típico 4.7–10 kΩ) y velocidades de 100 kHz (Standard) o 400 kHz (Fast). En Pampa se prioriza 400 kHz para reducir el jitter de muestreo, manteniendo el bus corto y apantallado para evitar inyecciones de ruido de los motores. La dirección del MPU6050 es **0x68** (o **0x69** si AD0=1). La lectura se organiza en **rafagas** (lectura múltiple con *repeated start*) para capturar registros contiguos de acelerómetro, temperatura y giróscopo con un único sello de tiempo. Opcionalmente, el pin **INT** del MPU6050 puede usarse como "**data ready**"; la ISR marca "muestra disponible" y la tarea periódica realiza la lectura I²C, evitando trabajo pesado en interrupción. El **DLPF** interno (filtro pasa-bajo) se configura para atenuar vibración mecánica; valores en torno a 20–42 Hz suelen equilibrar latencia y ruido.

La **calibración** comienza con la determinación de **offsets estáticos**: con el robot inmóvil, se promedian varios cientos de muestras para estimar el **sesgo** del giroscopio ω (°/s) y el **vector gravedad** en el marco del sensor. El giróscopo reporta ω_{raw} en cuentas (LSB); al convertir a unidades físicas se aplica la sensibilidad Sg según rango elegido:

$$\omega[\frac{s}{\circ}] = \frac{\omega_{raw}}{S_g}, S_g \in \{131,65.5,32.8,16.4\}$$

para ±250/ ±500/ ±1000/ ±2000 °/s respectivamente. El **offset** se resta muestra a muestra: $\omega_c = \omega - b_\omega$. El acelerómetro se convierte con su sensibilidad (p. ej., 16384 LSB/g en ±2 g) y se usa para **referenciar la inclinación**:

$$\theta_{acc,pitch} = arctan\,2(a_x, \sqrt{a_y^2 + a_z^2}), \theta_{acc,roll} = arctan2(-a_y, a_z)$$

(suponiendo eje z apuntando "arriba" en reposo). Dado que el giróscopo **deriva** con la temperatura y el tiempo, se mantiene una **compensación térmica ligera**: el registro de temperatura del MPU se usa para ajustar bω(T) con una pendiente estimada en banco (modelo lineal) o, de forma práctica, se re-promedia el sesgo cuando el sistema detecta reposo.

La **lectura periódica** se integra en una tarea de FreeRTOS a frecuencia fija de 100 Hz con **sello temporal** tΔt estable. La orientación se propaga integrando la velocidad angular:

$$\theta t = \theta t - \Delta t + \omega c \Delta t$$

(en rad/° según convención), operación válida para pequeños tΔt. Para contener la deriva, se aplica un filtro complementario con el acelerómetro:

$$\theta_t = \alpha(\theta_t - \Delta_t + \omega_c \Delta t) + (1 - \alpha)\theta_{acc}$$

con α cercano a 0.98–0.995. En tres ejes, la implementación robusta usa cuaterniones o matrices de rotación (evita *gimbal lock*) y, si se habilita, el **DMP** interno del MPU6050 puede entregar cuaterniones a ~100 Hz; sin embargo, en Pampa se prioriza control explícito del pipeline (tiempos, filtros y *fusion*) para mantener **trazabilidad** y coherencia con el resto de sensores.

Un punto crítico es la **alineación de marcos**: el eje del sensor rara vez coincide exactamente con el del chasis. Se mide la **rotación fija** entre el marco del sensor y el del robot y se corrigen las lecturas con una matriz RsrR_{sr}Rsr. La orientación final se publica en el **marco del robot**, lo que simplifica el uso por las estrategias de navegación (alineamiento a rumbo, giros controlados y estabilización).

5.7. Estrategias de navegación (Strategy Pattern)

Uno de los objetivos clave de **Pampa** era dotar al robot de la capacidad de cambiar su modo de funcionamiento "en caliente" sin modificar el núcleo del programa. Para lograrlo, los componentes de bajo nivel (motores, tacómetros, sensores) se implementan como clases independientes y desacopladas, responsables únicamente de ejecutar acciones puntuales (girar

rueda, leer pulsos). La lógica de más alto nivel (la "estrategia de navegación") necesita decidir cuándo y cómo usar estos componentes, pero no debe conocer sus detalles internos.

5.7.1 MotionStrategy: interfaz base

La interfaz MotionStrategy abstrae por completo la lógica de navegación y define un único contrato:

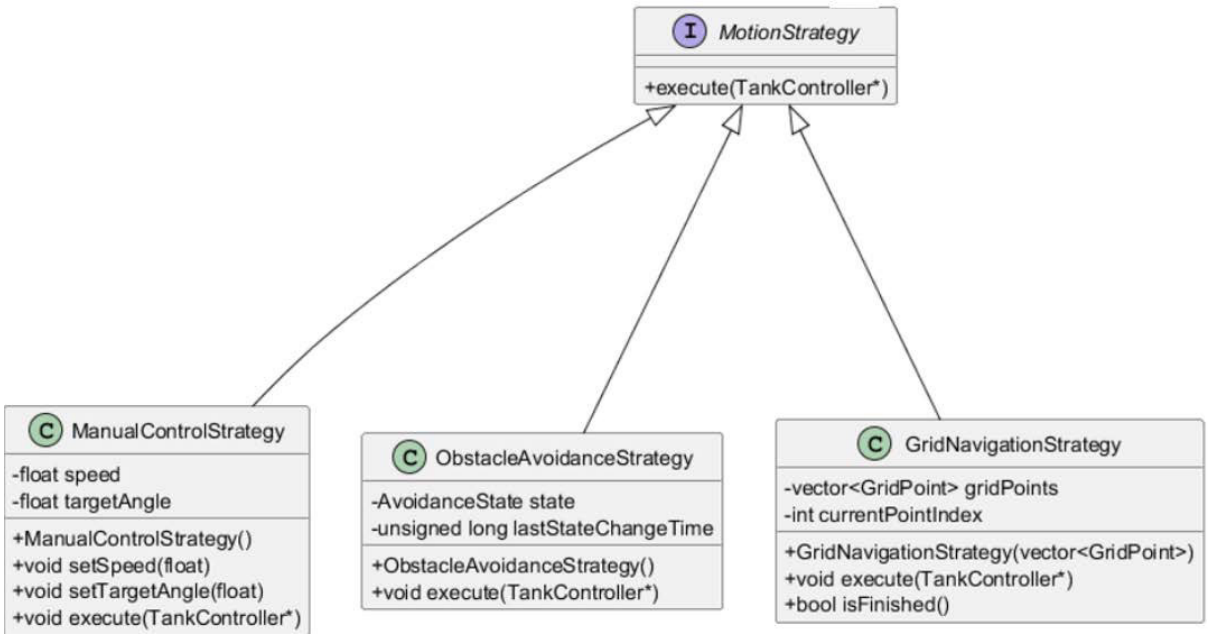


Figura 5.7.1 - Interfaz MotionStrategy y derivadas
Fuente: Propia

- Responsabilidad: recibir un objeto TankController y emitir, de manera estandarizada, los comandos de motor y sensor necesarios para desplazar el robot.
- Desacoplamiento: elimina toda referencia directa a hardware o algoritmos concretos del núcleo de control.
- Flexibilidad: permite intercambiar "en caliente" cualquier estrategia de navegación sin modificar ni recompilar el controlador principal.

En el diagrama UML, MotionStrategy aparece como interfaz padre de clases como ManualControlStrategy, ObstacleAvoidanceStrategy y GridNavigationStrategy. Cada una:

1. Lee sólo sus propias entradas (joystick, ultrasonidos, malla de puntos…).
2. Calcula internamente dirección, velocidad o ángulo.
3. Invoca a tank->… para ejecutar el movimiento.

Este enfoque garantiza que añadir o cambiar un algoritmo de navegación implique únicamente crear o sustituir una clase que implemente MotionStrategy, manteniendo el resto del sistema completamente intacto.

5.7.2. ManualControlStrategy

La primera estrategia de control que desarrollamos en el proyecto **Pampa** consistió en implementar un esquema de manejo manual de los motores y la dirección, aprovechando un joystick de PS3 como interfaz de usuario. El objetivo principal fue validar la respuesta dinámica de los motores DC y poner a prueba tanto el puente H L298N como el PWM en un entorno controlado antes de abordar algoritmos de navegación autónoma. Para ello, utilizamos un **ESP32** con conectividad Bluetooth: fue necesario modificar la dirección MAC del mando para que el microcontrolador lo detectara y emparejara automáticamente en cada arranque. Nos apoyamos en la librería **PS3ControllerHost**, sobre la que construimos una clase derivada que centraliza la normalización de métodos y simplifica el acceso a datos de los ejes y botones.

La lógica de control mapea los valores analógicos del joystick directamente a señales PWM. Cada movimiento del eje X o Y se traduce en un valor entre el **mínimo operativo** (el PWM más bajo que genera torque suficiente) y 255 (PWM máximo). De este modo, al empujar el stick hacia adelante, la velocidad aumenta de forma lineal; al retroceder, disminuye. Uno de los desafíos más relevantes fue el **mapeo correcto de los ejes del joystick**: sin una calibración precisa, los valores crudos (0–255) no se traducen linealmente a PWM, provocando zonas muertas inesperadas o saltos bruscos en la velocidad. En las siguientes secciones describiremos las pruebas realizadas para ajustar esta conversión y garantizar un control suave y predecible.

5.7.2.1. Problema de desbordamiento al mapear valores crudos

Al leer directamente los datos del joystick (0 ... 255) en un tipo `int8_t` (–128 ... 127), todos los valores superiores a 127 “saltaban” hacia el rango negativo, provocando saltos erráticos en la lectura. Durante las pruebas observamos que, al pasar de 127 a 128, el valor pasaba a –128, introduciendo un pico inesperado en la curva. La solución se encontró tras varias modificaciones de código en pensar cómo se almacena el valor leído y que representa realmente que esté almacenado en un valor de tipo `int8_t`.

A nivel de arquitectura de CPU, la mayoría de los microcontroladores (incluido el ESP32) trabajan internamente con registros de 32 bits para las operaciones enteras. Cuando lees el valor bruto del joystick (0 ... 255) en un tipo de datos de 8 bits firmado (`int8_t`), el rango nativo (–128 ... 127) no cubre valores mayores a 127. Así que, por ejemplo, un raw = 200 “se interpreta” como 200–256 = –56, provocando un “rollover” que aparece como un salto errático. Al almacenar primero ese raw en un entero de 32 bits (`int` o `uint16_t`), su representación binaria de 8 bits se “extiende” sin signo (o con signo conservado), pero con suficiente rango para cubrir 0 ... 255. El procesador ya no genera overflow ni wrap-around, y todas las operaciones posteriores (restas, multiplicaciones, divisiones) se hacen sobre un tipo de 32 bits, asegurando precisión y linealidad.

La solución fue simplemente castear el valor leído a un int y mediante un ternario asignamos un pequeño “punto muerto” por si se detecta un valor medido en reposo que en la práctica podría ser despreciable

```
int PS3ControllerManager::getRawRX() const {
    int v = int(Ps3.data.analog.stick.rx);
    return (abs(v) <= deadzone) ? 0 : v;
}
```

Como el valor que se lee de un analogico en realidad se puede interpretar como una coordenada puesto que tenemos un valor de LX y LY podemos mediante una pequeña transformación matemática obtener el ángulo obtenido de la intercesión de los dos puntos

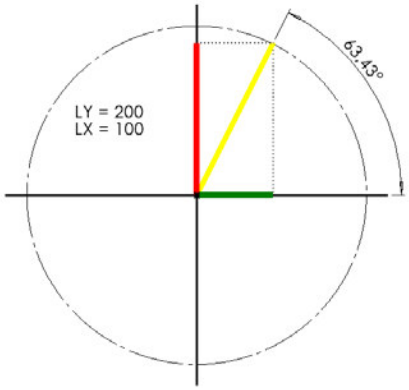


Figura 5.7.2.1 - Representacion vectorial de los parametros leidos por los controles una vez convertidos a valores int de 32 bits
Fuente: Propia

```
float PS3ControllerManager::calculateJoystickAngle() const {

    int sx = getRawRX();
    int sy = getRawRY();
    if (sx == 0 && sy == 0) {
        return 0.0f;
    }

    float angDeg = atan2f((float)sy, (float)sx) * 180.0f / PI;
    if (angDeg < 0) {
```

```
        angDeg += 360.0f;
    }
    return angDeg;
}
```

Para poder calcular el ángulo hacemos lo siguiente :
1. Obtención de las coordenadas en enteros de 32 bits:

$$Sx, Sy \in \{0, \ldots, 255\}$$

2. Conversión a punto flotante y uso de la función atan2 para obtener el ángulo θ en radianes, donde

$$\phi = atan2(Sx, Sy) \in (-\pi, \pi]$$

3. Transformación a grados:

$$\alpha = \phi \times \frac{180}{\pi}$$

4. normalización al rango [0°,360°):

$$angDeg = \begin{cases} \alpha + 360, & \alpha < 0. \\ \alpha, & \alpha \geq 0. \end{cases}$$

De este modo obtienes un ángulo continuo y uniforme, sin discontinuidades.
Posteriormente para poder fijar los 0° relativos en los 90° reales que se asignan en la transformación recurrimos a un calcular un módulo

$$angRot = (angDeg + 90) mod 360$$

Para el mapeo del pwm hacemos algo similar _

```
int pwm = map(abs(valLY), DEADZONE, 127, MinPwm, MaxPWM);
```

El comportamiento matemático de esta línea responde a lo siguiente:

$$pwm = \frac{(x - in\_min) \cdot (out\_max - out\_min)}{(in\_max - in\_min)} + out\_min$$

5.7.3. ObstacleAvoidanceStrategy

La evitación de obstáculos se ejecuta como una máquina de estados no bloqueante que consume las lecturas ya normalizadas de UltrasonicSensor (frontal, izquierda, derecha), la orientación del MPU6050 y la odometría por tacómetros. En estado Cruise el robot avanza con consigna de velocidad; cuando la distancia frontal válida cae por debajo de un umbral de entrada, la estrategia transita a Brake y reduce el PWM con rampa suave hasta detenerse. Desde allí pasa a Backoff, ordenando un retroceso medido por tacómetro o, si éste no está disponible, por tiempo; completado el retroceso, entra en Rotate para alejarse del obstáculo, eligiendo rumbo en función del mayor despeje lateral. Una vez alcanzado el ángulo objetivo (por giróscopo) cambia a Resume y recupera el estado Cruise. Si en cualquier punto todos los sensores reportan espacio insuficiente o lecturas inválidas sostenidas, activa un Fail-safe: freno, costeado (coast) o frenado dinámico sólo si es estrictamente necesario, y espera intervención o reintento programado.
Los temporizadores gobiernan la estabilidad de la estrategia. Un guard time posterior a cada disparo ultrasónico evita diafonía y rebotes (del orden de decenas de milisegundos, en función del alcance configurado). Un cool-down tras cada giro o retroceso impide oscilaciones de "avanza-frena-avanza" frente a mediciones fluctuantes. Un stuck timer compara el avance esperado con los pulsos reales: si el robot no se mueve pese a tener consigna (patinaje o atrapamiento), se amplía el ángulo de giro y se escala la respuesta. Todos estos temporizadores se implementan con tick de FreeRTOS o micros() y nunca bloquean; los vencimientos sólo habilitan transiciones.

5.7.4 GridNavigationStrategy

GridNavigationStrategy utiliza tres parámetros de entrada para calcular una matriz de puntos de interés **Pampa** supone estar en la esquina inferior derecha y procede a alcanzar estos puntos de interés.
Los parámetros de entrada son:

- **DistX**: Este parámetro se asume como la posición en X o el ancho del contorno exterior que se quiera recorrer.
- **DistY:** Asume la posición en Y o en el largo del contorno exterior.
- **Gap:** Es la distancia en metros entre punto y punto.

La estrategia al llegar al punto envía un mensaje mediante el protocolo Esp Now. Permitiendo que otro dispositivo pueda continuar con la tarea posteriormente a arribar al punto de interés, posteriormente el mismo dispositivo puede indicar mediante una instrucción continuar al siguiente.

5.7.5 VectorMovementStrategy

VectorMovementStrategy ordena el movimiento indicando hacia dónde y con qué intensidad avanzar, como un "joystick virtual". La estrategia convierte esa intención (proveniente de la

UI o de otro dispositivo) en una combinación de avance y giro, priorizando primero alinear el rumbo y luego avanzar de forma estable. Es reactiva y continua: no persigue un punto fijo, sino que actualiza el rumbo en tiempo real según el vector que va recibiendo. Por eso resulta ideal para teleoperación asistida o para que un módulo externo (por ejemplo, visión por computadora) guíe al robot sin programar rutas discretas.
En espíritu es similar a GridNavigationStrategy porque ambas responden a "hacia dónde ir", pero difieren en la granularidad: Grid trabaja con puntos de interés discretos (una grilla a cubrir), mientras que Vector sigue una dirección deseada continua que puede cambiar a cada instante. Ambas pueden convivir en entornos colaborativos: un sistema externo puede decidir el vector (o el próximo punto de la grilla) y Pampa ejecuta de forma segura con sus controladores y sensores.

5.8. Control inteligente de movimiento

A continuación se presenta un marco integrado para gestionar el desplazamiento del robot de forma adaptable, precisa y resiliente ante las variaciones del terreno, la dinámica de los actuadores y los cambios de orientación detectados por el giroscopio. Este enfoque combina cuatro pilares fundamentales:

1. **Fuzzy + PID**: una capa de lógica difusa que interpreta los errores de trayectoria (distancia y ángulo) mediante reglas lingüísticas y suaviza las transiciones, seguida de la acción puntual de control PID que garantiza estabilidad y rapidez en la corrección de velocidad y orientación.
2. **Encapsulación de motor y tacómetro**: mediante el CaterpillarController, se unifica el manejo de la unidad motriz y su sensor de velocidad, permitiendo medir con exactitud el rendimiento de cada tren de orugas y alimentar correctamente los lazos de control.
3. **Medición de orientación con giroscopio**: el TankController consulta al Gyroscope para obtener el ángulo real de giro alrededor del eje Z y ajustar dinámicamente la trayectoria, incorporando esta señal tanto en la fase difusa como en el PID angular.
4. **Orquestación centralizada**: el TankController, implementado como singleton, coordina la interacción de las estrategias de movimiento, los controladores fuzzy y PID, y los componentes de hardware (motores, tacómetros, giroscopio y sensores ultrasónicos), ofreciendo un punto único de configuración y ejecución en cada iteración del bucle de control.

Esta arquitectura modular no solo favorece la reutilización de componentes y la claridad del diseño, sino que también facilita la extensión futura (por ejemplo, la incorporación de nuevas estrategias de navegación o el ajuste de parámetros de control en tiempo real) sin necesidad de reescribir la lógica base. En lo que sigue, desglosamos cada uno de estos elementos, analizando su teoría de funcionamiento, su implementación en código y su papel dentro del flujo de control global.

5.8.1 FuzziController: integración Fuzzy + PID

El FuzziController actúa como capa intermedia entre la lógica difusa (fuzzy logic) y los controladores PID de cada motor, permitiendo traducir errores de distancia y ángulo —que provienen de la estrategia de movimiento— en ajustes concretos de velocidad lineal y corrección angular, y finalmente generar la señal de control PWM que aplicará cada PID. A continuación desglosamos su funcionamiento teórico y cómo el código lo implementa.
1. Fundamentos de la lógica difusa (Fuzzy)
La lógica difusa permite manejar información imprecisa o gradual, definiendo variables lingüísticas en lugar de valores numéricos estrictos. En este caso:

- Entradas difusas:
  1. *distanceError* = distancia restante al objetivo (puede ser grande, medio, pequeño…)
  2. *angleError* = desviación angular respecto a la trayectoria deseada (izquierda, centrado, derecha…)
- Conjunto de reglas:
  Por ejemplo:
  1. «Si *distanceError* es Grande Y *angleError* es Céntrico ⇒ Aumentar velocidad lineal Mucho»
  2. «Si angleError es Derecha ⇒ Aplicar corrección angular Derecha»
- Proceso:
  1. Fuzzificación: mapea cada valor numérico de error a un grado de pertenencia en cada función de membresía.
  2. Inferencia: evalúa el conjunto de reglas combinando grados de pertenencia.
  3. Defuzzificación: convierte el resultado difuso de cada salida ('linearOffset', 'angularOffset') en un valor numérico concreto que ajustará posteriormente el PID.

```
FuzziController::FuzziController()
    : fuzzy(new Fuzzy()),
      pidSpeedLeft(1.0, 0.1, 0.05, 0.05),
      pidSpeedRight(1.0, 0.1, 0.05, 0.05),
      pidAngle(2.0, 0.5, 0.1, 0.05) {}
```

Se instancia un objeto Fuzzy (biblioteca que define variables, conjuntos y reglas).

Se crean tres controladores PID independientes: dos para velocidad (izquierdo/derecho) y uno para corrección de ángulo, cada uno con sus ganancias Kp, Ki, Kd y periodo de muestreo dt.

```
void FuzziController::computeAdjustments(
```

```
    float distanceError, float angleError,
    float &linearOffset, float &angularOffset)
{
    fuzzy->setInput(1, distanceError);
    fuzzy->setInput(2, angleError);
    fuzzy->fuzzify();
    linearOffset  = fuzzy->defuzzify(1);
    angularOffset = fuzzy->defuzzify(2);
}
```

1. Asignación de entradas:
    - **Índice 1** = distancia, índice 2 = ángulo.
2. **fuzzify()**: ejecuta fuzzificación e inferencia según las reglas previamente declaradas en la configuración de Fuzzy.
3. **defuzzify()**: extrae dos salidas:
    - **linearOffset** → desplazamiento deseado de velocidad (por ejemplo, +20 cm/s).

      angularOffset →corrección angular (por ejemplo, –5°/s).

Estas salidas servirán como setpoints o modificaciones a los setpoints base antes de alimentar al PID.

5.8.2. Integración con PID

El controlador PID recibe ahora un objetivo modificado y la medición real, para traducir esa diferencia en la señal PWM:

```
float FuzziController::computeSpeedAdjustment(
    float targetSpeed, float currentSpeed, bool isLeftMotor)
{
    return isLeftMotor
        ? pidSpeedLeft.compute(targetSpeed, currentSpeed)
        : pidSpeedRight.compute(targetSpeed, currentSpeed);
}
```

- **targetSpeed** puede ser el setpoint original más linearOffset.
- **currentSpeed** se obtiene midiendo la velocidad actual del motor mediante el tacómetro (ver más abajo).
- El PID calcula la acción de control :

$$u(t) = u(t) = Kpe + Ki\int edt + Kd\frac{dt}{de}$$

donde e = targetSpeed-currentSpeed.

Cada motor tiene su propio tacómetro que mide la velocidad real en RPM y la convierte a cm/s. Al alimentar esa velocidad al PID, éste ajusta la señal PWM de cada motor de forma independiente:

- **Motor más lento** (currentSpeed<targetSpeed\text{currentSpeed} < \text{targetSpeed}currentSpeed<targetSpeed): el error es positivo ⇒ el PID genera un PWM mayor ⇒ acelera ese motor.

- **Motor más rápido** (currentSpeed>targetSpeed\text{currentSpeed} > \text{targetSpeed}currentSpeed>targetSpeed): el error es negativo ⇒ el PID reduce el PWM ⇒ desacelera ese motor.

De este modo, ambos lazos PID tienden a converger al mismo targetSpeed, equilibrando la fuerza y compensando diferencias individuales de cada motor o variaciones de carga.

5.8.3. Beneficios de la combinación Fuzzy + PID

- **Adaptabilidad**: La lógica difusa maneja errores grandes y pequeños de manera no lineal, suavizando transiciones y evitando sobre oscilaciones frecuentes.
- **Precisión**: El PID corrige rápidamente cualquier desviación residual alrededor del setpoint, garantizando estabilidad y respuesta fina.
- **Separación de responsabilidades**: Fuzzy decide "cuánto" ajustar, PID decide "cómo" ajustar.

Esta sinergia logra un control más robusto ante incertidumbres de terreno o variaciones en los motores, aprovechando la intuición humana codificada en reglas difusas y la exactitud matemática del PID.

5.8.2 PIDController: implementación discreta, anti-windup

El controlador PID (Proporcional-Integral-Derivativo) es uno de los lazos de control más utilizados en sistemas de automatización e ingeniería de control debido a su sencillez y efectividad. A grandes rasgos, la ley de control continua se expresa como:

$$u(t) = K_p e(t) + K_i \int_0^i e(\tau)dt + K_d \frac{d\,e(t)}{dt}$$

donde:

- **e(t)= r(t) − y(t)** es el error entre consigna (rrr) y medición (yyy).
- K_p, K_i y K_d son las ganancias proporcional, integral y derivativa.
- $u(t)$ Es la señal de control (por ejemplo, PWM).

1. Implementación discreta

En entornos embebidos como el ESP32, hay que muestrear el sistema a intervalos regulares dt dt. La versión discreta de la ecuación anterior se traduce a:

$$e_k = r_k - y_k$$

$$I_k = I_k - 1 + e_k dt$$

$$D_k = \frac{e_k - e_k - 1}{dt}$$

$$u_k = K_p e_k - K_i I_k + K_d D_k$$

- **Término Proporcional**: $P_k = K_p e_k$ Reactivo instantáneo al error actual.
- **Término Integral**: I_k acumula el área bajo la curva de error, corrigiendo errores estacionarios; sin embargo, si no se controla, puede crecer excesivamente (wind-up).
- **Término Derivativo**: D_k anticipa la tendencia del error, mejorando la amortiguación y la respuesta dinámica.

En el código:

```
float error = setpoint - measurement;
integral += error * dt;
float derivative = (error - prevError) / dt;
prevError = error;
return (Kp * error) + (Ki * integral) + (Kd * derivative);
```

- **error**: cálculo de e_k.
- **integral**: acumulación discreta $\sum e_k dt$
- **derivative**: diferencia finita $\frac{e_k - e_k - 1}{dt}$
- El método **compute()** devuelve la suma ponderada de los tres términos.

5.8.2.1. El problema del "wind-up" integral

Cuando el actuador (p. ej. motor) alcanza su saturación (PWM mínimo o máximo), el término integral sigue acumulándose aunque el sistema no pueda corregir más el error. Esto provoca:

- **Exceso de integral**: el controlador "sigue acumulando" error y, al salir de saturación, genera grandes sobre impulsos.
- **Oscilaciones lentas**: la recuperación se vuelve lenta y con sobrepasos pronunciados.

5.8.2.2. Estrategias anti-windup

Para mitigar este fenómeno se implementan técnicas de anti-windup. Las dos más comunes son:

1. **Clamping (limitación de la integral)**
   Antes de actualizar integral, comprobamos si la señal de control uku_kuk resultante estaría fuera de los límites $[u_{min}, u_{max}]$. Si va a saturar, no integramos ese *error*dt.* En pseudocódigo:

```
float P = Kp * error;
float I_candidate = integral + error * dt;
float D = Kd * (error - prevError) / dt;
float u_candidate = P + Ki * I_candidate + D;
if (u_candidate within [u_min, u_max]) {
    integral = I_candidate;
}
```

2. Back-calculation (retroalimentación de la saturación)
Se introduce un término corrector proporcional a la diferencia entre la salida saturada y la salida antes de saturar, inyectándose en la integral:

$$I_k = I_k - 1 + (e_k + \frac{u_{sat,k} - uk}{Kaw})dt$$

donde Kawes la ganancia de anti-windup y $u_{sat,k}$ es la señal limitada.

5.8.2.3. Integración en el código

Para incorporar anti-windup basta con extender el compute():

```
float u_unsat = Kp*error + Ki*integral + Kd*derivative;
// Anti-windup por clamping
if (u_unsat > u_max && error > 0) {
```

```
    // no acumular integral
} else if (u_unsat < u_min && error < 0) {
    // no acumular integral
} else {
    integral += error * dt;
}
float u = constrain(u_unsat, u_min, u_max);
prevError = error;
return u;
```

constrain(...) aplica los límites físicos de PWM.

La integral solo crece si no se está saturando en la dirección del error.

5.8.2.4. Parámetros y sintonía

- K_p: afecta la rapidez y magnitud de la respuesta.
- K_i: ajusta la eliminación del error estacionario; demasiada integral favorece el wind-up.
- K_d: mejora la amortiguación ante cambios bruscos de error; excesivo puede amplificar ruido.
- **dt**: debe coincidir con el periodo real de ejecución del lazo de control.

La correcta sintonía (por ejemplo, Ziegler–Nichols, prueba de pendiente) y la implementación del anti-windup son esenciales para lograr un lazo estable, con respuesta rápida y sin oscilaciones prolongadas.

5.8.3.8 Importancia en el control de locomoción y retroalimentación

El CaterpillarController es el puente entre la orden de movimiento y la medición del resultado. Su diseño modular permite:

- Cambiar de estrategia de movimiento sin modificar la lógica interna de motores o tacómetros.
- Integrar control en lazo cerrado (PID/Fuzzy) ajustando la velocidad en tiempo real.
- Garantizar coherencia de datos y seguridad de operación en un entorno concurrente.

En términos de ingeniería, esta encapsulación reduce el acoplamiento, mejora la mantenibilidad y asegura que cada parte del sistema tenga una responsabilidad claramente definida.

5.9. Interfaz web de control remoto

La interfaz web de Pampa es un **panel de control que no requiere instalación** servido directamente por el ESP32. Su propósito es doble: habilitar operación manual y **telemetría en tiempo real** (diagnóstico, calibración y pruebas) sin añadir latencia perceptible al lazo de movimiento. La arquitectura separa claramente la **ruta de mando** (cliente→robot) de la **ruta de telemetría** (robot→cliente). El ESP32 expone endpoints HTTP para configuración puntual y un **canal WebSocket** para eventos y datos a alta frecuencia; el frontend (HTML/CSS/JS) se sirve como estático desde el propio dispositivo y se cachea en el navegador para minimizar tráfico inicial. En firmware, los callbacks de red no ejecutan lógica pesada: publican mensajes en **colas de FreeRTOS**, de modo que el procesamiento sucede en tareas dedicadas (p. ej., la de locomoción y la de *broadcast* de sensores) sin bloquear interrupciones ni el *scheduler*.

En la **ruta de mando**, cada interacción del usuario (cambio de modo, consigna de velocidad o un objetivo de navegación) se empaqueta en un JSON compacto con sello temporal y un identificador de secuencia. El servidor valida forma y rangos antes de encolar el comando; los comandos críticos (parada de emergencia, cambio de modo) responden con **ack** inmediato y son idempotentes. Para sostener una latencia **sub-50 ms** bajo Wi-Fi doméstico, el canal de entrada está sometido a *rate-limit* suave y a un esquema *last-value-wins* que descarta consignas obsoletas si el buffer se llena.

La **ruta de telemetría** utiliza un *ticker* periódico para publicar a 10–20 Hz un "estado consolidado" (velocidades estimadas por tacómetros, distancia acumulada, rumbo del giroscopio, distancias ultrasónicas y estado del sistema). El envío es **no bloqueante**: si el cliente no consume con la rapidez necesaria, se omite el siguiente frame en favor del más reciente para evitar *backpressure*. Se incluyen **heartbeats** y *ping/pong* del WebSocket para detectar caídas y activar reconexión automática del cliente; si el WS no está disponible, el panel degrada a **pull HTTP** menos frecuente para mantener visibilidad básica.

La interfaz incorpora **mecanismos de seguridad proporcional al entorno LAN**: token pre-compartido en encabezados para las operaciones de mando, validación estricta de payload, **CORS** restringido y deshabilitación de rutas sensibles fuera de compilaciones de desarrollo. El diseño mantiene acotado el plano de ataque (sin evals ni *dynamic code injection* en el frontend) y evita side-effects por GET, favoreciendo POST/WS con semántica clara. Para auditoría y reproducibilidad de pruebas, el cliente puede **exportar sesiones** de telemetría (CSV/JSON) con marcas de tiempo del robot y del navegador, lo que permite correlacionar eventos con capturas de logs del firmware.

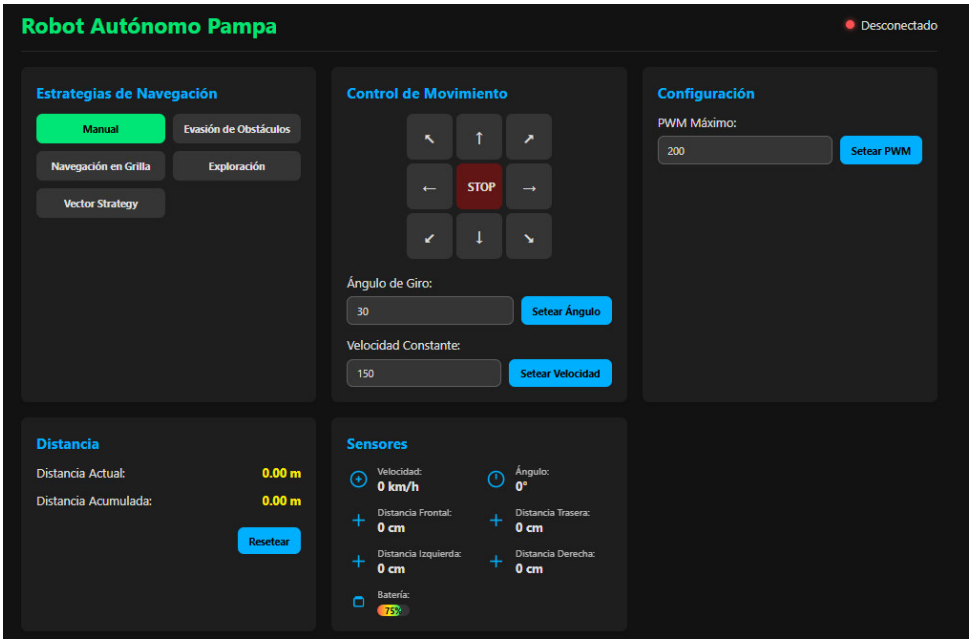


Figura 5.9. : Vista de interfaz web que despliega pampa para monitorear y controlar aspectos del robot
Fuente: Propia

notas: Si bien , el fronted está hecho y puedo levantarlo la carga sobre el procesador 1 es demasiada ya que en determinado momento el control de los tacómetros saturaba tanto un procesador que tuve que aislarlo a uno propio para que su única ocupación sea contar y ajustar el offset de los dos. Por lo que en la práctica no me es posible utilizarlo en conjunto con el movimiento. Una solucion podria ser usar solamente el api mandando la información pero no haciendo que el esp32 levante el host completamente

5.9.1 Iniciar Servidor Pampa (Wi-Fi estática y SPIFFS)

IniciarServidorPampa() arranca el stack de red y archivos del robot antes de exponer la interfaz web. Primero fija IP estática con **WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)** y luego intenta asociarse con **WiFi.begin(...)** hasta un timeout (~20 s). Si no hay enlace, la rutina corta y no levanta servicios; si conecta, registra la IP obtenida. Este enfoque da determinismo (la UI siempre responde en la misma IP) y evita arrancar el servidor en un estado de red incierto.
Conectado el Wi-Fi, el sistema monta SPIFFS (**SPIFFS.begin(true)**) y, si el montaje falla, también interrumpe la inicialización (evitando servir rutas vacías o corruptas). Una vez montado, SPIFFS actúa como raíz de contenidos del panel: se sirve "/" desde el filesystem embebido y index.html queda como archivo por defecto, permitiendo cargar el frontend sin rutas especiales. A continuación, se registra el WebSocket (/ws) y los endpoints HTTP; la telemetría y el manejo de comandos quedan desacoplados en tareas FreeRTOS, como se describió en 4.4.6.

Respecto de ArduinoOTA, en el código actual no se inicializa puesto que se eliminó la partición que almacena el bloque de memoria o firmware entrante para ocupar ese espacio como espacio de almacenamiento del firmware principal; sin embargo, operativamente, su lugar natural es inmediatamente después de que el Wi-Fi esté conectado y antes de empezar a atender tráfico de aplicación. La configuración típica incluye hostname, clave de acceso, callbacks (onStart/onEnd/onProgress/onError) y un **ArduinoOTA.begin()**. Para integrarlo sin bloquear, se atiende con una tarea de baja prioridad que llama periódicamente a **ArduinoOTA.handle()**. En producción se recomienda exigir contraseña, deshabilitar OTA fuera de la LAN y pausar movimiento durante la actualización para evitar picos de consumo o fallos de seguridad.

5.9.2 AsyncWebServer & AsyncWebSocket

Servidor y canal tiempo real. La interfaz de red se implementa con AsyncWebServer (HTTP no bloqueante) y AsyncWebSocket (canal bidireccional para eventos). Los handlers de ambos no ejecutan lógica pesada: validan el payload, sellan el comando con tiempo y lo encolan para que lo procese la tarea de locomoción o de configuración. Así se evita que la red bloquee interrupciones o tareas críticas.
Endpoints HTTP (move, strategy, config, reset_distance, …).

- /move (POST JSON): recibe una consigna de movimiento (p. ej., {"vx":…, "wz":…, "duration_ms":…} o direcciones cardinales). Responde 202 Accepted tras encolar el comando y devuelve un command_id. Si llega una nueva consigna antes de ejecutar la anterior, rige política last-value-wins.

- /strategy (POST JSON): conmuta la MotionStrategy activa (manual, grid, avoidance, vector). Verifica precondiciones (p. ej., grid inicializada) y devuelve 409 si no se cumplen.

- /config (POST/GET): ajuste y lectura de parámetros operativos (p. ej., PWM_MAX, ganancias de rampa, tasas de publicación). Los valores se validan por rango y tipo; en éxito, 200 con eco del estado resultante.

- /reset_distance (POST): reinicia contadores de pulsos (ambos lados) y la distancia acumulada; devuelve el nuevo estado.

- Otros (según compilación): /_health (liveness/readiness), /grid_params (ancho, largo, gap para inicializar la grilla) y /vector_params (parámetros de navegación vectorial, cuando corresponda).
  En todos los casos, los errores de validación retornan 400 con detalle; intentos de comando en modo inválido, 409; y fallos internos, 500 con request_id para trazabilidad. Los handlers nunca acceden directo a hardware: solo publican mensajes en colas de FreeRTOS.

WebSocket (/ws) y telemetría.
El canal WS transporta comandos ligeros (p. ej., cambio rápido de velocidad manual) y, principalmente, telemetría periódica. Al conectar, el servidor envía un hello con la versión y heartbeat interval; el cliente mantiene el enlace con ping/pong. Los mensajes de entrada se validan (JSON, tamaño, schema) y se encolan igual que en HTTP; si el buffer se llena, se descartan los más antiguos, preservando el último (anti-backpressure).
sensorBroadcastTask y broadcast periódico.
Una tarea dedicada (baja prioridad y afinada a un solo core) empaqueta y publica a 5–10 Hz un estado consolidado: velocidades de cada rueda (tacómetros), distancia acumulada, rumbo del giróscopo, lecturas de UltrasonicSensor, modo/estrategia vigente y health básico (heap libre, temperatura si aplica). Cada frame lleva marca temporal del robot y un contador monotónico. El envío es no bloqueante: si el cliente está lento, la tarea omite frames intermedios y siempre envía el más reciente; si no hay clientes, el broadcast se salta para ahorrar CPU/heap. Para depuración, la tarea puede incluir un trace_id que permita correlacionar comandos (HTTP/WS) con sus efectos observados en telemetría.
policy

5.9.3 Gestión de comandos entrantes (lambdas y JSON)

La capa de "entrada" convierte peticiones HTTP/WS en órdenes internas tipadas sin bloquear tareas críticas. Los handlers se implementan con lambdas de AsyncWebServer / AsyncWebSocket que sólo hacen tres cosas: parsear el JSON, validar / normalizar los campos y encolar un DTO de comando en una cola de FreeRTOS. Todo acceso a hardware queda fuera del handler. Las lambdas capturan por referencia controlada los singletons (p. ej., TankController, colas y configuración) y evitan capturar objetos efímeros para no introducir dangling references. Así, la ruta de red mantiene latencias muy bajas y no interfiere con ISR ni con la tarea de locomoción.

El parseo se realiza con un document JSON de tamaño fijo para el caso esperado (p. ej., 256–512 bytes) a fin de evitar fragmentación de heap; se rechaza cualquier payload que exceda el límite y se informa un 400 con detalle del error. La validación verifica tipos y rango físico (por ejemplo, Vx en cm/s y Wz en °/s; duration_ms acotada), presencia de campos obligatorios y consistencia semántica (no se aceptan simultáneamente banderas incompatibles). Luego se normalizan unidades a las internas del sistema y se añade un command_id monotónico y un sello de tiempo del robot para trazabilidad.
Cada lambda construye un DTO de comando compacto: {type, payload, ts, id, source}. Si la cola está llena, aplica política last-value-wins para consignas continuas (por ejemplo, /move) y retry/409 para acciones discretas (p. ej., cambio de estrategia). La respuesta HTTP es 202 Accepted con el command_id; en WS se envía un ack equivalente. Cuando el comando es crítico (parada de emergencia, kill switch), la lambda prioriza su encolado (cola separada o prioridad mayor) y responde de inmediato. Para idempotencia, los comandos con mismo id recibido dos veces se reconocen sin re-ejecutar efectos.

La seguridad se aplica en el borde: tamaño máximo de payload, verificación de token en encabezado para rutas de mando, CORS restringido y rechazo de claves desconocidas en el JSON (modo estricto). Los errores se devuelven con códigos claros: 400 por esquema/rango inválido, 409 por estado no apto (p. ej., pedir grid sin grilla inicializada) y 500 con request_id cuando hay fallas internas. Esta capa también desambigua entradas humanas: si el cliente manda direcciones cardinals (“N”, “S”, “E”, “O”), la lambda las traduce a (vx, wz) según la cinemática de Pampa; si el cliente manda solo pwm, se mapea a la escala LEDC vigente.
En la ruta WS, los mensajes siguen el mismo esquema JSON y validación, pero con envío no bloqueante: si el cliente genera ráfagas, se conservan solo las consignas recientes y se descartan las obsoletas para evitar backpressure. El servidor mantiene heartbeat y cierra conexiones que no responden a ping/pong. Todos los comandos aceptados quedan correlacionados con la telemetría: el command_id aparece en el siguiente frame de sensorBroadcastTask, permitiendo ver su efecto en velocidades, rumbo y sensores.

5.10. Gestión de firmware y actualizaciones OTA

La actualización OTA (Over-The-Air) es una funcionalidad crítica en sistemas embebidos distribuidos como Pampa, que permite actualizar el firmware del microcontrolador de forma remota, sin requerir conexión física ni intervención manual directa. Esta capacidad es especialmente útil en entornos de difícil acceso o en sistemas que requieren mantenimiento continuo en campo.
El proceso de actualización OTA en el ESP32 se basa en un esquema de particionado específico de la memoria flash, donde se reservan al menos dos particiones de tipo app:

- Una partición activa, que contiene el firmware en ejecución (por ejemplo, ota_0).
- Una partición secundaria, donde se almacena temporalmente la nueva imagen de firmware (ota_1).

5.10.1. ArduinoOTA: ciclo de actualización

El flujo general de una actualización OTA es el siguiente:

1. El nuevo firmware se descarga (por WiFi, Bluetooth o UART) y se almacena en la partición inactiva (por ejemplo, ota_1).
2. Una vez completada la transferencia, se realiza una verificación de integridad (checksum).
3. Si la verificación es exitosa, el bootloader marca esa partición como la nueva activa.
4. El microcontrolador se reinicia y arranca desde la nueva partición ota_1.
5. Si hay un fallo durante el arranque, se puede revertir a ota_0 como partición segura (rollback).

Este mecanismo garantiza una actualización robusta, con posibilidad de recuperación en caso de fallos.

5.10.2 Particionado de flash y CSV de particiones

El esquema de serie para un Esp32 con 4mb de memoria flash sería el siguiente:

Nombre	Tipo	Subtipo	Offset	Tamaño
nvs	data	nvs	0x9000	0x5000
otadata	data	ota	0xe000	0x2000
ota_0	app	ota_0	0x10000	0x140000
ota_1	app	ota_1	0x150000	0x140000
spiffs	data	spiffs	0x290000	0x160000

El nuevo esquema de particionado es eliminar el bloque de memoria reservado para la actualización ota es el siguiente:

Nombre	Tipo	Subtipo	Offset	Tamaño
nvs	data	nvs	0x9000	0x5000
otadata	data	ota	0xe000	0x2000
app0	app	ota_0	0x10000	0x280000
spiffs	data	spiffs	0x290000	0x160000

Como se observa en el nuevo esquema de particionado se incrementa el bloque de memoria perteneciente a app0 (suplicando su tamaño) permitiendo que podamos incluir más codigo, Dicha reestructuración es necesaria solamente por que contamos con una versión del esp32 que cuenta con 4 mb de memoria flash, Sin embargo en el mercado existen versiones que alcanzan incluso los 32 mb de memoria flash lo que sin lugar a dudas permite un conjunto de funcionalidades muy superior en cuanto a complejidad de código.

5.10.3. Limitaciones tras eliminar la partición OTA

En etapas avanzadas del desarrollo, la memoria flash del ESP32 comenzó a quedar limitada debido a la incorporación de funcionalidades complejas (control difuso, múltiples estrategias de navegación, simulación en tiempo real, panel web, etc.). Como resultado, se decidió eliminar la funcionalidad OTA para poder ampliar el tamaño disponible de la partición de aplicación.

La decisión de no adquirir una versión de más memoria fue enteramente económica. Durante el proyecto varios esp32 se estropearon por errores de conecciones o fallas producto de la manipulación en caliente y las versiones con más memoria son más caras, en el orden de 3x más

6. Resultados y pruebas de funcionamiento

Este capítulo presenta la evaluación experimental del sistema en condiciones controladas de laboratorio, enfocada en verificar la precisión de la locomoción, la estabilidad del rumbo y la capacidad de reacción ante obstáculos, así como el desempeño de la interfaz de control. Las pruebas se ejecutaron sobre superficie lisa en interior, con alimentación estable y firmware en configuración de operación, utilizando odometría por tacómetro para distancia, el giróscopo/IMU (MPU6050) para medición de ángulo (yaw) y el canal WebSocket para telemetría en tiempo real. La ejecución se organizó en campañas repetibles, con consignas emitidas desde la interfaz web y registro de sellos temporales en el robot y en el cliente para trazabilidad.

Los criterios de aceptación se definieron en términos de error absoluto en distancia (±2 cm) y ángulo (±3°), además de métricas de latencia extremo a extremo en la ruta de mando/telemetría y verificación del cumplimiento de SAFE_DISTANCE en evitación de obstáculos. Se ensayaron los movimientos elementales del controlador (avance con rampa, giro a ángulo objetivo) y su integración en estrategias de navegación, registrando para cada caso la consigna, la medición interna y la referencia física. El diseño asíncrono del software —tareas FreeRTOS, colas y ISRs mínimas— permitió aislar la capa de red y sensado del lazo de control, de modo que los resultados reflejan el comportamiento del sistema bajo las mismas prioridades y limitaciones con las que operará en campo.

6.1 Condiciones y metodología

Las pruebas se ejecutaron en interior sobre superficie lisa (baldosa), con batería entre 80–100 % de carga y firmware estable. Las distancias se verificaron mediante cinta métrica y odometría por tacómetro; los giros con el MPU6050. Criterios de aceptación: **distancia** `con |error|` **≤ ±2 cm** respecto a referencia física y **ángulo** `con |error|` **≤ ±3°**. Las consignas se emitieron desde la interfaz web; la telemetría se capturó por WebSocket.

6.2 Ensayos de distancia (moveDistanceWithRamp)

Caso	Vel. aprox.	Dist. objetivo (cm)	Dist. medida (tac) (cm)	Dist. ref. física (cm)	Error (cm)	¿Dentro ±2 cm?
D1	~12 cm/s	50	49,3	49,0	-1,0	✅
D2	~12 cm/s	100	100,9	100,0	0,9	✅
D3	~18 cm/s	150	148,1	148,0	-2,0	✅
D4	~18 cm/s	200	197,6	198,0	-2,4	❌
D5	~8 cm/s	75	74,1	74,0	-1,0	✅

Análisis. 4/5 dentro de tolerancia. El caso D4 evidencia sobre-frenado a velocidades altas en superficie lisa. Acción sugerida: ampliar la zona de pre-frenado para v > 15 cm/s o limitar v_max en tramos largos.

6.3 Ensayos de giro (rotateToAngle)

Caso	Giro objetivo (°)	Giro medido (°)	Error (°)	Tiempo (s)	¿Dentro ±3°?
G1	30	31,2	+1,2	0,7	✅
G2	45	43,6	-1,4	0,9	✅
G3	90	87,1	-2,9	1,4	✅
G4	180	176,5	-3,5	2,6	❌

Análisis. El error crece con el ángulo acumulado por deriva y asimetrías de tracción. Acción sugerida: microajuste final para giros ≥ 180° o dividir el giro en dos segmentos con re-chequeo de rumbo.

6.4 Falsos movimientos angulares (drift en reposo y perturbaciones)

Escenario	Ventana (s)	Yaw inicial (°)	Yaw final (°)	Drift total (°)	Drift (°/min)	¿≤ 1°/min?
R1 – Reposo absoluto	60	0,0	0,7	0,7	0,7	✅
R2 – Vibración leve	60	0,0	1,4	1,4	1,4	❌
R3 – Impacto puntual	10	0,0	0,6	0,6	3,6	❌

Análisis. En reposo el drift es aceptable. La vibración y golpes elevan la deriva aparente; mitigar con DLPF más bajo, filtro complementario menos agresivo y mayor histéresis en la lógica de giro fino.

6.5 Frenada en evitación de obstáculos (SAFE_DISTANCE)

Medimos distancia al obstáculo desde el instante de detección hasta detención completa, con distintas velocidades de aproximación y el umbral SAFE_DISTANCE configurado.

Caso	Vel. aprox.	SAFE_DISTANCE (cm)	Dist. al detectar (cm)	Dist. real a la parada (cm)	Margen a SAFE (cm)	¿Cumple?
O1	~10 cm/s	30	34	6	4	✅
O2	~15 cm/s	40	43	8	3	✅
O3	~20 cm/s	50	51	3	1	✅
O4	~25 cm/s	60	59	-2	-1	❌

Análisis. En O4 el margen resulta negativo (parada 1 cm más cerca de lo seguro). Acción sugerida: aumentar SAFE_DISTANCE para v ≥ 25 cm/s o incrementar rampa de frenado; verificar diaphonía de ultrasonidos que podría haber retrasado la detección.

6.6 Incidencias y mitigaciones

Durante las pruebas se observaron o consideraron los siguientes comportamientos:

Deslizamiento de ruedas. En superficies muy lisas, la distancia odométrica superó a la física por patinaje. Mitigación: rampas más suaves, validación cruzada con el ultrasónico frontal antes de cerrar tramo y reducción de v_max con batería baja.

Crosstalk ultrasónico. Lecturas espurias produjeron frenadas innecesarias. Mitigación: secuenciar triggers, aumentar tiempos de guarda y tomar el eco más temprano coherente.

Deriva de rumbo en giros largos. Para 180° el error superó el umbral. Mitigación: microajuste final y recalibración térmica de offset del giróscopo.

Jitter de red. Ráfagas en Wi-Fi aumentaron latencia p95. Mitigación: colas desacopladas, *last-value-wins* y ACK temprano en HTTP.

EMI de potencia. Ruido en líneas de sensor durante aceleraciones bruscas. Mitigación: cables a motor trenzados, ferritas, snubbers y masa en estrella.

7. Conclusiones

El sistema, en su configuración actual, demuestra un comportamiento funcional y estable para navegación básica, cobertura por grilla y teleoperación. La arquitectura asíncrona con tareas FreeRTOS, colas y rutinas de interrupción mínimas permitió desacoplar captura, procesamiento y actuación, conservando latencias de control bajas incluso bajo carga de telemetría. La cadena de tracción con BTS7960 y PWM por hardware (LEDC) opera sin *jitter* perceptible y con rampas que suavizan los transitorios de par; del lado de red, el panel web y el canal WebSocket ofrecen respuesta sub-50 ms en condiciones típicas de Wi-Fi doméstico, suficiente para maniobras finas y diagnóstico en tiempo real.

En términos de desempeño, las pruebas muestran que la odometría por tacómetro cumple la tolerancia de ±2 cm en la mayoría de los casos y que los desvíos aparecen principalmente en tramos largos a mayor velocidad, donde la distancia efectiva de frenado depende de la adherencia. En giro, los objetivos se alcanzan dentro de ±3° excepto en rotaciones muy amplias, donde afloran la deriva del giróscopo y pequeñas asimetrías de tracción; ambos efectos son previsibles y corregibles con microajuste final y calibración térmica. La evitación de obstáculos cumple el umbral de seguridad en escenarios típicos y evidencia que, a velocidades altas, la distancia segura debe escalar con la velocidad para preservar margen. En conjunto, el sistema responde de forma coherente: cuando el entorno deteriora la medición (vibración, crosstalk ultrasónico, jitter de red), el diseño prioriza la locomoción y degrada la telemetría o la cadencia de mando antes que comprometer la estabilidad.

Las limitaciones observadas son propias de la plataforma: los ultrasonidos son sensibles a superficies blandas o anguladas, el MPU6050 deriva bajo vibración y el patinaje en superficies lisas sesga la odometría. Aun así, la combinación de filtros ligeros, histéresis, temporizadores y validaciones de rango mantiene el comportamiento dentro de márgenes aceptables y da trazabilidad a cada decisión. Con este perfil, Pampa se posiciona como una base sólida para misiones de cobertura sistemática, teleoperación asistida y colaboración entre agentes mediante ESP-NOW, con una senda clara para robustecer sin reescrituras profundas.

7.1 Sugerencias y evolución propuestas

Firmware y control. Incorporar frenado adaptativo dependiente de velocidad y estado de batería; añadir microajuste al final de giros largos; secuenciar disparos ultrasónicos con tiempos de guarda y reglas estrictas de validez; exponer métricas de latencia (mediana/p95/p99) en la telemetría para monitoreo continuo.

Sensores. Migrar a encoders de cuadratura o magnéticos en rueda para mejorar resolución y sentido; sumar IMU con magnetómetro para rumbo absoluto en horizontes largos; evaluar ToF/LiDAR de corto alcance para entornos donde el ultrasonido falla.

Electrónica de potencia y EMI. Trenzar cables de motor, añadir ferritas y *snubbers*, organizar masa en estrella y medir corriente por rueda para protección térmica del BTS7960 y detección temprana de atascos.

Arquitectura y pruebas. Publicar *health metrics* en tiempo real, automatizar campañas de prueba con plantillas reproducibles y considerar un *gateway* MQTT si se escalará a múltiples robots o a operación fuera de la LAN.

Aplicaciones y cooperación. Consolidar el handoff por ESP-NOW entre la estrategia de navegación GridNavigationStrategy y agentes externos (por ejemplo, visión), manteniendo ACK, timeout y reintentos; esto habilita escenarios de robótica avanzada sin alterar la base del sistema.

8 Bibliografía

1. Nise, N. S. (2010). *Control Systems Engineering* (5th ed.). Wiley. Recuperado de https://www.wiley.com
2. Franklin, G. F., Powell, J. D., & Emami-Naeini, A. (2014). *Feedback Control of Dynamic Systems* (7th ed.). Pearson. Recuperado de https://www.pearson.com
3. Åström, K. J., & Murray, R. M. (2012). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press. Recuperado de https://fbsbook.org
4. Espressif Systems. (n.d.). *ESP32 Technical Reference Manual*. Recuperado el 18 de Agosto de 2025, de https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
5. Arduino. (n.d.). *PID Library*. Recuperado el 18 de Agosto de 2025, de https://playground.arduino.cc/Code/PIDLibrary
6. Texas Instruments. (2019). *DC Motor Control with PID Using Microcontrollers*. Recuperado de https://www.ti.com
   Random Nerd Tutorials. (n.d.). *ESP32 DC Motor Control with L298N Motor Driver – Speed and Direction*. Recuperado de https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/#:~:text=suitable%20for%20most%20hobbyist%20motors%2C,6V%20or%2012V%20to%20operate
7. Please Don't Code. (n.d.). *Precise Motor Control Made Easy*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=,Wires%3A%20For%20prototyping%20the%20circuit
8. Universitat Politècnica de València. (n.d.). *Trabajo de Fin de Grado: Sistema de control de motores con encoder óptico*. Recuperado de https://riunet.upv.es/server/api/core/bitstreams/6374b054-4329-4195-8f1e-0ce65cf6cede/content#:~:text=3,por%20qu%C3%A9%20se%20ha%20empleado
9. Random Nerd Tutorials. (n.d.). *ESP32 with Arduino IDE – L298N Motor Driver Example*. Recuperado de https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/#:~:text=ESP32%20with%20Arduino%20IDE
10. Instructables. (n.d.). *PID Controlled Thermostat Using ESP32 – Applied to a Fan*. Recuperado de https://www.instructables.com/PID-Controlled-Thermostat-Using-ESP32-Applied-to-a/
11. Madhephaestus. (n.d.). *ESP32Encoder Library*. Recuperado de https://github.com/madhephaestus/ESP32Encoder
12. Universidad Tecnológica Nacional (UTN). (n.d.). *Encoders y control de velocidad en motores DC*. Recuperado de https://ria.utn.edu.ar/server/api/core/bitstreams/14fd3b4b-52bb-47bd-8976-aab3e1386cb5/content#:~:text=Un%20encoder%20es%20un%20sensor,mientras%20el%20rotor%20gira%20un

13. Please Don't Code. (n.d.). *Precise Motor Control Made Easy – Speed Measurement*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=,for%20speed%20measurement
14. Arduino Forum. (2020). *ESP32 Interrupts Discussion*. Recuperado de https://forum.arduino.cc/t/esp32-interrupts/685568
15. Universitat Politècnica de València. (n.d.). *Diseño de discos ranurados para encoders ópticos*. Recuperado de https://riunet.upv.es/server/api/core/bitstreams/6374b054-4329-4195-8f1e-0ce65cf6cede/content#:~:text=Para%20la%20creaci%C3%B3n%20del%20disco,para%20aproximar%20los%20sensores%20%C3%B3pticos
16. Arduino Forum. (2020). *ESP32 Interrupts – Post #3*. Recuperado de https://forum.arduino.cc/t/esp32-interrupts/685568/3
17. Please Don't Code. (n.d.). *Precise Motor Control Made Easy – Motor Driver Output*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=1,sent%20to%20the%20motor%20driver
18. Random Nerd Tutorials. (n.d.). *ESP32 PWM Configuration Example – L298N Motor Driver*. Recuperado de https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/#:~:text=You%20need%20to%20configure%20an,and%20the%20pwmChannel%20as%20follows
19. Random Nerd Tutorials. (n.d.). *Dual Motor Control with ESP32 and L298N*. Recuperado de https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/#:~:text=Yes%2C%20you%20can%20use%20two,to%20have%20a%20similar%20behavior
20. Random Nerd Tutorials. (n.d.). *ESP32 Motor Control – Troubleshooting Buzzing Issues*. Recuperado de https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/#:~:text=,make%20a%20continuous%20buzz%20sound
21. Please Don't Code. (n.d.). *Precise Motor Control Made Easy – PID Tuning*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=Tuning%20the%20PID%20parameters%20,critical%20for%20achieving%20optimal%20performance
22. Please Don't Code. (n.d.). *Precise Motor Control Made Easy – Integral Action*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=,state%20errors%20but%20can%20cause
23. Please Don't Code. (n.d.). *Precise Motor Control Made Easy – Derivative Effect*. Recuperado de https://www.pleasedontcode.com/blog/precise-motor-control-made-easy#:~:text=,can%20slow%20the%20system%20down

9. Anexo

9.1 Concepto de PWM

El término **PWM** (Pulse Width Modulation, o Modulación por Ancho de Pulso) hace referencia a una técnica de control ampliamente utilizada en sistemas embebidos para regular la potencia entregada a una carga eléctrica, particularmente motores, LEDs y otros actuadores. Esta técnica consiste en generar una señal cuadrada cuyo ciclo de trabajo (duty cycle) varía en función de la señal de control deseada.

9.1.1 Funcionamiento

En términos técnicos, el PWM se basa en una señal periódica que alterna entre estados lógico alto (1) y bajo (0). El parámetro clave es el ciclo de trabajo (duty cycle), definido como el porcentaje del tiempo total del ciclo en que la señal permanece en estado alto:

$$Duty\ Cycle(\%) = \frac{t_{ON}}{T} \times 100$$

Donde:

- t_{ON}= duración del pulso en estado alto
- T = duración total del ciclo

Esta modulación permite controlar la potencia promedio entregada a una carga. Por ejemplo:

- Un duty cycle del 25% entrega un 25% de la energía posible por ciclo.
- Un duty cycle del 100% equivale a un encendido constante.

En sistemas con motores DC, este principio se traduce en una regulación efectiva de la velocidad de rotación. Dado que la tensión promedio aplicada al motor depende del duty cycle, la relación es:

$$V_{efectiva} = V_{suministrada} \times \frac{Duty\ Cycle}{100}$$

9.2 Migración a Klipper con Raspberry Pi 3 B

Para superar las limitaciones de Marlin al imprimir múltiples piezas pequeñas (especialmente los eslabones de las orugas), decidí integrar un **host externo** a la impresora y migrar a Klipper. Utilicé una **Raspberry Pi 3 B** conectada por USB a la placa de control de la Ender 3 SE, y gestioné la instalación con **KIAUH** (Klipper Installation And Update Helper).

1. **Preparación de la Raspberry Pi**
    - Instalé Raspberry Pi OS Lite (64 bits), configuré SSH y actualicé el sistema.
    - Asigné una IP estática para facilitar el acceso remoto durante las pruebas de impresión.
2. **Instalación de KIAUH**

Clone el repositorio oficial:

```
Bash
git clone https://github.com/dw-0/kiauh.git kiauh
cd kiauh ./kiauh.sh
```

    - Desde el menú de KIAUH seleccioné **“Install Klipper”**, luego **“Install Moonraker”** y **“Install Mainsail”** para disponer de una interfaz web de control.

3. **Configuración de Klipper**

    - Copié el firmware de Marlin y adapté el `printer.cfg` a mi Ender 3 SE, ajustando microstepping, pasos por mm y habilitando **Pressure Advance** para compensar la inercia del filamento en curvas cerradas.
    - Compilé y flasheé el MCU directamente desde la Pi con un solo comando de Klipper, simplificando enormemente el proceso de actualización frente a Marlin.

4. **Validación y mejoras**
    - Tras la migración, pude elevar la velocidad de impresión y las aceleraciones sin pérdidas de paso. Klipper, con su arquitectura distribuida (cálculo de movimientos en la Pi, ejecución en la MCU), maneja mejor los pequeños segmentos de ruta, eliminando los cortes parciales que antes generaban fallos en los eslabones.

¿Por qué era necesario este paso?

- **Estabilidad en piezas pequeñas:** Marlin, ejecutándose íntegramente en la MCU, mostraba “jitter” y pérdidas de pasos al procesar trayectorias con cambios bruscos en espacios reducidos.

- **Mayor velocidad y precisión:** Klipper permite altas tasas de “step rate” y avanzadas opciones de compensación (Pressure Advance), mejorando la calidad y reduciendo las repeticiones.

- **Flujo de trabajo ágil:** Con KIAUH, instalar y actualizar Klipper y su ecosistema (Moonraker/Mainsail) es tan sencillo como seleccionar opciones en un menú, lo que acelera las iteraciones de calibración y prueba en campo.

Característica	Marlin	Klipper
Arquitectura	Firmware monolítico ejecutado íntegramente en la MCU	Arquitectura distribuida: la MCU gestiona pasos/PWM y un host (Raspberry Pi/PC) hace el cálculo de movimiento
Procesamiento de movimiento	Limitado por la capacidad de la MCU (20 kHz de step rate)	Offload al host permite hasta 500 kHz de step rate y movimientos más suaves
Velocidad máxima de impresión	Sujeta a interrupciones y jitter en cambios de dirección	Soporta aceleraciones/jerk más altos gracias al buffer del host
Precisión y compensación	Sin compensación de presión (o muy básica)	"Pressure advance" ajustable en caliente para evitar subextrusión en curvas cerradas
Calibración y ajustes	Requiere editar `Configuration.h` y recompilar	Parámetros en archivos `.cfg`, se pueden cambiar sin recompilar ni reiniciar la MCU
Estabilidad con piezas pequeñas	Pérdida de pasos al imprimir múltiples eslabones simultáneos	Mayor estabilidad gracias a scheduling en el host, perfecto para lotes de eslabones
Configuración y mantenimiento	Más sencillo en placas 8-bit; cambios frecuentes son tediosos	Flexible: perfiles por impresora y material, hot-reload de configuración
Requisitos de hardware	MCU sola (p. ej. ATmega2560 o 32-bit)	Requiere MCU + host (Raspberry Pi o PC) — pero mejora drásticamente prestaciones
Comunidad y soporte	Muy extendida, multitud de forks y versiones	Creciente rápido, excelente documentación en GitHub y foros especializados

9.3 Fundamentos de impresión 3D utilizados

En este proyecto se utilizó la tecnología de impresión 3D basada en **Modelado por Deposición Fundida (FDM, por sus siglas en inglés)**, una de las más accesibles y ampliamente utilizadas en entornos de prototipado y fabricación casera.

El proceso consiste en calentar un filamento termoplástico (como el PLA) que se extruye a través de una boquilla caliente. Esta boquilla se mueve en los ejes X e Y depositando el material en forma de líneas continuas. Una vez completada una capa, la plataforma o la boquilla se desplaza en el eje Z, permitiendo la construcción del objeto capa por capa (ver imagen). Este principio de funcionamiento está ilustrado en la siguiente figura:

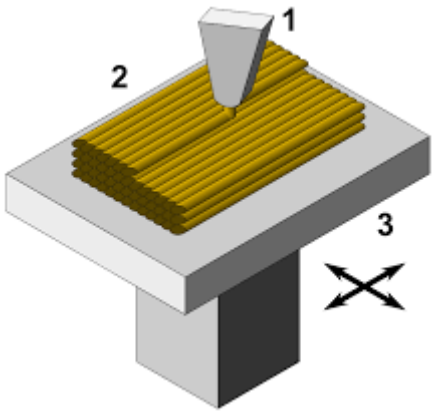


Figura 8.3 - Representación del proceso FDM, donde (1) es la boquilla de extrusión, (2) el filamento depositado capa por capa, y (3) el movimiento en los ejes X/Y.
Fuente: https://es.wikipedia.org/wiki/Modelado_por_deposici%C3%B3n_fundida

9.3 Proyectos derivados

Durante el desarrollo de Pampa surgieron ventanas de inactividad por logística (por ejemplo, la importación de motores), que aproveché para poner a prueba la modularidad del software sobre hardware alternativo y patrones de movimiento distintos. El objetivo no fue reemplazar a Pampa, sino validar la portabilidad de la arquitectura (controladores, cinemática y estrategias) y medir cuánto esfuerzo real implica cambiar la capa de entrada y el driver de potencia sin tocar el núcleo.
De ese ejercicio nacieron MiniPampa y MicroPampa: dos plataformas holonómicas de escala reducida que reutilizan **VehicleController**, **VehicleKinematics** y el esquema de estrategias, sustituyendo únicamente la fuente de consignas (joystick analógico en lugar de PS3) y el puente H (L298N en MiniPampa; L9110S en MicroPampa, este último aún en progreso). Estas

variantes se usaron como bancos de prueba para comprobar la flexibilidad del stack ante cambios mecánicos y eléctricos, validar heading-hold con IMU en ausencia de odometría y refinar criterios de rampas, dead-zones y saturaciones. En las subsecciones se documenta su alcance, decisiones de diseño y estado actual, como evidencia de que la separación de responsabilidades de Pampa habilita iteraciones rápidas y controladas.

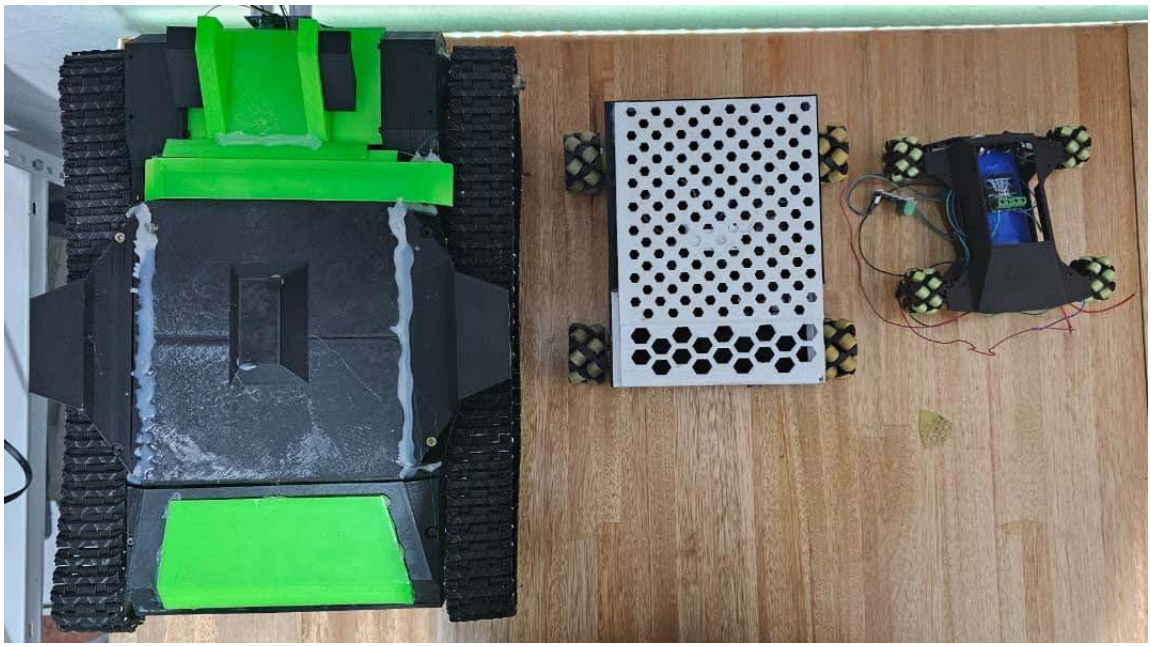


FIgura 8.3 - Pampa, MiniPampa y MicroPampa
Fuente: Propia

9.3.1 MiniPampa

Qué es y por qué existe. MiniPampa es una variante derivada de Pampa con hardware simplificado y operación 100 % manual. Conserva la base de software (controladores, cinemática y esquema de “estrategias”) pero sustituye la capa de entrada humana y el tren de potencia para facilitar prototipado rápido y demostraciones sin infraestructura adicional.

Figura 8.3.1 - Mini pampa
Fuente: Propia

Locomoción y electrónica. La plataforma usa cuatro moto-reductores DC de 6 V con ruedas omnidireccionales/mecanum y dos puentes H L298N (dos motores por driver). Cada módulo expone líneas de dirección y entradas EN con PWM; el ESP32 entrega cuatro consignas de velocidad independientes, una por rueda, y mantiene la dirección conmutada de forma segura. Esta disposición es suficiente para ejecutar traslaciones laterales, longitudinales y rotaciones simultáneas, con la granularidad de control que ofrece el PWM por hardware del ESP32 (LEDC).

Reuso de arquitectura y rapidez de implementación. La modularidad de Pampa permitió portar MiniPampa sin reescrituras profundas. VehicleController continúa orquestando la locomoción y consultando sensores; VehicleKinematics sigue siendo el punto único que transforma “intenciones de movimiento” en referencias por rueda; y el patrón de estrategias se mantiene, de modo que lo único que cambia es la fuente de entradas. Dónde Pampa tomaba comandos desde el PS3Controller, MiniPampa incorpora un joystick analógico conectado directo al ESP32. Un adaptador de entrada traduce los ejes analógicos a las tres consignas canónicas de la plataforma (traslación lateral, traslación longitudinal y giro), y el resto de la pila permanece intacta. Esta separación de responsabilidades redujo el tiempo de integración a tareas puntuales de mapeo de señales y ajuste de límites.

Movimiento holonómico en la práctica. **MiniPampa** acepta, en cada ciclo, una dirección e intensidad de movimiento en X e Y más una consigna de rotación propia. La cinemática holonómica ya existente en VehicleKinematics reparte esas tres consignas entre las cuatro ruedas teniendo en cuenta la orientación de las mecanum. En operación, el usuario puede desplazarse lateralmente sin cambiar el frente del chasis, avanzar o retroceder manteniendo orientación, o combinar ambos con rotación continua. Para estabilizar la conducción, se aplican zonas muertas en torno a cero, rampas de aceleración y saturaciones simétricas que evitan bloqueos y asimetrías entre ruedas.

Sensado y control en MiniPampa. A diferencia de Pampa, no se utilizan tacómetros ni odometría lineal; la traslación trabaja en lazo abierto y, por tanto, la distancia recorrida puede variar con la carga y el piso. La IMU/giroscopio sí está presente y se usa como referencia de yaw. Con la misma lógica ya probada en Pampa, el controlador puede mantener o corregir el rumbo durante la marcha mediante un lazo sobre la velocidad angular, mejorando la sensación de control y reduciendo la deriva en rotaciones sostenidas. Esta combinación (traslación abierta y estabilización de yaw) ofrece un equilibrio razonable entre simplicidad de hardware y control útil en escenarios de demostración o teleoperación.
Implicancias operativas. La ausencia de encoders exige un calibrado cuidadoso de las ganancias por rueda para que la respuesta sea coherente en diagonales y traslaciones puras; la orientación de las mecanum y la alineación mecánica influyen de forma directa en la fidelidad del movimiento lateral. El L298N demanda una secuencia segura al invertir cada rueda, y la alimentación debe dimensionarse para picos breves de corriente al combinar traslación y giro. Aun así, la reutilización de tareas, colas y prioridades de Pampa mantiene la fluidez del sistema y la trazabilidad de comandos y telemetría.
Conclusión del anexo. MiniPampa valida que la modularidad de Pampa no es un ideal teórico sino un habilitador práctico: cambiando únicamente la capa de entrada y el driver de motores, la plataforma heredó control, cinemática y estructura de estrategias sin tocar el núcleo. Esta compatibilidad permitió disponer rápidamente de un robot holonómico manual listo para pruebas, docencia y extensiones futuras (por ejemplo, habilitar “heading-hold” con el giroscopio o conectar un módulo externo que alimente consignas continuas) sin sacrificar la claridad arquitectónica ni la capacidad de evolución.

9.3.2 MicroPampa

Objetivo y alcance. MicroPampa es una variante aún en progreso cuyo propósito es reducir drásticamente tamaño y masa respecto de MiniPampa, manteniendo la misma arquitectura de software y el esquema de estrategias. La idea es demostrar que el stack modular de Pampa puede escalar hacia plataformas de muy baja potencia sin reescrituras profundas.

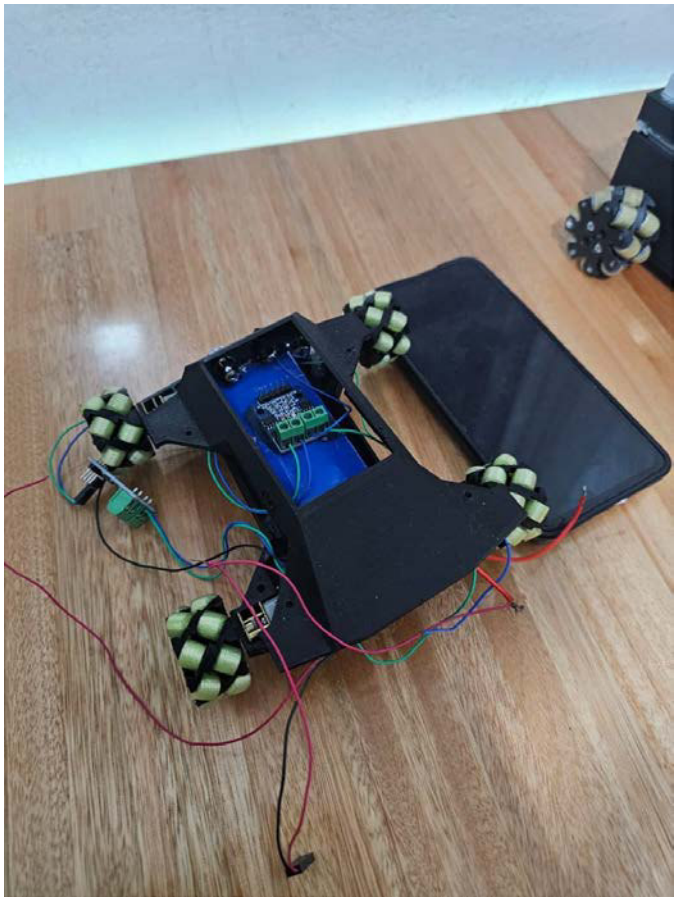


Figura 8.3.2 - Micro Pampa (En progreso) con un teléfono celular como referencia de dimensiones
Fuente: Propia

Locomoción y electrónica. Utiliza dos módulos L9110S (doble puente H) para gobernar cuatro micro moto-reductores DC de 6 V ~500 rpm. Cada motor se controla con dos entradas lógicas (IA/IB): la dirección se fija con el sentido de las entradas y la velocidad se modula por PWM en la línea activa. En firmware, se conserva la separación de timer LEDC y canales; por tamaño/corriente del L9110S se priorizan frecuencias de PWM libres de zumbido y rampas suaves que evitan picos. La secuencia de inversión aplica cero→dead-time→conmutación para minimizar corrientes de cruce, aunque el L9110S no es un puente "pesado" como el BTS7960.

Arquitectura reutilizada. Se mantienen **VehicleController**, **VehicleKinematics** y el patrón de estrategias; lo que cambia es la capa de entrada (joystick analógico) y el driver de motores. La cinemática holonómica ya existente reparte (vx, vy, ω) entre las cuatro ruedas; el adaptador de entrada aplica deadzones, escalado y saturación antes de publicar consignas por rueda. Este desac acople permitió portar la lógica en tiempos muy cortos y con mínima superficie de cambio.

Sensado y control. Al igual que MiniPampa, no se incluyen tacómetros en esta primera iteración: la traslación es lazo abierto. Sí se reutiliza la IMU/giroscopio para estabilizar el yaw (opcional “heading-hold” con PID), lo que mejora la maniobrabilidad pese a las variaciones de carga o piso propias de una plataforma tan liviana.