UACJ | Universidad Autónoma de Ciudad Juárez

DMCU

UNIVERSIDAD AUTÓNOMA DE CIUDAD JUÁREZ

DIVISIÓN MULTIDISCIPLINARIA CIUDAD UNIVERSITARIA

Curso: Diseño Mecatrónico

Grupo: C

Diseño de un Robot Manipulador Esférico Paralelo

Fecha de entrega:
19 de Noviembre del 2025

Docente:

Dr. Francesco José Garcia Luna

Alumnas:

Angelina E. Castillo Rodarte 174885
Fernanda A. Castañeda Huerta 192771
Guadalupe Sanchez Martinez 207164
Krystal Garcia Cruz 201430

Índice general

Capítulo 1

Introducción

Un Manipulador Esférico Paralelo(SPM por sus siglas en inglés) proporciona movimiento de orientación puro del efector final alrededor de un centro fijo y precisión para tareas de orientación rápida y dinámica. A diferencia de manipuladores seriados, un SPM suele ofrecer una mayor relación rigidez/peso y mejores prestaciones dinámicas, lo que los hace adecuados para aplicaciones de precisión como estabilización, seguimiento de orientación y ensamblado [1].

En este proyecto se desarrolla un SPM de 3 grados de libertad (3-DOF) no planar, controlado por actuadores NEMA 17; paso a paso mediante drivers A4988, con adquisición de orientación/aceleración por un sensor IMU MPU-9250 y una interfaz basada en Arduino con CNC Shield para facilitar el montaje modular y el control de los drivers.

El presente documento describe el diseño e implementación de un SPM. Para este proyecto, la estabilización se lleva a cabo mediante el uso de una placa de desarrollo como el Arduino Mega. La arquitectura propuesta integra herramientas de software robótico (ROS2 , micro-ROS , RVIZ) en un entorno Linux , creando un ecosistema que vincula el modelo físico con la simulación virtual. En las siguientes secciones se muestran los fundamentos técnicos, el diseño del sistema así como la simulación, el código, los modelos y los recursos generados.

Capítulo 2

Antecedentes

Los SPM son robots en los que el efector final se mueve únicamente por rotaciones alrededor de un punto fijo, sin desplazamiento lineal. Este tipo de arquitectura permite obtener una alta rigidez estructural, bajo peso relativo y una excelente respuesta dinámica.

Además, el estudio del SPM proporciona una base teórica sólida para comprender la importancia del diseño geométrico, la formulación cinemática y el análisis de desempeño, aspectos esenciales en el desarrollo del presente proyecto.

Diversas investigaciones recientes han explorado el diseño, control y desempeño de manipuladores no planares. Entre ellas, destacan dos enfoques principales, Los sistemas robóticos aplicados a la impresión 3D no planar, que buscan mejorar la calidad estructural y estética de las piezas mediante trayectorias tridimensionales. Los SPM, que ofrecen una arquitectura mecánica compacta y precisa para el control de orientación.

El trabajo de Sadeqi [1], presenta el diseño y análisis de desempeño de un manipulador paralelo esférico tipo 3-RRR, abordando los siguientes aspectos fundamentales como Modelado cinemático directo e inverso, que describe con precisión la orientación del efector final, el Análisis dinámico a partir de ecuaciones de movimiento, considerando la interacción entre las articulaciones y la geometría del mecanismo y la evaluación de desempeño mediante parámetros como rigidez, precisión y rango de movimiento.

El segundo artículo propone un método de generación de trayectorias basado en posiciones para guiar el movimiento del robot durante la deposición del material. Este enfoque se caracteriza por su adaptabilidad a geometrías curvas y superficies complejas, así como por la optimización del control del extrusor en robots con 3 GDL, lo que permite un mejor acabado superficial y una mayor estabilidad estructural.

Capítulo 3

Objetivos

3.1. Objetivo General

Diseñar un robot manipulador esférico paralelo de 3 grados de libertad.

3.2. Objetivos Específicos

- Investigar los componentes adecuados para la construcción de un robot manipulador esférico paralelo de 3 grados de libertad.
- Calcular la cinemática inversa de un robot manipulador esférico paralelo de 3 grados de libertad.
- Integrar el robot Manipulador Esférico Paralelo.
- Validar el robot Manipulador Esférico Paralelo.

Capítulo 4

Marco teórico

4.1. ROS2

ROS2 (por sus siglas en inglés, Robot Operating System 2) es un conjunto de librerías y herramientas de software para construir aplicaciones robóticas. Desde controladores y algoritmos de vanguardia hasta herramientas de desarrollo potentes, ROS2 proporciona las herramientas de código abierto necesarias para proyectos robóticos. Desde su inicio en 2007, la comunidad de ROS ha evolucionado significativamente, y el objetivo del proyecto ROS2 es adaptarse a estos cambios, aprovechando lo mejor de ROS1 y mejorando lo que no funcionaba adecuadamente [2].

RViz2 es un puerto de RViz para ROS2 que funciona como visualizador 3D de robots. Proporciona una interfaz gráfica para que los usuarios puedan ver su robot, datos de sensores, mapas y más. Se instala por defecto con ROS2 y requiere una versión de escritorio con un entorno de escritorio o administrador de ventanas de Linux como Fedora o Arch Linux, pero preferentemente Ubuntu [3].

Los proyectos de ROS2 se organizan en paquetes, son unidades de software que contienen código, datos y otros recursos. Para crear un paquete de ROS2, se utiliza el comando ros2 pkg create, que genera la estructura básica del paquete. Los paquetes pueden contener nodos, bibliotecas, archivos de configuración y otros recursos necesarios para el funcionamiento del sistema robótico [4].

Capítulo 5

Metodología

El diagrama de flujo del proyecto constituye una representación visual que facilita la comprensión clara y organizada de las etapas que integran el desarrollo del sistema. En esta estructura se evidencia la secuencia lógica de operación, donde el programa ejecuta de manera iterativa los procesos de lectura, procesamiento, toma de decisiones y actuación. La Figura 5.1 presenta dicha representación, en la cual se destacan los bloques fundamentales del sistema y la interrelación que existe entre ellos. Además de proporcionar una visión general del funcionamiento del sistema, este diagrama cumple la función de guiar al lector a lo largo de la metodología, permitiendo identificar de manera intuitiva las dependencias entre etapas, los puntos de retroalimentación y las decisiones críticas que influyen en el desarrollo del proyecto. A continuación, se describen las etapas que conforman el flujo metodológico del proyecto:

1. Inicio del proyecto En esta etapa se da comienzo formal al proyecto. Se establecen los objetivos principales, el alcance y los recursos necesarios para llevar a cabo el desarrollo del manipulador robótico.

2. Diseño del modelo CAD en SolidWorks: Se realiza la creación del modelo tridimensional del manipulador utilizando el software SolidWorks. Aquí se definen las dimensiones, geometrías y componentes estructurales que formarán parte del prototipo final.

3. Análisis cinemático inverso para determinar las posiciones de los motores en función de la orientación: En esta fase se lleva a cabo el cálculo matemático de la cinemática inversa. Su propósito es determinar los ángulos o posiciones que deben adoptar los motores para alcanzar una posición y orientación deseada del efector final.

4. Comprobación de la cinemática inversa: Se evalúa si los resultados obtenidos del análisis cinemático inverso son correctos y coherentes con el modelo diseñado. En caso de detectar inconsistencias, se regresa a la etapa de análisis para realizar ajustes y garantizar el buen funcionamiento del manipulador.

5. Fabricación y ensamble de las piezas estructurales mediante impresión 3D: Una vez validado el análisis cinemático, se procede a la fabricación de las piezas estructurales mediante impresión 3D. Posteriormente, se realiza el ensamble físico de todos los componentes mecánicos que conforman el manipulador.

6. Implementación del sistema de control electrónico: Se integran los sistemas electrónicos encargados del control del manipulador, como microcontroladores, drivers de motores, sensores y el cableado correspondiente. También se desarrolla la programación necesaria para sincronizar el funcionamiento entre la electrónica y la mecánica del sistema.

7. Pruebas de validación para evaluar la precisión y el rango de movimiento del manipulador: En esta etapa se realizan pruebas experimentales para comprobar el desempeño del manipulador. Se evalúa si los movimientos obtenidos corresponden a los esperados, midiendo precisión, repetibilidad y alcance.

8. Fin del proyecto: En esta fase se documentan los resultados y se cierran todas las actividades correspondientes al desarrollo del manipulador robótico.

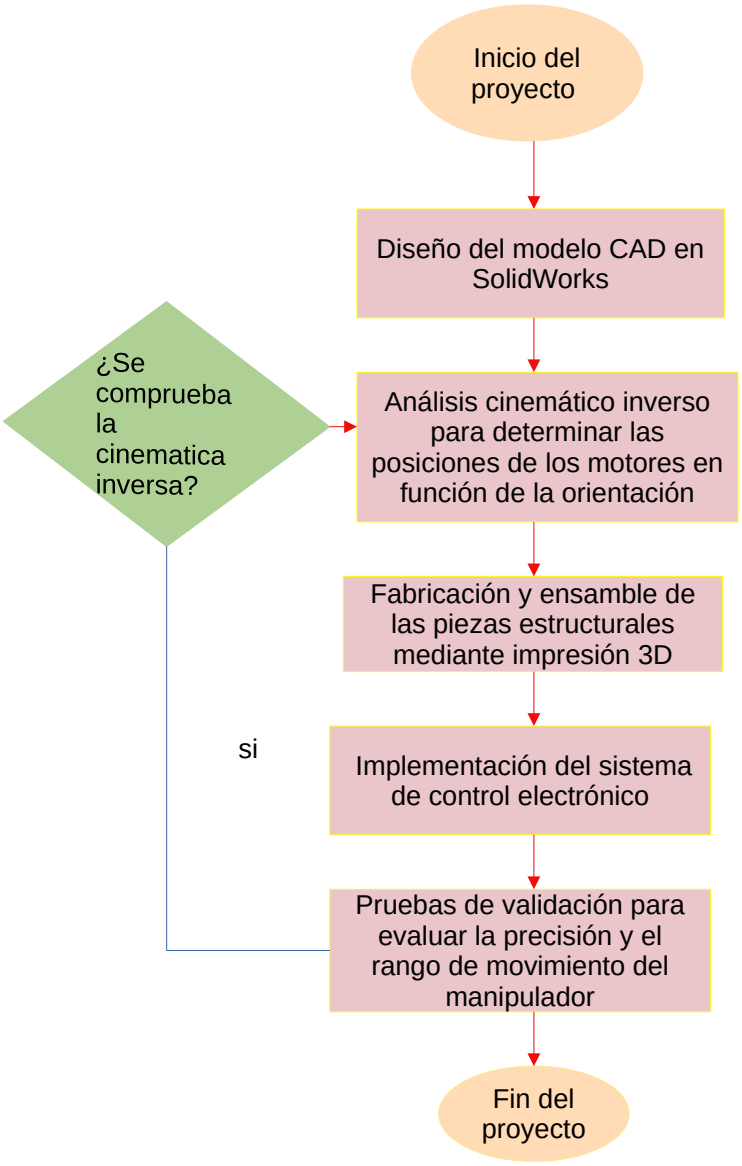


Figura 5.1: Diagrama de flujo

5.1. Descripción de la Plataforma

La plataforma corresponde a SPM de tres grados de libertad , diseñada para realizar movimientos de orientación en el espacio sin desplazar el centro de la plataforma móvil. Este mecanismo resulta ideal para aplicaciones de precisión donde se requiere un control exacto de la orientación.

El uso de software de modelado y simulación permitió optimizar la geometría de los eslabones y la ubicación de los actuadores, lo que resultó en un mayor espacio de trabajo y en el correcto funcionamiento del mecanismo.

Para el desarrollo de un SPM se requiere una combinación de componentes electrónicos y mecánicos que permitan su correcto funcionamiento. El sistema necesita una unidad de control que procese las señales de entrada, ejecute los algoritmos de control y coordine el movimiento de los actuadores. Además, se incluyen módulos de accionamiento que actúan como intermediarios entre la unidad de control y los motores, regulando la potencia y el sentido de giro.

Los actuadores son los encargados de generar el movimiento en las articulaciones del robot, garantizando precisión en la posición del efector final. Para el sensado, se emplean sensores que proporcionan información sobre la orientación, posición o velocidad del sistema, lo cual resulta esencial para la retroalimentación y el control del movimiento. Finalmente, el robot cuenta con una estructura mecánica formada por eslabones, uniones y soportes, que proporcionan estabilidad y rigidez estructural al conjunto.

5.2. Diseño electrónico

El diseño electrónico del robot SPM se fundamenta en la integración coordinada de diversos módulos, cuyo propósito es asegurar que el efector final pueda desplazarse libremente a través de cada uno de los eslabones del mecanismo. Para lograrlo, se incorporaron motores paso a paso, ya que permiten controlar con precisión la posición y el movimiento angular de cada articulación.

En el Anexo 3, figura 13 se encuentra el esquema principal, donde se detalla la conexión de un motor NEMA 17 hacia la Shield CNC V3, la cual opera en conjunto con un solo driver A4988. Esta etapa del diseño permite visualizar de forma clara cómo se distribuyen las señales de control y la alimentación hacia uno de los motores del robot.

En la Figura 14 se presenta una vista más completa del sistema electrónico. Allí se muestra la conexión de los tres drivers A4988 con sus respectivos motores NEMA 17, evidenciando la estructura que se utilizará para generar el movimiento coordinado de todo el robot. Además, se aprecia la posición asignada al sensor destinado al efector final, el cual se integrará para proporcionar retroalimentación o detección según las necesidades operativas del robot.

En conjunto, estas figuras y el diseño electrónico explican la arquitectura general del sistema de control, para una distribución adecuada de señales, potencia y elementos sensores, lo que permite un funcionamiento estable y preciso del robot SPM.

5.3. Diseño 3D

El diseño tridimensional del robot se desarrolló utilizando software CAD paramétrico, lo que permitió ajustar dimensiones, tolerancias y uniones mecánicas de manera iterativa. El modelo final incluye tanto la estructura principal como los elementos móviles y de soporte, asegurando compatibilidad entre todas las piezas antes del proceso de fabricación.

Los planos y archivos de fabricación, desglosados por pieza, se encuentran en el Anexo 2 e incluyen las vistas, cotas, especificaciones geométricas y tolerancias recomendadas para cada componente. Adicionalmente, se añadieron referencias de ensamblaje para facilitar la correcta orientación de las partes durante el montaje.

Las piezas fueron fabricadas mediante impresión 3D utilizando material PLA, prácticamente listo para el ensamblaje, con los elementos de fijación y transmisión: tornillos, tuercas, collarines y baleros, los cuales son necesarios para unir y asegurar correctamente las piezas impresas y los ejes motrices.

5.4. Análisis de la Cinemática inversa

Los robots paralelos, a diferencia de los robots seriales, están formados por varios eslabones que conectan la base con el efector final. Esta configuración permite distribuir el peso entre las piernas, otorgando al sistema una mayor rigidez y precisión. Entre los manipuladores paralelos más conocidos se encuentran el robot Delta, el robot Stewart y el manipulador esférico paralelo (SPM), que es el tipo utilizado en este proyecto.

El manipulador SPM de tres grados de libertad (3-DOF) se caracteriza por permitir la rotación del efector final alrededor de un punto fijo, conocido como centro de rotación. Cada una de las tres piernas se mueve mediante un motor paso a paso. Gracias a esta configuración, el SPM puede generar movimientos suaves y coordinados en los tres ejes de rotación: x, y y z.

La cinemática inversa es el proceso mediante el cual se calculan los ángulos de las articulaciones o los pasos de los motores necesarios para alcanzar una posición u orientación deseada. En el caso del SPM, las variables de entrada son las coordenadas del efector final (x, y, z), mientras que las variables de salida son los ángulos de los tres motores $(\theta_1, \theta_2, \theta_3)$. Se obtienen las relaciones matemáticas que permiten determinar cuánto debe girar cada motor.

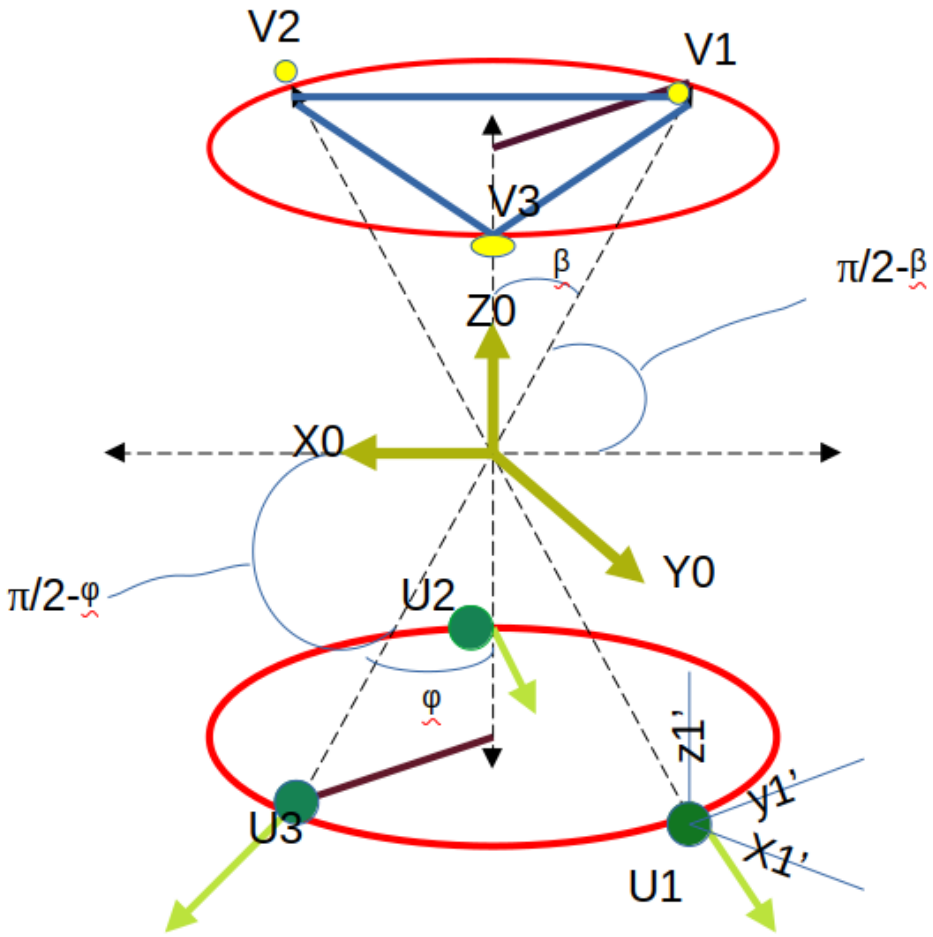


Figura 5.2: **Representación geométrica de un manipulador esférico de tres grados de libertad.**

En la Figura 5.2 se aprecia que los puntos U_1, U_2 y U_3 se encuentran sobre una circunferencia fija asociada al plano de la base, mientras que los puntos V_1, V_2 y V_3 pertenecen a la circunferencia móvil correspondiente a la plataforma. El punto O representa el centro esférico común a todas las articulaciones. El movimiento de orientación del efector final se produce por las rotaciones independientes de cada actuador $\mathbf{q}_i$ en torno a su eje $\mathbf{u}_i$.

A continuación, se definen las principales variables empleadas en el análisis cinemático inverso del SPM:

- $\mathbf{r}_i$: Vector que une el centro O con la posición inicial.
- $\mathbf{p}_i$: Base frame.
- $\mathbf{P}'_i$: El vector de la plataforma.
- **i** : Es el motor.
- **O** : Centro esférico.
- $\mathbf{q}_i$: El angúlo que debe girar el motor i.
- R : Matriz de rotación 3×3 que representa la orientación deseada de la plataforma respecto al marco base.

- $\mathbf{u}_i$: Vector unitario que indica el eje de rotación de la junta $\mathbf{u}_i$
- $\mathbf{r}_i \times \mathbf{r}'_i$: Es el vector perpendicular al plano formado por las posiciones inicial y final.
- $\mathbf{r}_i.\mathbf{r}'_i$: Es el coseno del ángulo entre ambos vectores.

Para el desarrollo de este proyecto, se implementó una versión simplificada del modelo cinemático inverso, en la cual los cálculos se realizan directamente en el microcontrolador Arduino. De esta manera, el sistema convierte las coordenadas deseadas en los desplazamientos angulares necesarios para cada motor, los cuales son transformados posteriormente en pasos mediante los controladores A4988.

Capítulo 6

Resultados

6.1. Componentes de hardware utilizados

En esta sección se presentan los componentes seleccionados para la implementación del SPM. Los elementos fueron elegidos considerando su disponibilidad, compatibilidad y desempeño dentro del sistema.

La IMU MPU-9250 se seleccionó por su capacidad para medir aceleraciones, velocidades angulares y campo magnético en tres ejes, lo que permite una estimación precisa de la orientación del efector final. Su integración de nueve grados de libertad y su bajo consumo energético la convierten en una opción adecuada para sistemas de control.

El driver A4988 fue elegido por su simplicidad de configuración y su capacidad para controlar motores paso a paso con diferentes modos de microstepping. Esto posibilita un movimiento más suave y preciso de los actuadores, facilitando la sincronización entre las piernas del manipulador.

Se utilizaron motores paso a paso NEMA 17 debido a su equilibrio entre torque, tamaño y precisión angular. Estos motores son ampliamente compatibles con diversos controladores y ofrecen la resolución necesaria para el posicionamiento del efector final dentro del espacio de trabajo del robot.

Finalmente, se incorporó una CNC Shield v3 para Arduino como interfaz de conexión, ya que permite integrar fácilmente los drivers y los motores con la unidad de control. Su diseño modular simplifica el cableado y la configuración del sistema, además de ser compatible con múltiples plataformas de control basadas en Arduino.

Cuadro 6.1: Características de los componentes

MPU-9250	A4988	Motor NEMA 17	CNC Shield (v3)
Módulo IMU 9 ejes	Driver DMOS Microstepping.	Tamaño 43.2 x 43.2 mm	Shield modular utilizando Arduino para controlar 4 ejes.
Convertidores ADC de 16 bits.	Alimentación de 8 V a 35 V.	Típicamente 200 pasos por revolución (1.8º por paso).	Sockets, pines y librerías.
Giroscopio seleccionable.	Disipación variable con refrigeración o sin ventilación.	Cambian según corriente nominal, torque y longitud del eje.	
Sensor de 9 ejes integrado para fusión de sensores e interrupciones por movimiento.	Limitación y protección de corriente con control decay.		
Calibración de componentes para uso en estimaciones de actitud.	Ajustar la corriente del motor.		

6.2. Construcción del robot

La construcción del SPM de tres grados de libertad se realizó una vez finalizados los diseños electrónico y mecánico. En esta etapa se integraron las piezas impresas en 3D con los componentes electrónicos, asegurando la correcta alineación de los ejes de rotación en el punto O del manipulador.

El proceso de ensamblaje se inició con la base, donde se montaron los tres motores paso a paso NEMA 17 sobre el soporte del motor. Cada motor se acopló a su brazo inferior, garantizando el movimiento angular. Posteriormente, se ensamblaron los brazos intermedios y la plataforma móvil, verificando que la geometría del sistema mantuviera la simetría de 120° entre las piernas.

Durante la fase de construcción, se emplearon elementos de fijación como tornillos, tuercas, collarines y baleros, los cuales permitieron reducir el juego mecánico y mejorar la estabilidad.

En la figura 6.1 se presenta el modelo 3D del robot SPM, donde se identifican las piezas mecánicas y las estructuras impresas en 3D que componen los eslabones y el soporte del efector final.

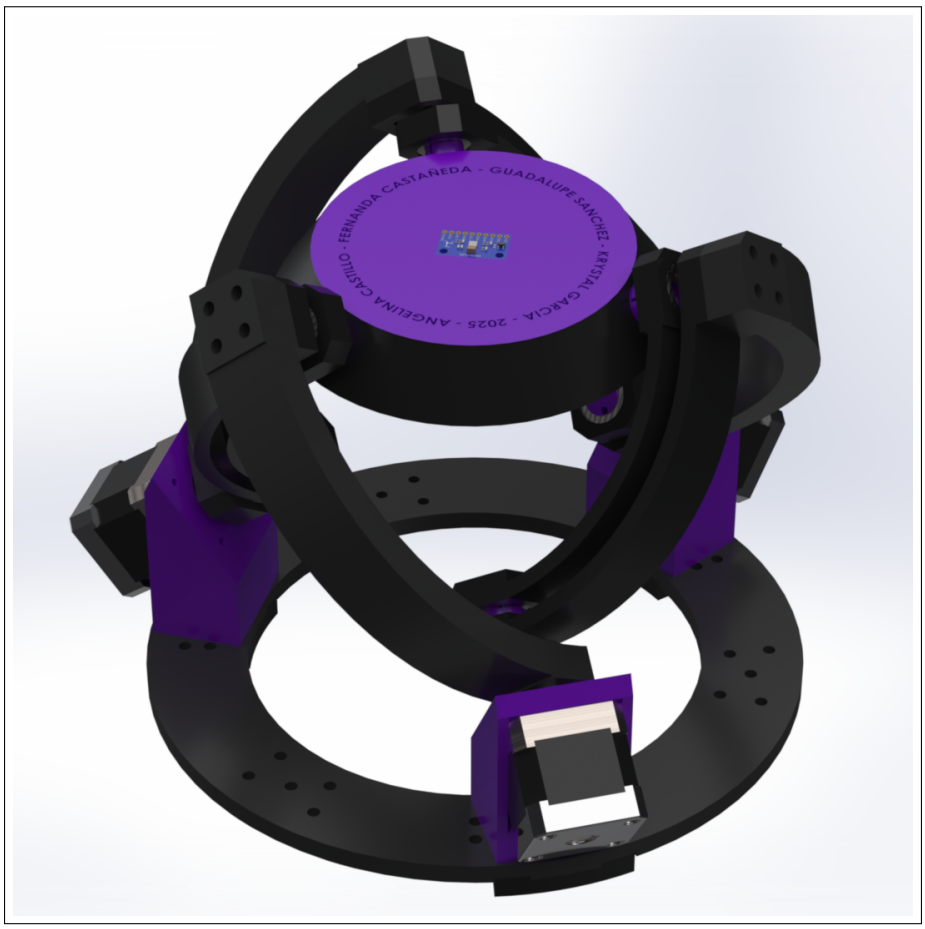


Figura 6.1: Render del Robot SPM

6.3. Cinemática inversa de un SPM (3-DOF)

La cinemática inversa en un SPM de tres grados de libertad consiste en determinar los ángulos de las articulaciones que permiten alcanzar una orientación deseada del efector final. Para resolver la cinemática inversa se implica obtener los ángulos articulares necesarios para lograr una rotación específica en torno a los ejes x, y y z.

Consideramos un manipulador esférico con centro de rotación en el punto O. Cada una de las tres piernas conecta la base de manera que el efector final tiene solo orientación (no traslación) respecto a O.

El vector del punto de la plataforma $\mathbf{p}'_i$ se obtiene aplicando la orientación de la plataforma mediante la matriz de rotación R, tal como se muestra en la Ecuación (6.1):

$$\mathbf{p}'_i = R\,\mathbf{p}_i. \tag{6.1}$$

Esta ecuación indica que la posición del punto de la plataforma en el espacio depende de la plataforma móvil. Cada motor i tiene un eje de rotación definido por el vector $\mathbf{u}_i$, y el punto de la plataforma correspondiente se encuentra a lo largo de dicha dirección desde el centro O. Para analizar el movimiento en el plano perpendicular a $\mathbf{u}_i$, se construyen los vectores, asi como la Ecuación (6.2) y (6.3):

$$\mathbf{a}_{i,\perp} = \mathbf{r}_i - (\mathbf{u}_i \cdot \mathbf{r}_i)\,\mathbf{u}_i, \tag{6.2}$$

$$\mathbf{b}_{i,\perp} = \mathbf{r}'_i - (\mathbf{u}_i \cdot \mathbf{r}'_i)\,\mathbf{u}_i. \tag{6.3}$$

Estas expresiones eliminan la componente del vector sobre el eje $\mathbf{u}_i$, dejando únicamente su proyección en el plano de rotación del motor. El ángulo actuador q_i se define como el giro necesario del eje $\mathbf{u}_i$ para pasar de la posición inicial $\mathbf{r}_i$ a la posición final $\mathbf{r}'_i$. Este ángulo puede obtenerse a partir del producto vectorial y escalar de los vectores proyectados, mediante la función atan2, que permite determinar el signo del giro:

$$q_i = \text{atan2}(\mathbf{u}_i \cdot (\mathbf{a}_{i,\perp} \times \mathbf{b}_{i,\perp}),\ \mathbf{a}_{i,\perp} \cdot \mathbf{b}_{i,\perp})\,. \tag{6.4}$$

De manera equivalente, expresando el ángulo directamente con los vectores $\mathbf{r}_i$ y $\mathbf{r}'_i$:

$$q_i = \text{atan2}(\mathbf{u}_i \cdot (\mathbf{r}_i \times \mathbf{r}'_i),\ \mathbf{r}_i \cdot \mathbf{r}'_i)\,. \tag{6.5}$$

Esta última forma, Ecuación (6.5), es especialmente útil en la implementación computacional, ya que utiliza directamente las posiciones iniciales y finales de los puntos. Por tanto, el conjunto de ecuaciones desde (6.1) hasta (6.5) permite determinar el ángulo que debe girar cada motor i en función de la orientación deseada de la plataforma y la geometría del sistema.

6.4. Conexión Eléctrica

La conexión eléctrica constituye una etapa fundamental en la integración del robot SPM, ya que permite enlazar los elementos electrónicos responsables del control y la alimentación del sistema. En esta fase se establecieron las conexiones entre los actuadores, los módulos de control y la unidad central, asegurando el correcto funcionamiento del manipulador.

Una vez ensamblado el sistema, se procedió a realizar la conexión de los motores a la Shield CNC V3, empleando tres drivers A4988 configurados con microstepping de 1/16 de paso. El control de los motores se efectuó mediante un microcontrolador Arduino, encargado de ejecutar los algoritmos de cinemática inversa y generar las señales de control necesarias para el movimiento coordinado del efector final.

Para los controladores de motores paso a paso como el A4988, la corriente máxima del motor depende directamente del voltaje de referencia. Este voltaje se ajusta mediante un potenciómetro ubicado en el driver, el cual regula la corriente midiendo la caída de tensión sobre una resistencia R_s. La relación general entre el voltaje de referencia y la corriente máxima de fase se expresa mediante la Ecuación (6.6):

$$V_{\text{ref}} = I_{\max} \cdot K_A, \tag{6.6}$$

donde $I_{\max}$ es la corriente máxima permitida en cada fase del motor y K_A es una constante que depende de la resistencia de sensado. Para el caso del driver A4988, dicha constante está dada por:

$$K_A = 8\, R_s, \tag{6.7}$$

ya que el circuito interno del A4988 multiplica la tensión medida en las resistencias R_s por un factor aproximado de 8 para determinar el límite de corriente. Sustituyendo la Ecuación (6.7) en la Ecuación (6.6), se obtiene la relación práctica:

$$V_{\text{ref}} = I_{\max} \cdot 8\, R_s. \tag{6.8}$$

De este modo, el voltaje de referencia depende tanto de la corriente deseada como del valor de las resistencias del driver.

Para el procedimiento, un motor NEMA 17 cuya corriente por fase es $I_{\text{motor}} = 0{,}4$ A y el driver A4988 tiene resistencias $R_s = 0{,}1\ \Omega$. Sustituyendo estos valores en la Ecuación (6.8), se obtiene:

$$V_{\text{ref}} = 0{,}4\text{ A} \times 8 \times 0{,}1\ \Omega = 0{,}32\text{ V}. \tag{6.9}$$

Sin embargo, en la Ecuación 6.9 se obtiene el valor de voltaje de referencia, para evitar sobrecalentamiento, es común ajustar el voltaje de referencia para limitar la corriente a un porcentaje menor, por ejemplo al 70 %. Aplicando este ajuste, se tiene:

$$V_{\text{ref, ajustado}} = 0{,}32 \times 0{,}7 = 0{,}22\text{ V}. \tag{6.10}$$

En la práctica, este valor que se obtuvo en la Ecuación 6.10 se mide con un multímetro mientras se ajusta el potenciómetro del driver.

En motores paso a paso, la posición angular se mide en radianes, por lo que es necesario convertir los pasos por revolución en unidades angulares. La conversión básica de grados a radianes se expresa mediante la Ecuación 6.11:

$$1^\circ = \frac{\pi}{180}\text{ rad}. \tag{6.11}$$

Si el motor tiene N_{steps} pasos por revolución, el desplazamiento angular correspondiente a un paso completo se obtiene con la Ecuación (6.12):

$$\theta_{\text{step}}^{\text{rad}} = \frac{2\pi}{N_{\text{steps}}}. \tag{6.12}$$

Por ejemplo, en un motor NEMA 17 de 200 pasos por revolución, el desplazamiento angular correspondiente a un paso completo se obtiene aplicando la Ecuación (6.12):

$$\theta_{\text{step}}^{\text{rad}} = \frac{2\pi}{200} \approx 0{,}0314 \text{ rad/paso}. \tag{6.13}$$

Este valor indica que cada pulso que recibe el driver produce un giro de aproximadamente 0,0314 rad (equivalente a 1,8°). Cuando se activa el modo de microstepping con un factor M, el desplazamiento angular por micropasos se reduce de forma proporcional. En este caso, la relación se determina mediante la Ecuación (6.14):

$$\theta_{\text{microstep}}^{\text{rad}} = \frac{\theta_{\text{step}}^{\text{rad}}}{M} = \frac{2\pi}{N_{\text{steps}} \cdot M}. \tag{6.14}$$

De esta manera, al aumentar el valor de M, se mejora la resolución angular del sistema, permitiendo movimientos más precisos.

Para alcanzar un ángulo deseado $\theta_{\text{deseado}}^{\text{rad}}$, es necesario calcular el número total de micropasos requeridos. Este valor se obtiene al dividir el ángulo objetivo entre el desplazamiento angular por micro-paso, como se muestra en la Ecuación (6.15):

$$N_{\text{req}} = \frac{\theta_{\text{deseado}}^{\text{rad}}}{\theta_{\text{microstep}}^{\text{rad}}}. \tag{6.15}$$

Sin embargo, dado que el número de pasos que el motor puede ejecutar debe ser un número entero, el controlador debe redondear el valor obtenido en la Ecuación (6.15). El número real de pasos ejecutados se define entonces mediante la Ecuación (6.16):

$$N_{\text{real}} = \text{round}(N_{\text{req}}), \tag{6.16}$$

lo cual significa que el motor realizará el número entero de micro-pasos más cercano al requerido. A partir de este valor, se puede determinar el ángulo realmente alcanzado, aplicando la Ecuación (6.17):

$$\theta_{\text{real}}^{\text{rad}} = N_{\text{real}} \cdot \theta_{\text{microstep}}^{\text{rad}}. \tag{6.17}$$

Finalmente, la diferencia entre el ángulo deseado y el ángulo realmente alcanzado constituye el error angular del sistema. Este error se expresa mediante la Ecuación (6.18):

$$e^{\text{rad}} = \theta_{\text{deseado}}^{\text{rad}} - \theta_{\text{real}}^{\text{rad}}. \tag{6.18}$$

Para el funcionamiento básico del control del motor paso a paso, se elaboró un pseudocódigo que resume la lógica empleada para convertir un ángulo solicitado en radianes en la cantidad de pasos que el motor debe ejecutar. Este pseudocódigo permite explicar de manera clara el flujo del programa, en él se definen las constantes principales del motor, el cálculo de la resolución angular por micro–paso y el procedimiento para determinar los pasos necesarios según la diferencia entre el ángulo actual y el deseado. De esta forma, el pseudocódigo sirve como referencia conceptual del comportamiento del controlador mostrado en el Algoritmo 6.1.

Listing 6.1: Control de motor paso a paso con BasicStepperDriver y ángulos en radianes

```
// -------------------------------------------------------
// Configuracion inicial
// -------------------------------------------------------
SubProceso setup
        // -------------------------------------------------------
        // Control de motor paso a paso por angulo en radianes
           Pseudocodigo
        // -------------------------------------------------------
        // Constantes del motor// Pines del motor
        Definir MOTOR_STEPS, rpm, MICROSTEPS, DIR, STEP, ENABLE Como
           Entero;
        MOTOR_STEPS <- 200;
        rpm <- 60;
        MICROSTEPS <- 16;
        DIR <- 7;
        STEP <- 9;
        ENABLE <- 8;
        // Variables del programa
        Definir theta_step, theta_micro, current_angle, target_angle
           Como Real;
        Definir delta_angle, steps_required, steps_to_move, real_angle,
            error Como Real;
        IniciarComunicacionSerial(9600);
        // Inicializacion del objeto motor (simulada)
        Definir stepper Como Real;
        // steper.
        iniciar(rpm,MICROSTEPS);
        definirEstadoActivoDeEnable(LOW);
        habilitar();
        theta_step <- (2*PI)/MOTOR_STEPS;
        theta_micro <- theta_step/MICROSTEPS;
        current_angle <- 0;
        Escribir ('==== CONTROL DE ANGULOS EN RADIANES ====');
        Escribir ('Escribe el angulo en radianes y presiona ENTER (ej:
           0.87)');
        Escribir ('-----------------------------------------------------')
           ;
FinSubProceso

// -------------------------------------------------------
// Bucle principal
// -------------------------------------------------------
Algoritmo loop
        Definir target_angle, delta_angle, steps_required,
           steps_to_move Como Real;
        Definir real_angle, error, current_angle, theta_micro Como Real
           ;
        Definir serial_disponible Como Logico;
        current_angle <- 0;
        theta_micro <- 0.01745;
        serial_disponible <- Verdadero; // Ejemplo: 1 grado en radianes
            por paso
```

```
        Si serial_disponible=Verdadero Entonces // Simulacion de que
           hay datos disponibles
              Escribir 'Ingrese el angulo deseado (en radianes): ';
              Leer target_angle;
              delta_angle <- target_angle-current_angle;
              steps_required <- delta_angle/theta_micro;
              steps_to_move <- Redon(steps_required);
              real_angle <- steps_to_move*theta_micro;
              error <- delta_angle-real_angle;
              // Simulamos el movimiento del motor
              current_angle <- current_angle+real_angle;
              Escribir '==== RESULTADOS ====';
              Escribir 'Angulo deseado: ', target_angle;
              Escribir 'Resolucion angular: ', theta_micro, ' rad/
                 step';
              Escribir 'Pasos calculados: ', steps_required;
              Escribir 'Pasos ejecutados: ', steps_to_move;
              Escribir 'Angulo real movido: ', real_angle;
              Escribir 'Error: ', error;
              Escribir 'Posicion acumulada: ', current_angle;
              Escribir '----------------------';
        FinSi
FinAlgoritmo

SubProceso IniciarComunicacionSerial(baudios)

FinSubProceso

SubProceso SerialDisponible

FinSubProceso

SubProceso iniciar(rpm,mcstep)

FinSubProceso

SubProceso definirEstadoActivoDeEnable(var)

FinSubProceso

SubProceso habilitar

FinSubProceso
```

El pseudocódigo presentado en el Algoritmo 6.2 describe el funcionamiento básico del control del motor paso a paso correspondiente al eje Y empleando una CNC Shield y un Arduino Mega 2560.

Su propósito es ilustrar de forma clara la secuencia de instrucciones necesarias para generar movimiento alternado en ambos sentidos de giro.

En este algoritmo se definen los pines utilizados para el paso, la dirección y la habilitación del motor, así como las variables de tiempo que determinan la velocidad del movimiento.

Posteriormente, el programa principal ejecuta un bucle infinito en el que se llama a un subproceso encargado de generar los pulsos adecuados para el driver.

Este subproceso controla la activación del motor, establece la dirección de giro y ejecuta una serie de pasos en un sentido y luego en el contrario, simulando un movimiento oscilatorio.

Listing 6.2: Control de motor paso a paso en el eje Y con CNC Shield y Arduino Mega 2560 pseudocodigo

```
// --- Programa principal ---
Algoritmo Principal
        // Declaracion de constantes (pines) y variables
        Definir y_paso, y_dire, y_habi, retardo, tiempo Como Entero;
        // Variables de tiempo
        y_paso <- 5;
        y_dire <- 3;
        y_habi <- 8;
        retardo <- 2000;
        tiempo <- 120;
        // Configurar pines como salida
        //
        ConfigurarPin(y_paso,SALIDA);
        ConfigurarPin(y_dire,SALIDA);
        ConfigurarPin(y_habi,SALIDA);
        // Ciclo infinito
        Mientras Verdadero Hacer
                Llamar_giro(y_paso,y_dire,y_habi);
        FinMientras
FinAlgoritmo

// --- Subproceso para hacer girar el motor ---
SubProceso giro(paso_,dire_,habi_)
        // Deshabilitar motor
        EscribirPin(habi_,LOW);
        // Establecer direccion hacia un lado
        EscribirPin(dire_,LOW);
        // Hacer tiempo pasos en esa direccion
        Para i<-1 Hasta tiempo Con Paso 1 Hacer
                EscribirPin(paso_,HIGH);
                EsperarMicrosegundos(retardo);
                EscribirPin(paso_,LOW);
                EsperarMicrosegundos(retardo);
        FinPara
        // Cambiar direccion al lado opuesto
        EscribirPin(dire_,HIGH);
        // Repetir los pasos en la direccion contraria
        Para i<-1 Hasta tiempo Con Paso 1 Hacer
                EscribirPin(paso_,HIGH);
                EsperarMicrosegundos(retardo);
                EscribirPin(paso_,LOW);
                EsperarMicrosegundos(retardo);
        FinPara
        // Habilitar motor nuevamente
        EscribirPin(habi_,HIGH);
        // Pausar 1 segundo
        Esperarval(1);
FinSubProceso // segundos

```

```
50 SubProceso ConfigurarPin(ysub,output)
51
52 FinSubProceso
53
54 SubProceso Llamar_giro(ypaso,ydire,yhabi)
55
56 FinSubProceso
57
58 SubProceso EscribirPin(pin,bool)
59
60 FinSubProceso
61
62 SubProceso EsperarMicrosegundos(rest)
63
64 FinSubProceso
65
66 SubProceso Esperarval(val)
67
68 FinSubProceso
```

6.5. Recolección de resultados

En esta etapa se implementó la comunicación entre los sensores y el entorno de simulación a través de nodos desarrollados en *ROS2*. En particular en el Anexo 1, se muestra el diseñó de un nodo dedicado a la adquisición de datos provenientes del sensor MPU-9250, el cual permite obtener información de aceleración, velocidad angular y campo magnético en tres ejes.

El sensor fue conectado a un microcontrolador Arduino, encargado de realizar la lectura continua de los valores crudos generados por el MPU-9250. Estos datos son posteriormente enviados hacia una interfaz en Python, donde se procesan y publican en los tópicos correspondientes del entorno ROS2. De esta forma, se dispone de la información del sensor en tiempo real dentro del sistema robótico.

En el nodo de Python, se implementó la publicación de mensajes en los tópicos `/data_raw` y `/mag`, los cuales contienen los datos inerciales y de magnetómetro respectivamente. Además, se desarrolló una suscripción a dichos tópicos con el método `filter madgwick`, encargado de filtrar y suavizar las lecturas del campo magnético, mejorando la estabilidad en la estimación de la orientación.

De manera general, el flujo de comunicación se establece de la siguiente forma: el Arduino adquiere las mediciones del MPU-9250, las envía a través del puerto serial a la interfaz en Python, y este último nodo publica la información en ROS2, donde otros nodos pueden suscribirse para emplear los datos en la simulación y control del robot.

6.6. Diseño del Experimento

El diseño experimental se planteó para validar la posibilidad de convertir la orientación del control en movimientos suaves, estables y repetibles del robot mediante ROS2 y Arduino. La motivación principal surgió de la necesidad de manipular un robot de forma intuitiva, eliminando la complejidad de las interfaces convencionales basadas en pantallas o paneles industriales.

Los controles de videojuegos proporcionan una interfaz familiar y fácil de usar, sensores integrados como acelerómetros y giroscopio, un diseño ergonómico que facilita la precisión y

comodidad, y además ofrecen potencial de retroalimentación háptica. Todo esto los convierte en dispositivos ideales para la operación de robots no convencionales.

El objetivo general del experimento fue controlar un robot esférico paralelo mediante los sensores IMU del control DualSense usando ROS2 y Arduino, garantizando estabilidad, suavidad y precisión. Para lograr esto se plantearon objetivos específicos: obtener una orientación estable utilizando el filtro de Madgwick, convertir esta orientación en ángulos articulares, implementar la cinemática inversa para obtener los pasos de tres motores NEMA 17, enviar los pasos al robot mediante comunicación serial, y validar la coherencia entre la orientación real del control y el movimiento del robot.

Las variables independientes consideradas incluyeron la ganancia del filtro de orientación, el nivel de suavizado aplicado a las señales, la configuración del sensor IMU del control, el offset inicial de calibración, el microstepping configurado en los drivers y la velocidad de los motores. Por otro lado, las variables dependientes fueron la precisión angular del robot, el error acumulado en la orientación, el tiempo de respuesta entre movimiento del control y movimiento del robot, la suavidad del movimiento del manipulador y la estabilidad cuando el control se deja en reposo. Las variables de control incluyeron la rigidez del robot, el voltaje de alimentación, el tipo de motor y drivers, la posición inicial del robot y el tipo de filtro utilizado, que en este caso fue el filtro de Madgwick.

Los materiales y herramientas utilizados para el experimento fueron el control DualSense de PS5, la librería `pydualsense` con un parche aplicado para corregir un error de lectura en el IMU, ROS2 (Jazzy), el filtro de Madgwick, Arduino UNO, motores NEMA 17 con drivers A4988 y la estructura mecánica del robot esférico paralelo.

El procedimiento general consistió en la lectura del IMU del DualSense mediante `pydualsense`, la normalización de los datos del acelerómetro y giróscopo, la aplicación del filtro de Madgwick para obtener orientación en cuaterniones, y la conversión de estos cuaterniones a ángulos (roll, pitch, yaw). Posteriormente se realizó una calibración inicial de 2 segundos basada en un offset promedio, se aplicó la cinemática inversa para obtener los ángulos articulares, y se convirtieron estos ángulos a pasos mediante la expresión de la Ecuación (6.19):

$$pasos = \frac{200}{16}. \tag{6.19}$$

Finalmente, los pasos absolutos se enviaron mediante puerto serial a Arduino para lograr un movimiento sincronizado de los tres motores.

6.7. Análisis del Experimento

En esta sección se presentan los resultados obtenidos durante la manipulación del robot mediante el control DualSense, así como el análisis de desempeño, estabilidad y precisión del sistema.

Durante el procesamiento de las señales del IMU se observó que las señales del acelerómetro presentaban niveles considerables de ruido, lo cual afectaba inicialmente la estabilidad del sistema. Para mitigar este problema se aplicó un filtro suave que redujo las vibraciones, se empleó el filtro de Madgwick para obtener una orientación estable y se normalizaron los vectores para evitar saturación. Estas técnicas permitieron obtener una orientación limpia y robusta, mejorando significativamente la confiabilidad del sistema.

La calibración mediante offset promedio funcionó correctamente. Al mantener el control inmóvil durante dos segundos, se extrajo la orientación base y se ajustó a $[0, 0, 0]$, lo que permitió que el robot se ubicara correctamente en su posición HOME. Además, el sistema contaba con la posibilidad de recalibración mediante servicios ROS2, facilitando ajustes dinámicos durante la operación.

En cuanto a la cinemática inversa y el movimiento del robot, la conversión de ángulos a pasos absolutos resolvió los problemas de acumulación que ocurrían con pasos relativos. El uso de pasos absolutos eliminó el error acumulado, los motores realizaron movimientos más coherentes y predecibles, y la sincronización de los tres motores evitó comportamientos bruscos, garantizando una manipulación fluida del manipulador.

Durante el experimento se identificaron algunos problemas, como el ruido excesivo generado por el acelerómetro, vibraciones menores en la estructura mecánica, la necesidad de un retardo adicional en Arduino para evitar la saturación del búfer serial, el hecho de que el robot no siempre regresaba a HOME con precisión debido a su ganancia limitada, y que la cinemática inversa producía valores grandes en ciertos ángulos. Sin embargo, los resultados positivos superaron estas dificultades: la orientación obtenida con el filtro de Madgwick fue estable, el movimiento del robot era completamente manipulable, el sistema respondía con suavidad gracias al filtrado, RViz mostraba una orientación limpia sin movimientos bruscos y el control DualSense resultó intuitivo y agradable de usar.

Comparado con métodos tradicionales basados en interfaces industriales, el uso del control DualSense ofreció una experiencia más natural, sin necesidad de programar una interfaz gráfica compleja, y la experiencia de usuario fue significativamente superior. Finalmente, se identificaron líneas futuras de trabajo, tales como implementar retroalimentación háptica basada en un IMU colocado en el efector final, mejorar la precisión de la cinemática inversa, desarrollar una interfaz gráfica interactiva, migrar la comunicación a ROS2 serial para mayor eficiencia y aplicar la tecnología a sistemas de cámaras de seguimiento.

Bibliografía

[1] S. Sadeqi *et al.*, "Design and performance analysis of a 3-rrr spherical parallel manipulator." https://pmc.ncbi.nlm.nih.gov/articles/PMC6453095/, 2019. Accessed: 2025-09-26.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] R. Automation, "Rviz2 - user manual." https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html, 2025. Accessed: 2025-03-15.

[4] O. Robotics, "Creating a package." https://github.com/ros/solidworks_urdf_exporter, 2025. Accessed: 2025-02-25.

Anexo A. Código de python y Arduino

Listing 3: Nodo en Python para la lectura del sensor MPU-9250 y publicación en ROS2.

```python
import rclpy
from rclpy.node import Node
import serial
import time

from geometry_msgs.msg import Vector3
from sensor_msgs.msg import Imu, MagneticField
from std_msgs.msg import Header

import numpy as np

class MPU9250Node(Node):
    def __init__(self):
        super().__init__('mpu9250_node')

        self.port = '/dev/ttyACM2'
        self.baud = 115200
        self.ser = None

        self.connect_serial()

        self.timer = self.create_timer(0.1, self.read_serial)

        self.imu = Imu()
        self.mag = MagneticField()

        self.imu_pub = self.create_publisher(Imu, 'imu/data_raw', 10)
        self.mag_pub = self.create_publisher(MagneticField, 'imu/mag',
           10)

    def connect_serial(self):
        while self.ser is None:
            try:
                self.ser = serial.Serial(self.port, self.baud, timeout
                   =1)
                self.get_logger().info(f"Conectado a {self.port}")
            except serial.SerialException as e:
                self.get_logger().warn(f"No se pudo abrir {self.port}:
                   {e}")
                time.sleep(2)

    def read_serial(self):
        if self.ser is None:
            self.connect_serial()
            return

        try:
            line = self.ser.readline().decode(errors='ignore').strip()
            if not line:
                return
```

```
48
49              parts = line.split(';')
50              if len(parts) < 3:
51                  self.get_logger().warn(f"Linea incompleta ignorada: {
                        line}")
52                  return
53
54              try:
55                  accel_data = [float(v) for v in parts[0].replace('ACCEL
                        :', '').split(',')]
56                  gyro_data  = [float(v) for v in parts[1].replace('GYRO:
                        ', '').split(',')]
57                  mag_data   = [float(v) for v in parts[2].replace('MAG:'
                        , '').split(',')]
58
59                  accx, accy, accz = accel_data
60                  gx, gy, gz = np.deg2rad(gyro_data)
61                  mx, my, mz = mag_data
62
63                  header = Header()
64                  header.frame_id = 'imu'
65                  header.stamp = self.get_clock().now().to_msg()
66
67                  self.imu.header = header
68                  self.imu.angular_velocity.x = gx
69                  self.imu.angular_velocity.y = gy
70                  self.imu.angular_velocity.z = gz
71                  self.imu.linear_acceleration.x = accx
72                  self.imu.linear_acceleration.y = accy
73                  self.imu.linear_acceleration.z = accz
74
75                  self.mag.header = header
76                  self.mag.magnetic_field.x = mx
77                  self.mag.magnetic_field.y = my
78                  self.mag.magnetic_field.z = mz
79
80                  self.imu_pub.publish(self.imu)
81                  self.mag_pub.publish(self.mag)
82
83              except ValueError:
84                  self.get_logger().warn(f"No se pudo convertir a float:
                        {line}")
85                  return
86
87              self.get_logger().info(f"ACCEL: {accel_data} | GYRO: {
                    gyro_data} | MAG: {mag_data}")
88
89          except serial.SerialException as e:
90              self.get_logger().warn(f"Error de serial: {e}")
91              self.ser.close()
92              self.ser = None
93
94  def main(args=None):
```

```
    rclpy.init(args=args)
    node = MPU9250Node()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        if node.ser is not None:
            node.ser.close()
        node.destroy_node()
        rclpy.shutdown()

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

Listing 4: Nodo suscriptor en Python para escuchar el tópico `topico_steps`.

```
from rclpy.node import Node
from std_msgs.msg import String
import rclpy

class PreSubscriber(Node):

    def __init__(self):
        super().__init__('pre_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topico_steps',
            self.listener_callback,
            10)
        self.subscription  # evitar advertencia de variable no usada

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)


def main(args=None):
    rclpy.init(args=args)

    pre_subscriber = PreSubscriber()

    rclpy.spin(pre_subscriber)

    # Destruir el nodo explicitamente
    pre_subscriber.destroy_node()
    rclpy.shutdown()


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

Listing 5: Código en Arduino para la lectura del sensor MPU-9250 y envío de datos vía puerto serial.

```ino
#include <Wire.h>
#include <MPU9250_asukiaaa.h>

MPU9250_asukiaaa mySensor;

void setup() {
  Serial.begin(115200);
  Wire.begin();

  mySensor.setWire(&Wire);
  mySensor.beginAccel();
  mySensor.beginGyro();
  mySensor.beginMag();
}

void loop() {
  mySensor.accelUpdate();
  mySensor.gyroUpdate();
  mySensor.magUpdate();

  Serial.print("ACCEL:");
  Serial.print(mySensor.accelX());
  Serial.print(",");
  Serial.print(mySensor.accelY());
  Serial.print(",");
  Serial.print(mySensor.accelZ());
  Serial.print(";GYRO:");
  Serial.print(mySensor.gyroX());
  Serial.print(",");
  Serial.print(mySensor.gyroY());
  Serial.print(",");
  Serial.print(mySensor.gyroZ());
  Serial.print(";MAG:");
  Serial.print(mySensor.magX());
  Serial.print(",");
  Serial.print(mySensor.magY());
  Serial.print(",");
  Serial.println(mySensor.magZ());
  delay(100);
}
```

Listing 6: Código Nodo Publicador de Datos del Control.

```python
#! /usr/local/bin python3

import rclpy
from rclpy.node import Node
from rclpy.publisher import Publisher
from scipy.ndimage import gaussian_filter1d
from std_msgs.msg import Header
from sensor_msgs.msg import Imu
import time
import matplotlib.pyplot as plt
from pydualsense import pydualsense
```

```
12 import numpy as np
13 import re
14
15 class ControllerBridge(Node):
16     def __init__(self):
17         super().__init__('controller_bridge')
18
19         self.imu = Imu()
20         self.imu_pub = self.create_publisher(Imu, 'imu/data_raw', 15)
21
22         freq = 15  # Hz
23         self.timer = self.create_timer(1 / freq, self.timer_callback)
24
25         # Conexion con el mando
26         self.controller = pydualsense()
27         self.controller.init()
28
29         /# Inicializar variables
30         self.accx = self.accy = self.accz = 0.0
31         self.gx = self.gy = self.gz = 0.0
32
33         # Calibracion
34         self.calibrating = True
35         self.calibration_samples = []
36         self.calibration_duration = 2.0  # segundos
37         self.start_time = self.get_clock().now()
38
39         self.bias_ax = 0.0
40         self.bias_ay = 0.0
41         self.bias_az = 0.0
42         self.bias_gx = 0.0
43         self.bias_gy = 0.0
44         self.bias_gz = 0.0
45
46         # Buffer del giroscopio
47         self.buffer_size = 15
48         self.gyro_data = np.zeros((0, 3))
49
50     def timer_callback(self):
51         now = self.get_clock().now()
52
53         # Lectura del Control
54         self.accx = float(self.controller.state.accelerometer.X)
55         self.accy = float(self.controller.state.accelerometer.Y)
56         self.accz = float(self.controller.state.accelerometer.Z)
57
58         acc = np.array([self.accx, self.accy, self.accz])
59         acc = acc / np.linalg.norm(acc) * 9.81
60         self.accx, self.accy, self.accz = acc
61
62         self.gx = float(np.deg2rad(self.controller.state.gyro.Roll *
               0.1))
63         self.gy = float(np.deg2rad(self.controller.state.gyro.Pitch *
```

```
            0.1))
        self.gz = float(np.deg2rad(self.controller.state.gyro.Yaw *
            0.1))

        # Calibracion (2s)
        if self.calibrating:
            elapsed = (now - self.start_time).nanoseconds / 1e9

            # Guardar muestras
            self.calibration_samples.append([
                self.accx, self.accy, self.accz,
                self.gx, self.gy, self.gz
            ])

            if elapsed < self.calibration_duration:
                self.get_logger().info("Calibrando... NO MUEVAS EL
                    CONTROL")
                return

            # Terminar calibracion
            samples = np.array(self.calibration_samples)
            acc_mean = samples[:, 0:3].mean(axis=0)
            gyro_mean = samples[:, 3:6].mean(axis=0)

            # Bias esperado: acc = [0, 9.81, 0]
            self.bias_ax = acc_mean[0] - 0.0
            self.bias_ay = acc_mean[1] - 9.81
            self.bias_az = acc_mean[2] - 0.0

            # Gyro debe ser 0
            self.bias_gx = gyro_mean[0]
            self.bias_gy = gyro_mean[1]
            self.bias_gz = gyro_mean[2]

            self.calibrating = False
            self.get_logger().info(">>> Calibracion completa <<<")
            self.get_logger().info(f"Bias acc: {self.bias_ax:.4f}, {
                self.bias_ay:.4f}, {self.bias_az:.4f}")
            self.get_logger().info(f"Bias gyro: {self.bias_gx:.6f}, {
                self.bias_gy:.6f}, {self.bias_gz:.6f}")
            return

        # Calibracion
        acc_x = self.accx - self.bias_ax
        acc_y = self.accy - self.bias_ay
        acc_z = self.accz - self.bias_az

        gx_c = self.gx - self.bias_gx
        gy_c = self.gy - self.bias_gy
        gz_c = self.gz - self.bias_gz

        # Suavizado del Giroscopio
        new_row = np.array([[gx_c, gy_c, gz_c]])
```

```
        self.gyro_data = np.vstack([self.gyro_data, new_row])

        if len(self.gyro_data) > self.buffer_size:
            self.gyro_data = self.gyro_data[-self.buffer_size:]

        if len(self.gyro_data) >= 5:
            gx_s = gaussian_filter1d(self.gyro_data[:, 0], sigma=2)[-1]
            gy_s = gaussian_filter1d(self.gyro_data[:, 1], sigma=2)[-1]
            gz_s = gaussian_filter1d(self.gyro_data[:, 2], sigma=2)[-1]
        else:
            gx_s, gy_s, gz_s = gx_c, gy_c, gz_c

        # Publicar

        self.imu.header.stamp = self.get_clock().now().to_msg()
        self.imu.header.frame_id = 'imu'

        self.imu.angular_velocity.x = gx_s
        self.imu.angular_velocity.y = gy_s
        self.imu.angular_velocity.z = gz_s

        self.imu.linear_acceleration.x = acc_x
        self.imu.linear_acceleration.y = acc_y
        self.imu.linear_acceleration.z = acc_z

        self.imu.orientation.w = 1.0

        self.imu_pub.publish(self.imu)

        # Debug
        self.get_logger().info(f"Accelerometer: {acc_x:.3f}, {acc_y:.3f
            }, {acc_z:.3f}")
        self.get_logger().info(f"gyroscope: {gx_s:.3f}, {gy_s:.3f}, {
            gz_s:.3f}")


def main(args = None):
    rclpy.init(args = args)

    controller_bridge = ControllerBridge()

    rclpy.spin(controller_bridge)

    controller_bridge.destroy_node()
    rclpy.shutdown()

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

Listing 7: Código Nodo Suscriptor del Filtro Madgwick Cinematica Inversa.

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
```

```python
import numpy as np
import math
from sensor_msgs.msg import Imu
from std_msgs.msg import Int32MultiArray
import time
import serial


def angles_to_steps(q, steps_per_rev, microsteps, gear_ratio):
    steps_per_rad = (steps_per_rev * microsteps * gear_ratio) / (2 * math.pi)
    return [int(angle * steps_per_rad) for angle in q]


class MadgwickIKNode(Node):
    def __init__(self):

        super().__init__('madgwick_inverse_kinematics')

        self.sub = self.create_subscription(
            Imu, '/imu/data', self.cb, 10)

        self.pub = self.create_publisher(
            Int32MultiArray, '/spm/steps', 10)

        # -------- SERIAL --------
        self.serial_port = "/dev/ttyACM8"
        try:
            self.serial = serial.Serial(self.serial_port, 9600, timeout=0.1)
            time.sleep(2)
            self.get_logger().info("Arduino conectado.")
        except Exception as e:
            self.get_logger().error(f"Error serial: {e}")
            self.serial = None

        self.waiting_ack = False      # esperando respuesta del Arduino
        self.last_command = None      # ultimo comando enviado
        self.last_send_time = time.time()

        # Parametros
        self.alpha_joints = 0.10
        self.joint_filtered = np.zeros(3)
        self.GAIN = 0.5

        self.steps_per_rev = 200
        self.microsteps = 16
        self.gear_ratio = 1.0
        self.home_offset = np.array([700, 700, 700])

        # Calibracion IMU
        self.calibrating = True
        self.calibration_start_time = time.time()
```

```
        self.calibration_duration = 2.0
        self.calibration_samples = []
        self.offset = np.zeros(3)

    # ============================================================
    #             Lee si Arduino respondio "OK"
    # ============================================================
    def read_arduino(self):
        if not self.serial:
            return False

        try:
            line = self.serial.readline().decode().strip()
            if line != "":
                self.get_logger().info(f"[ARDUINO] {line}")
            if line == "OK" or line == "READY":
                return True
        except:
            pass

        return False

    # ============================================================
    #                     Callback IMU
    # ============================================================
    def cb(self, msg: Imu):

        # Si estamos esperando OK del Arduino  no mandar nada
        if self.waiting_ack:
            if self.read_arduino():
                self.waiting_ack = False
            else:
                return

        # POSICION IMU
        qw, qx, qy, qz = msg.orientation.w, msg.orientation.x, msg.
           orientation.y, msg.orientation.z

        roll = math.atan2(2*(qw*qx + qy*qz), 1 - 2*(qx*qx + qy*qy))
        pitch = math.asin(2*(qw*qy - qz*qx))
        yaw = math.atan2(2*(qw*qz + qx*qy), 1 - 2*(qy*qy + qz*qz))

        q_arr = np.array([roll, pitch, yaw])

        # -------- Calibracion --------
        if self.calibrating:
            self.calibration_samples.append(q_arr)
            if time.time() - self.calibration_start_time >= self.
               calibration_duration:
                self.offset = np.mean(self.calibration_samples, axis=0)
                self.calibrating = False
                self.get_logger().info("Calibracion completada.")
            else:
```

```
                return

        q_arr = q_arr - self.offset

        # -------- Unwrap + filtro --------
        diff = q_arr - self.joint_filtered
        diff = (diff + math.pi) % (2 * math.pi) - math.pi
        q_arr = self.joint_filtered + diff

        q_smooth = self.alpha_joints * q_arr + (1 - self.alpha_joints)
           * self.joint_filtered
        self.joint_filtered = q_smooth

        q_gain = self.joint_filtered * self.GAIN

        # -------- IK pasos relativos --------
        rel_steps = angles_to_steps(q_gain, self.steps_per_rev, self.
           microsteps, self.gear_ratio)

        abs_steps = (self.home_offset + np.array(rel_steps)).astype(int
           )

        if not hasattr(self, "last_steps"):
            self.last_steps = abs_steps.copy()

        if np.all(np.abs(abs_steps - self.last_steps) < 3):
            return

        self.last_steps = abs_steps.copy()


        if self.serial:
            serial_msg = f"{abs_steps[0]},{abs_steps[1]},{abs_steps
               [2]}\n"
            self.serial.write(serial_msg.encode())
            self.waiting_ack = True          # ahora esperamos
               confirmacion

        # Publicar en ROS
        ros_msg = Int32MultiArray()
        ros_msg.data = abs_steps
        self.pub.publish(ros_msg)

        self.get_logger().info(f"Steps to Arduino: {abs_steps}")


def main(args=None):
    rclpy.init(args=args)
    node = MadgwickIKNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

```

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

Listing 8: Enviando Señales Seriales desde Ros2

```
// ======================================================
// Pines CNC Shield + A4988
// ======================================================

#define DIR1 8
#define STEP1 9

#define DIR2 3
#define STEP2 2

#define DIR3 4
#define STEP3 5

#define ENABLE 7   // Activo en LOW

// HOME ABSOLUTO
long pos1 = 0, pos2 = 0, pos3 = 0;
const long HOME1 = 700;
const long HOME2 = 700;
const long HOME3 = 700;

#define STEP_DELAY 700

// ======================================================
// Setup
// ======================================================
void setup() {
  Serial.begin(9600);
  delay(1000);

  pinMode(DIR1, OUTPUT); pinMode(STEP1, OUTPUT);
  pinMode(DIR2, OUTPUT); pinMode(STEP2, OUTPUT);
  pinMode(DIR3, OUTPUT); pinMode(STEP3, OUTPUT);
  pinMode(ENABLE, OUTPUT);
  digitalWrite(ENABLE, LOW);   // Activar drivers

  Serial.println("Sistema iniciando...");
  Serial.println("Moviendo al HOME [700,700,700]");

  // MOVER A HOME COMO ABSOLUTO
  moverSimultaneo(HOME1, HOME2, HOME3, STEP_DELAY);

  pos1 = HOME1;
  pos2 = HOME2;
  pos3 = HOME3;

  Serial.println("HOME alcanzado.");
  Serial.println("Esperando posiciones ABSOLUTAS...");
}
```

```
50
51 void moverSimultaneo(long t1, long t2, long t3, long delstep) {
52
53   long d1 = t1 - pos1;
54   long d2 = t2 - pos2;
55   long d3 = t3 - pos3;
56
57   long a1 = abs(d1);
58   long a2 = abs(d2);
59   long a3 = abs(d3);
60
61   long maxSteps = max(a1, max(a2, a3));
62   if (maxSteps == 0) return;
63
64   // Direcciones
65   digitalWrite(DIR1, (d1 > 0) ? HIGH : LOW);
66   digitalWrite(DIR2, (d2 > 0) ? HIGH : LOW);
67   digitalWrite(DIR3, (d3 > 0) ? HIGH : LOW);
68
69   for (long i = 0; i < maxSteps; i++) {
70
71     if (i * a1 / maxSteps < a1) digitalWrite(STEP1, HIGH);
72     if (i * a2 / maxSteps < a2) digitalWrite(STEP2, HIGH);
73     if (i * a3 / maxSteps < a3) digitalWrite(STEP3, HIGH);
74
75     delayMicroseconds(delstep);
76
77     digitalWrite(STEP1, LOW);
78     digitalWrite(STEP2, LOW);
79     digitalWrite(STEP3, LOW);
80
81     delayMicroseconds(delstep);
82   }
83
84   // Actualizar a la nueva posicion absoluta
85   pos1 = t1;
86   pos2 = t2;
87   pos3 = t3;
88 }
89
90 // ======================================================
91 // Loop recibir POSICIONES ABSOLUTAS desde ROS
92 // Formato:  x,y,z
93 // ======================================================
94 void loop() {
95
96   if (Serial.available()) {
97     String data = Serial.readStringUntil('\n');
98
99     // Si la linea esta incompleta o muy larga, descartar
100    if (data.length() < 3 || data.length() > 20) {
101        Serial.println("ERR linea_corrupta");
102        return;
```

```
    }

    data.trim();


    for (char c : data) {
        if (!isdigit(c) && c != ',' && c != '-' ) {
            Serial.println("ERR caracteres_invalidos");
            return;
        }
    }

    // Buscar comas
    int c1 = data.indexOf(',');
    int c2 = data.lastIndexOf(',');

    if (c1 < 0 || c2 < 0 || c1 >= c2) {
      Serial.println("ERR formato");
      return;
    }

    long t1 = data.substring(0, c1).toInt();
    long t2 = data.substring(c1 + 1, c2).toInt();
    long t3 = data.substring(c2 + 1).toInt();

    // ---- Validar rangos ----
    if (t1 < 0 || t1 > 1400 ||
        t2 < 0 || t2 > 1400 ||
        t3 < 0 || t3 > 1400) {
        Serial.println("ERR fuera_de_rango");
        return;
    }

    Serial.print("ABS recibidos: ");
    Serial.print(t1); Serial.print(", ");
    Serial.print(t2); Serial.println(t3);

    moverSimultaneo(t1, t2, t3, 350);

    Serial.println("OK");
  }
}
```

Anexo B. Planos de piezas

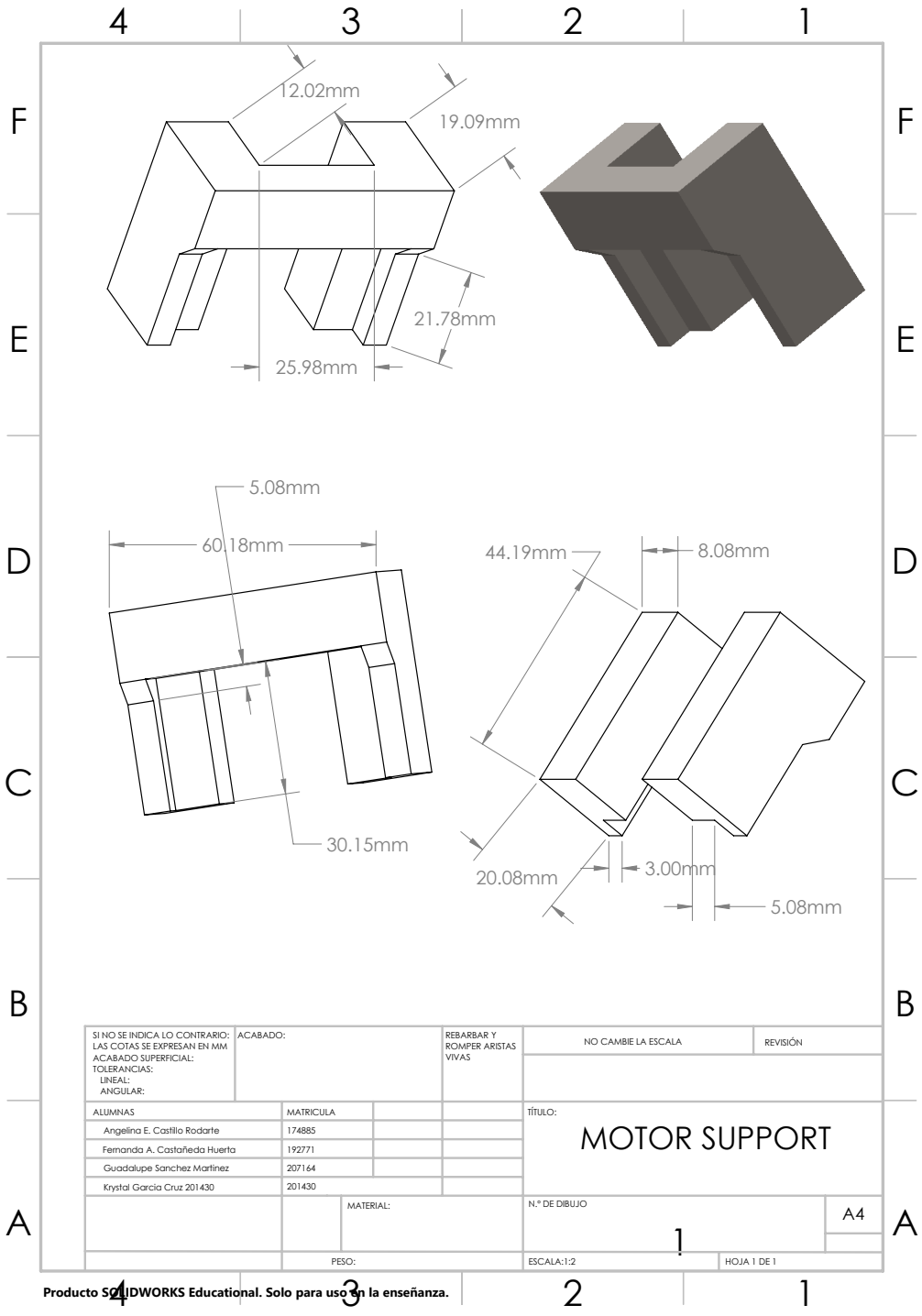


Figura 2: Planos del Motor Support.

Anexo C. Esquemas electrónicos

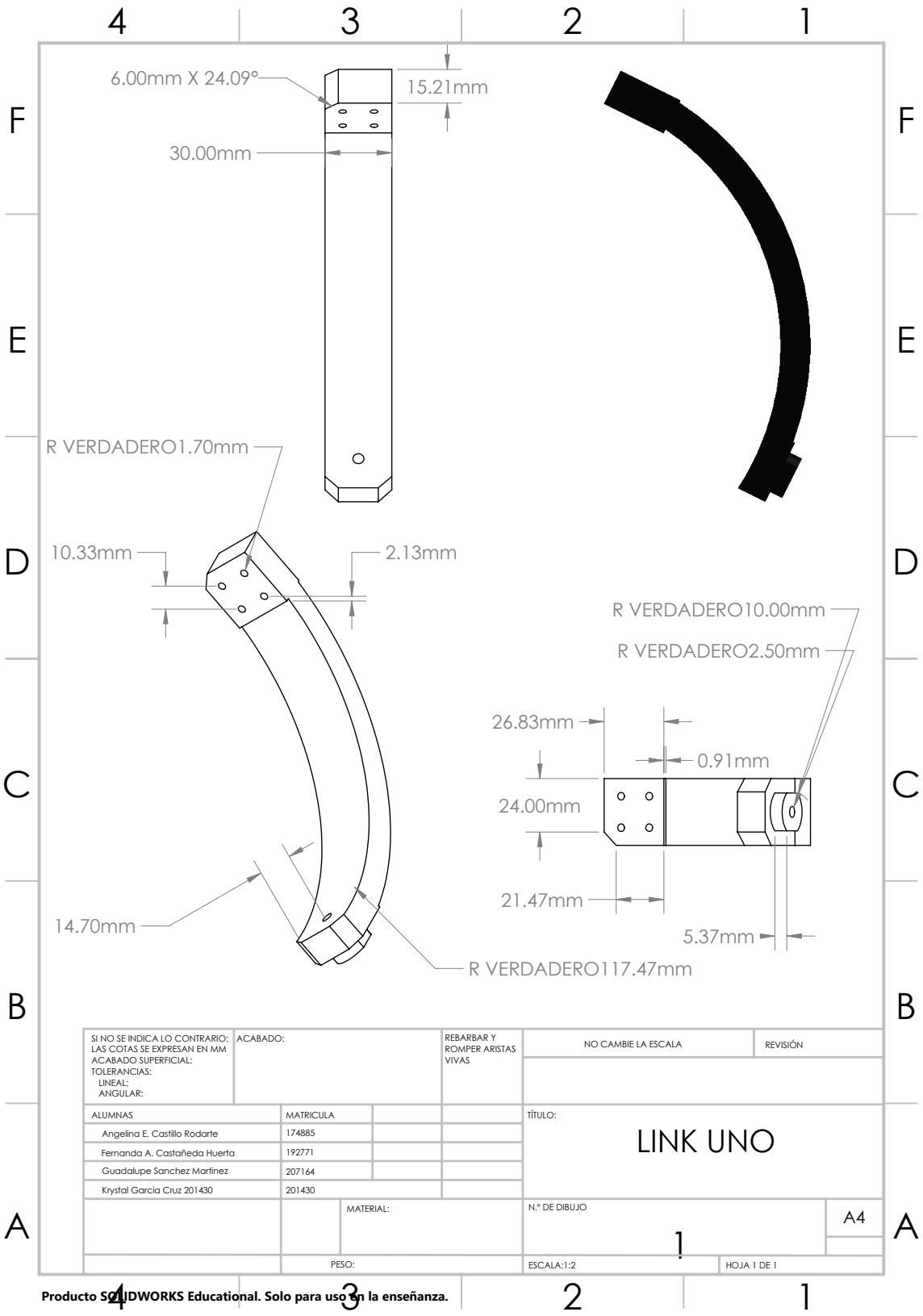


Figura 3: **Link 1**

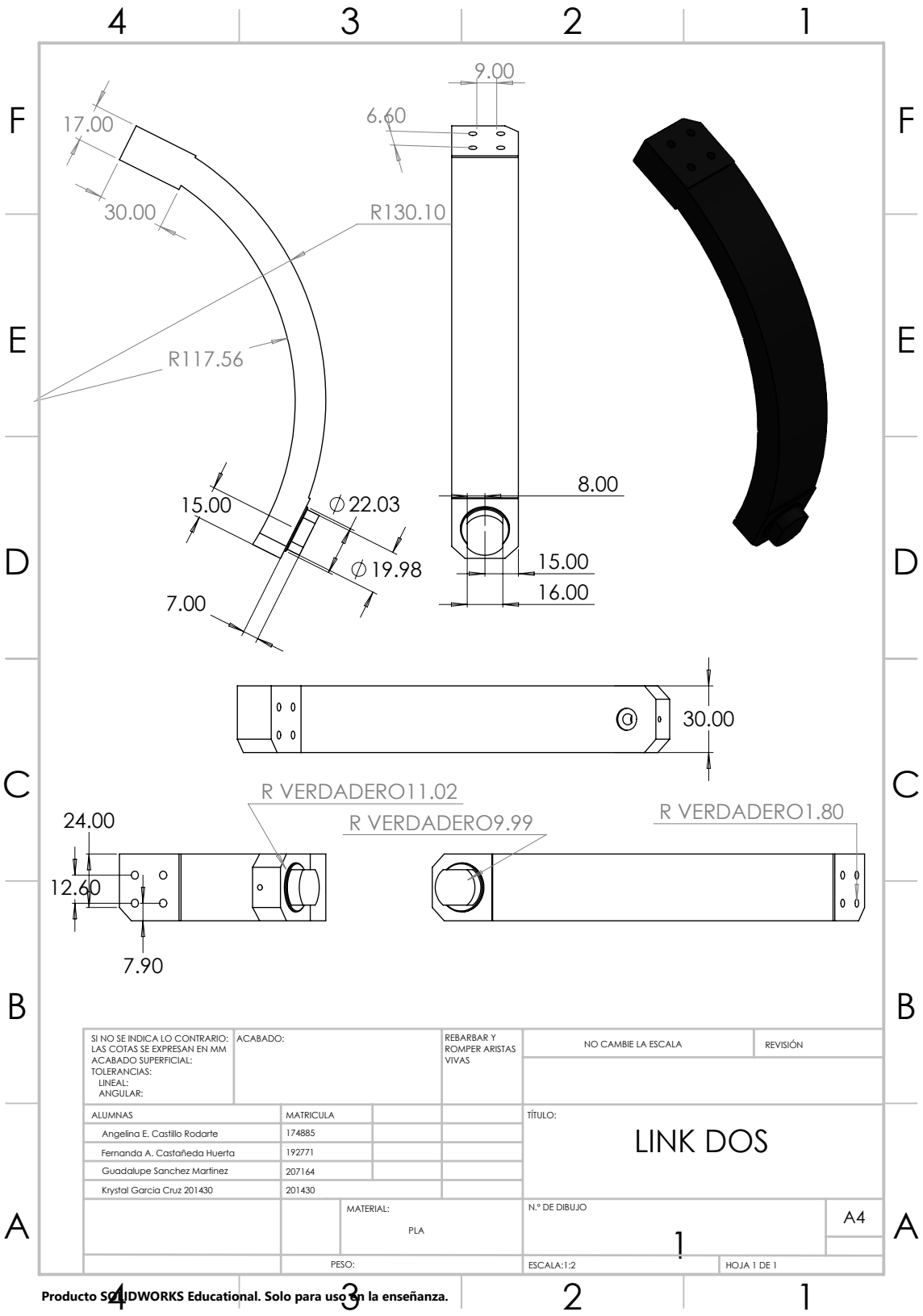


Figura 4: **Link 2**

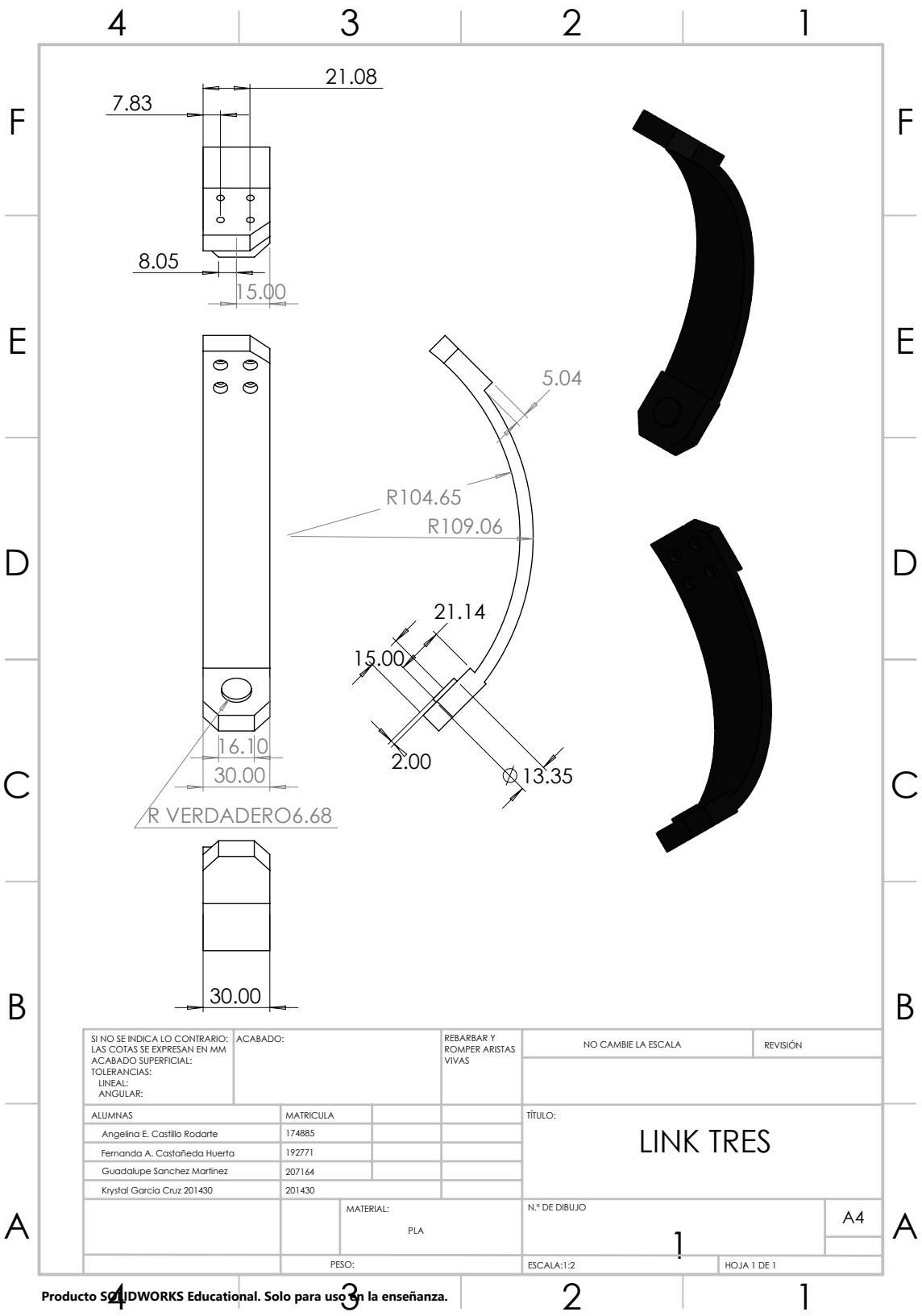


Figura 5: **Link 3**

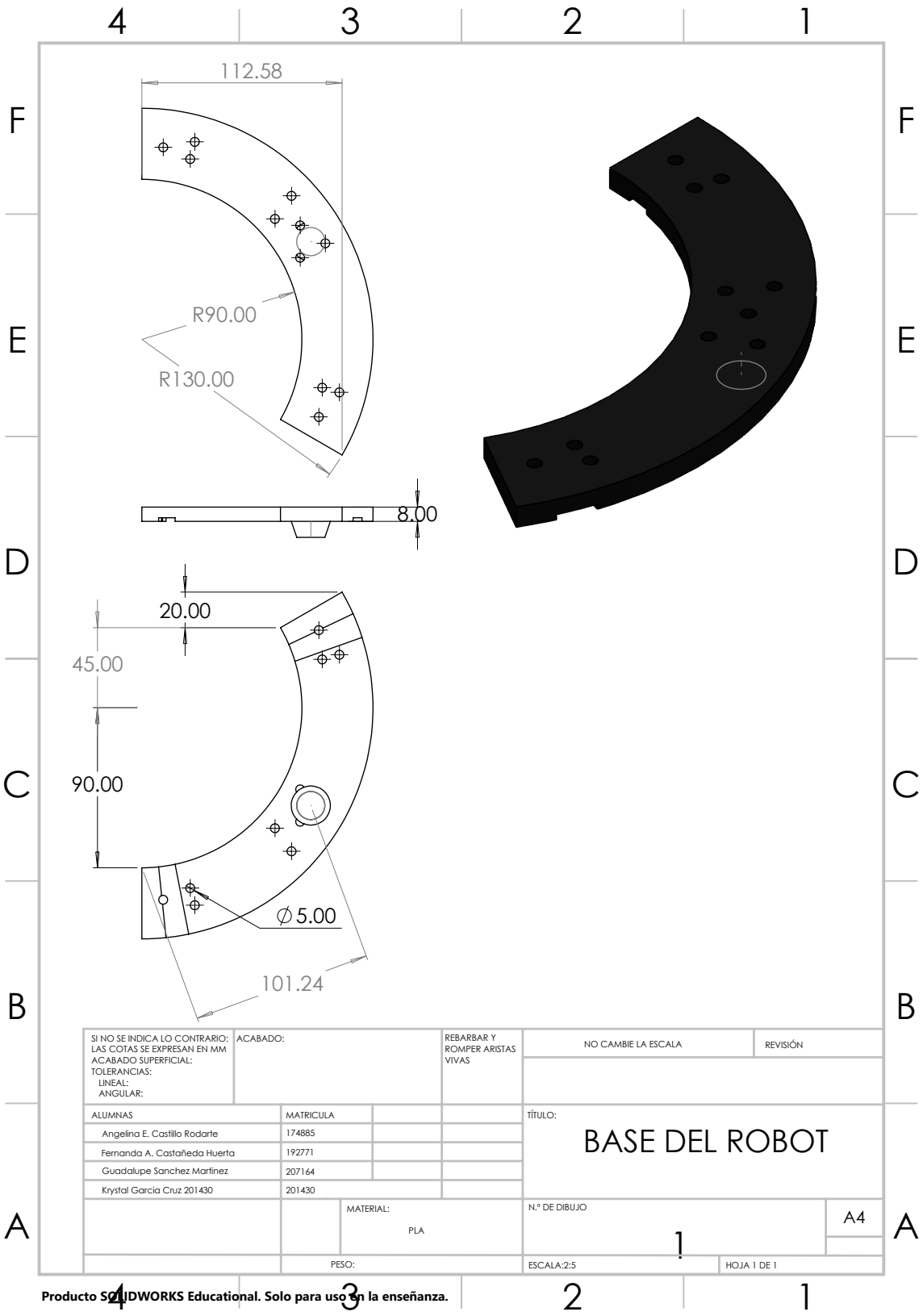


Figura 6: **Parte de base para ensamblar del robot**

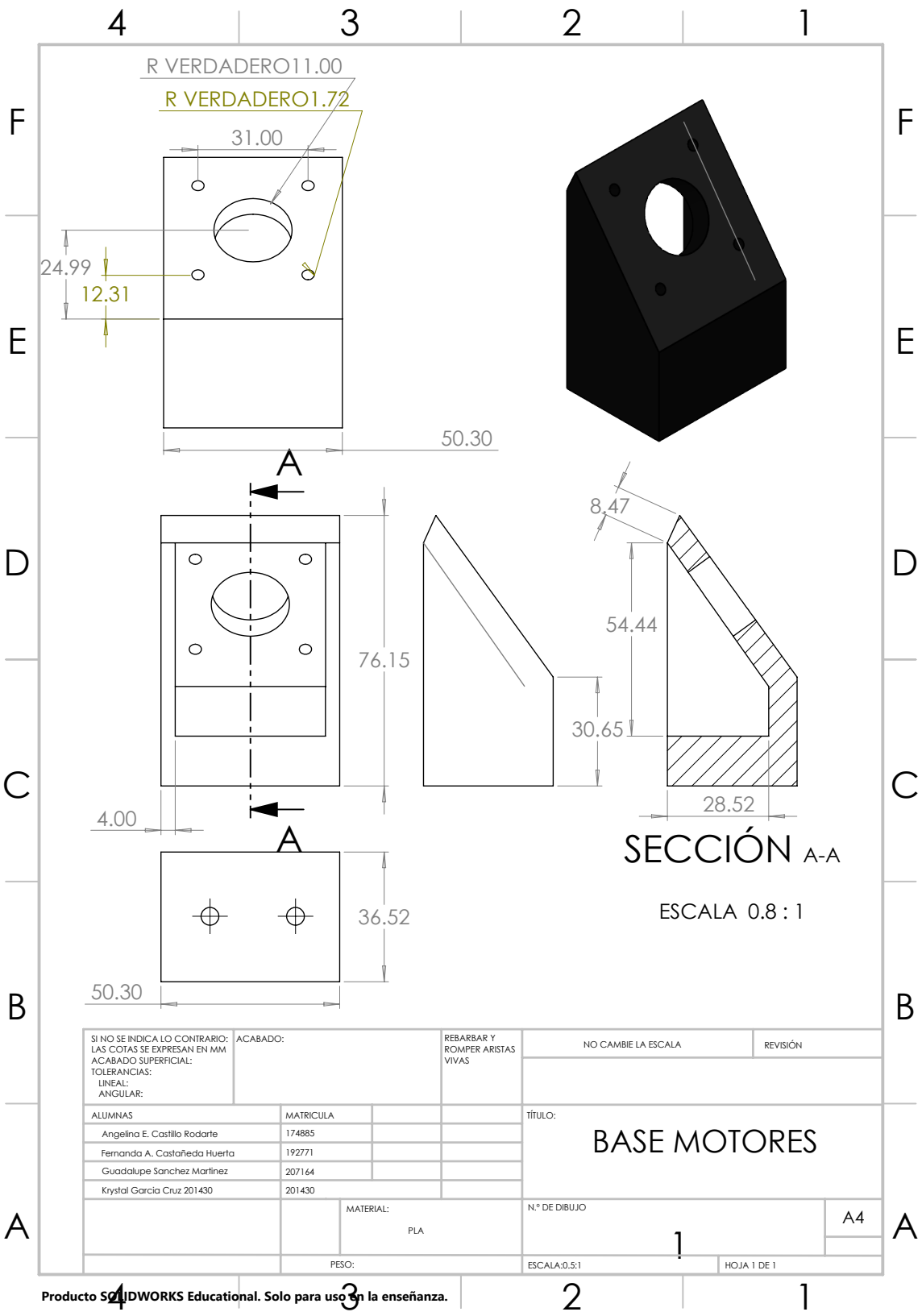


Figura 7: **Base de motores nema 17**

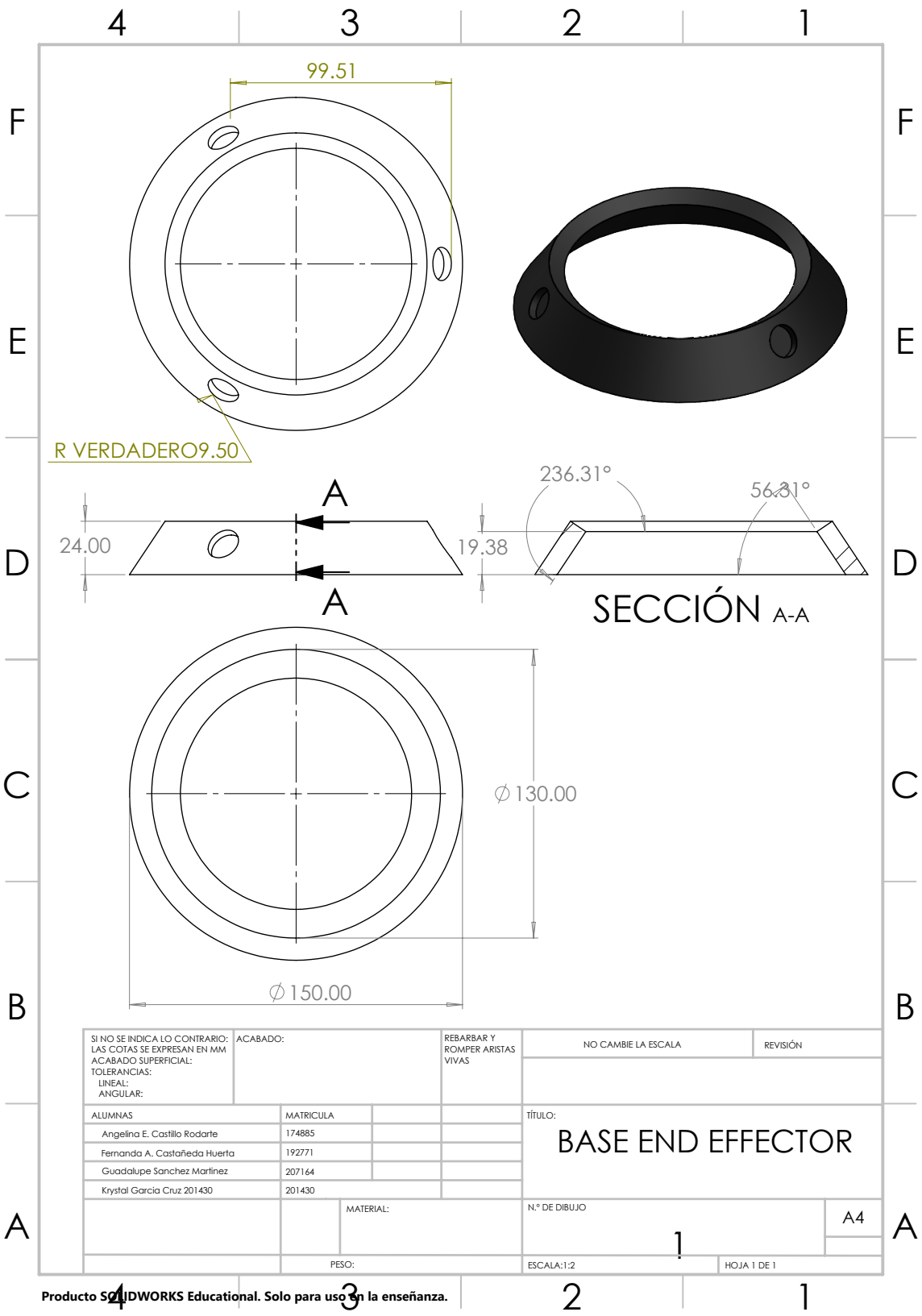


Figura 8: **Aro de end- effector del robot**

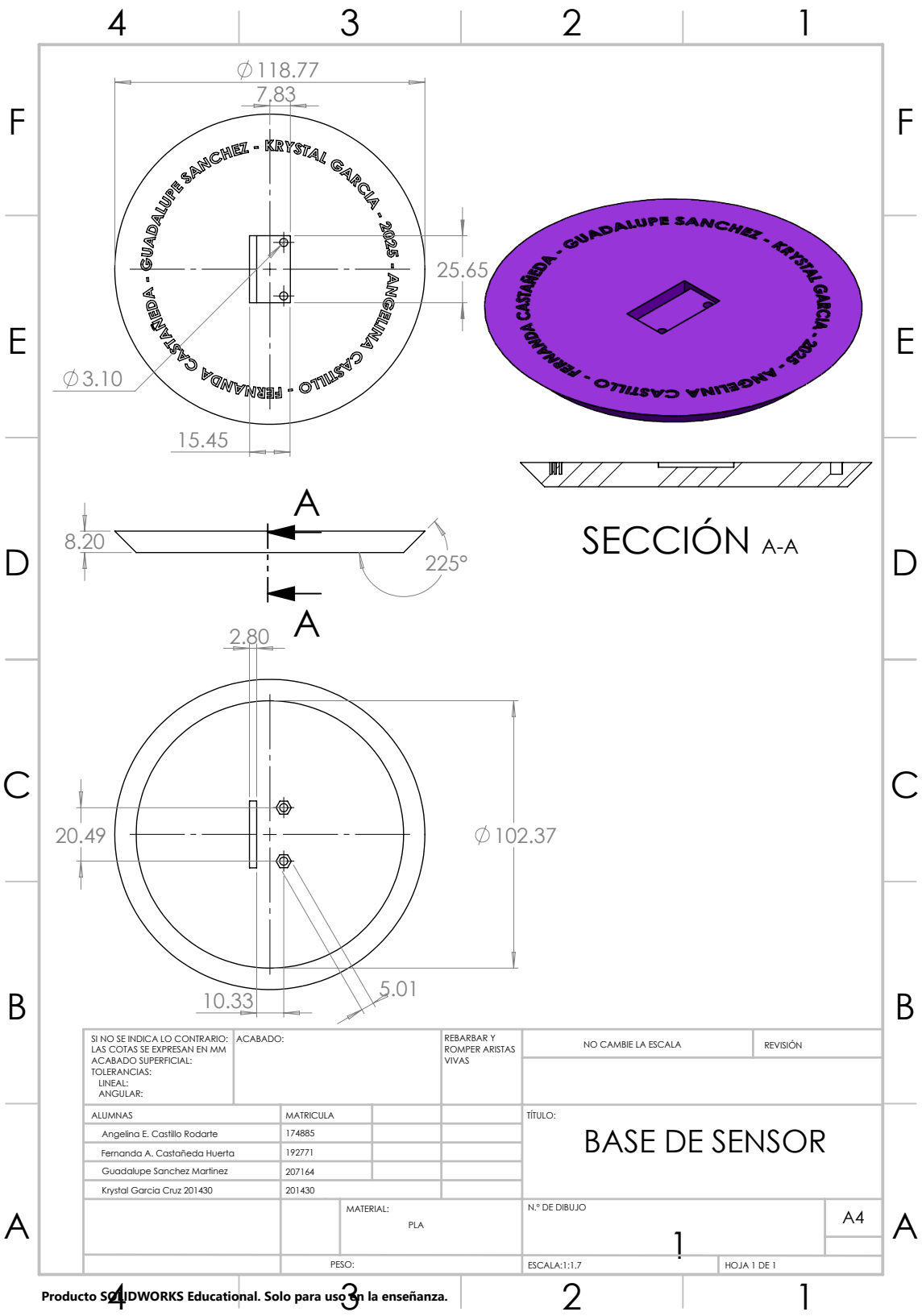


Figura 9: **Base del sensor**

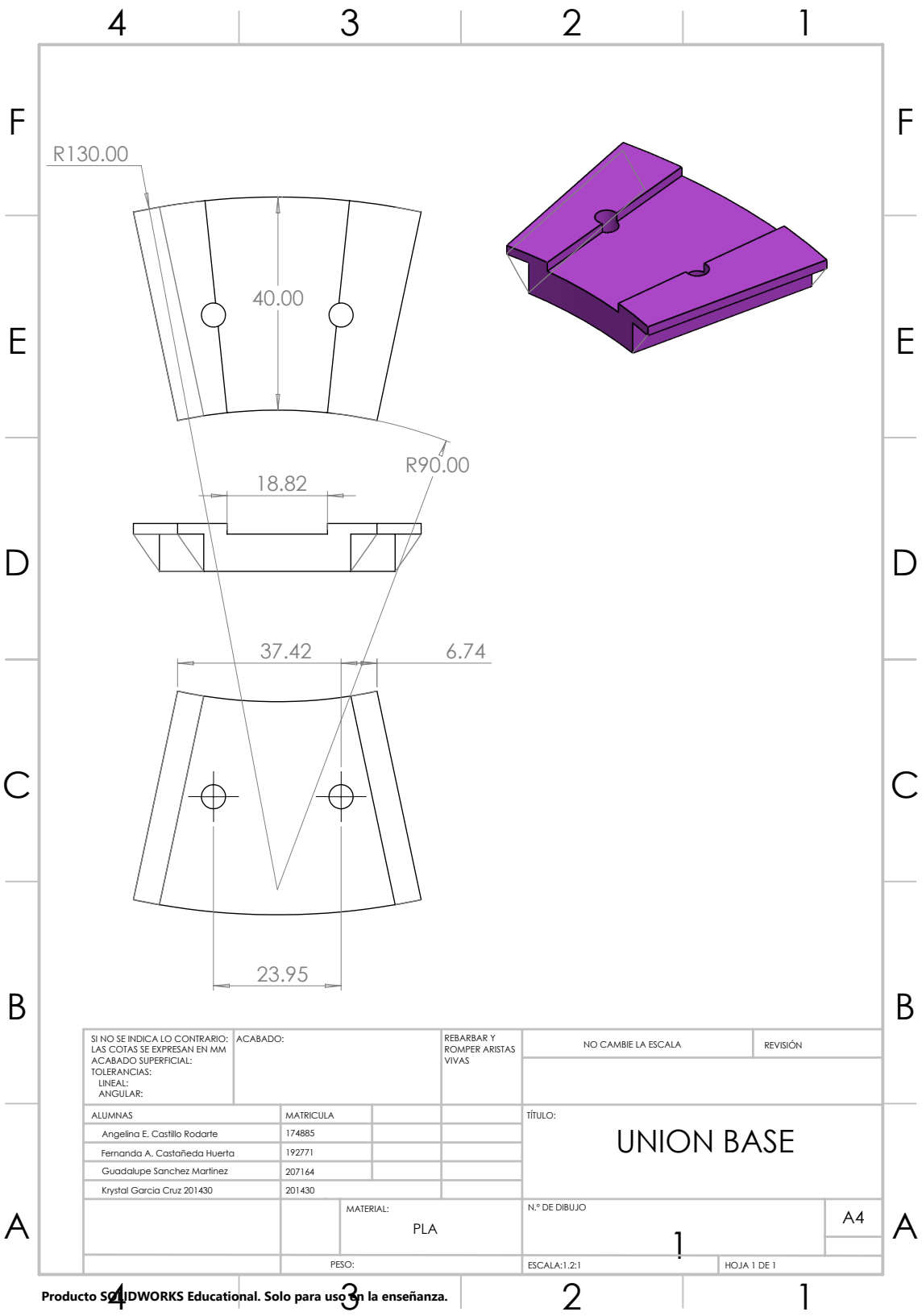


Figura 10: **Pieza unión de base**

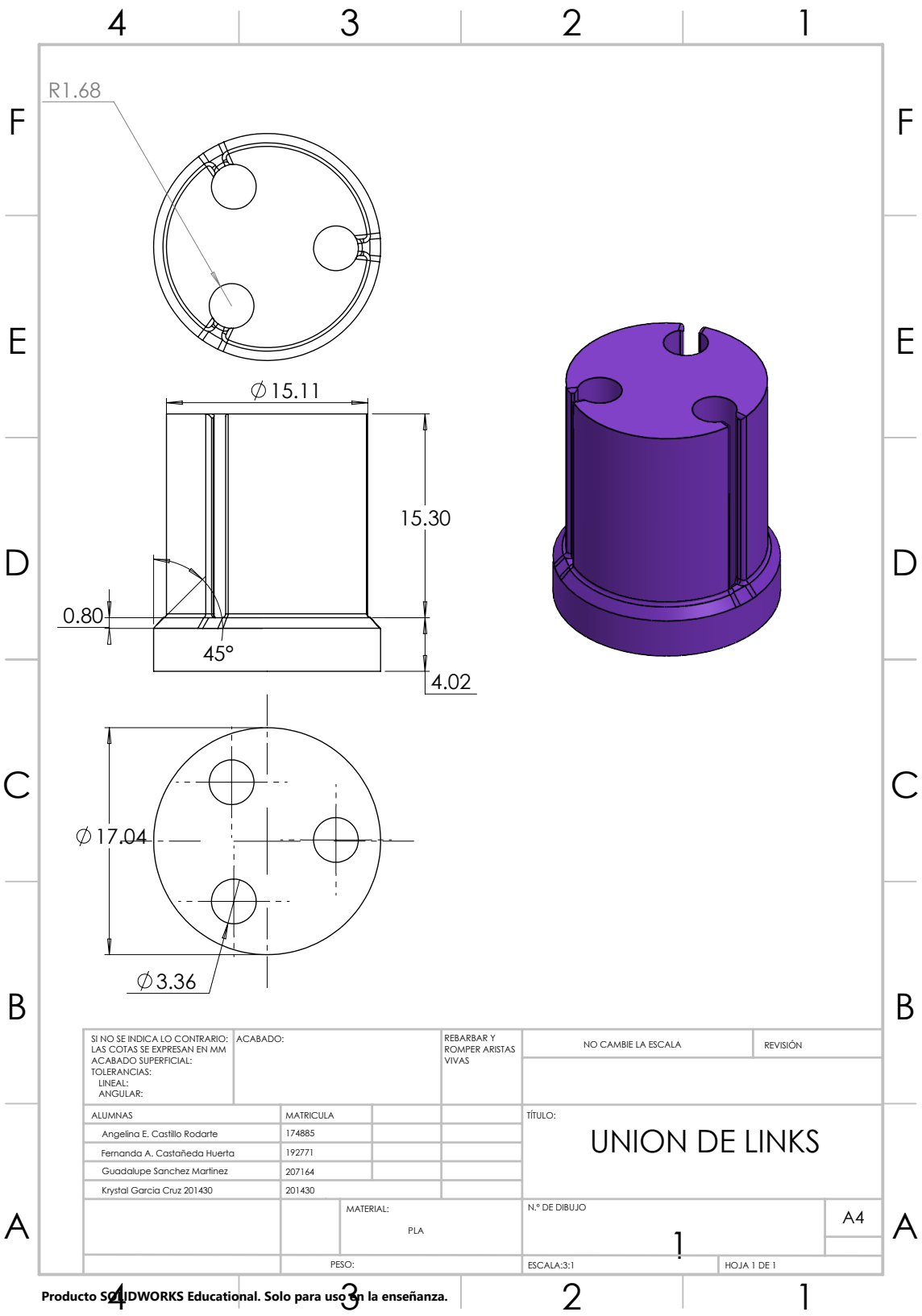


Figura 11: **Pieza de unión de linos**

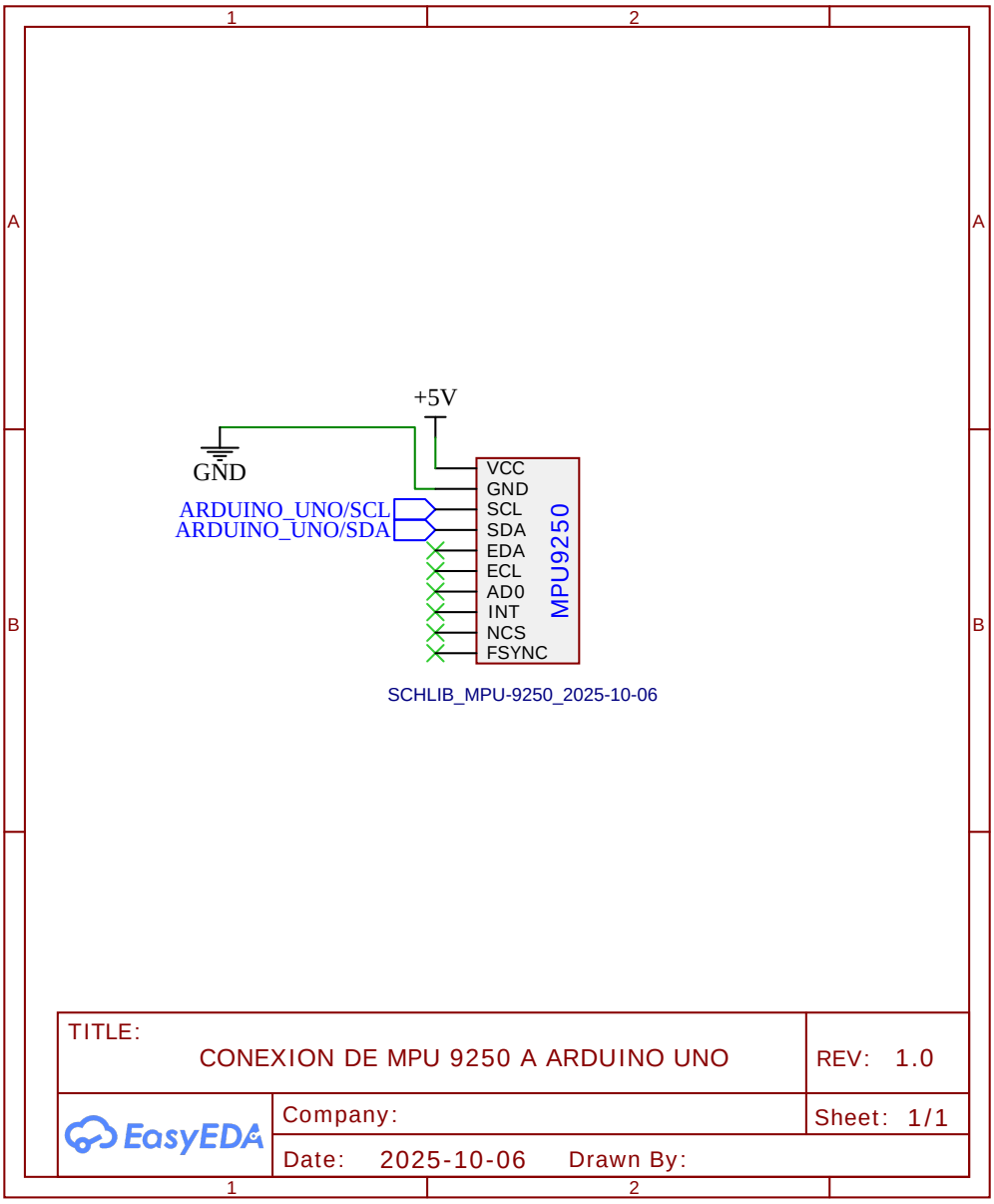
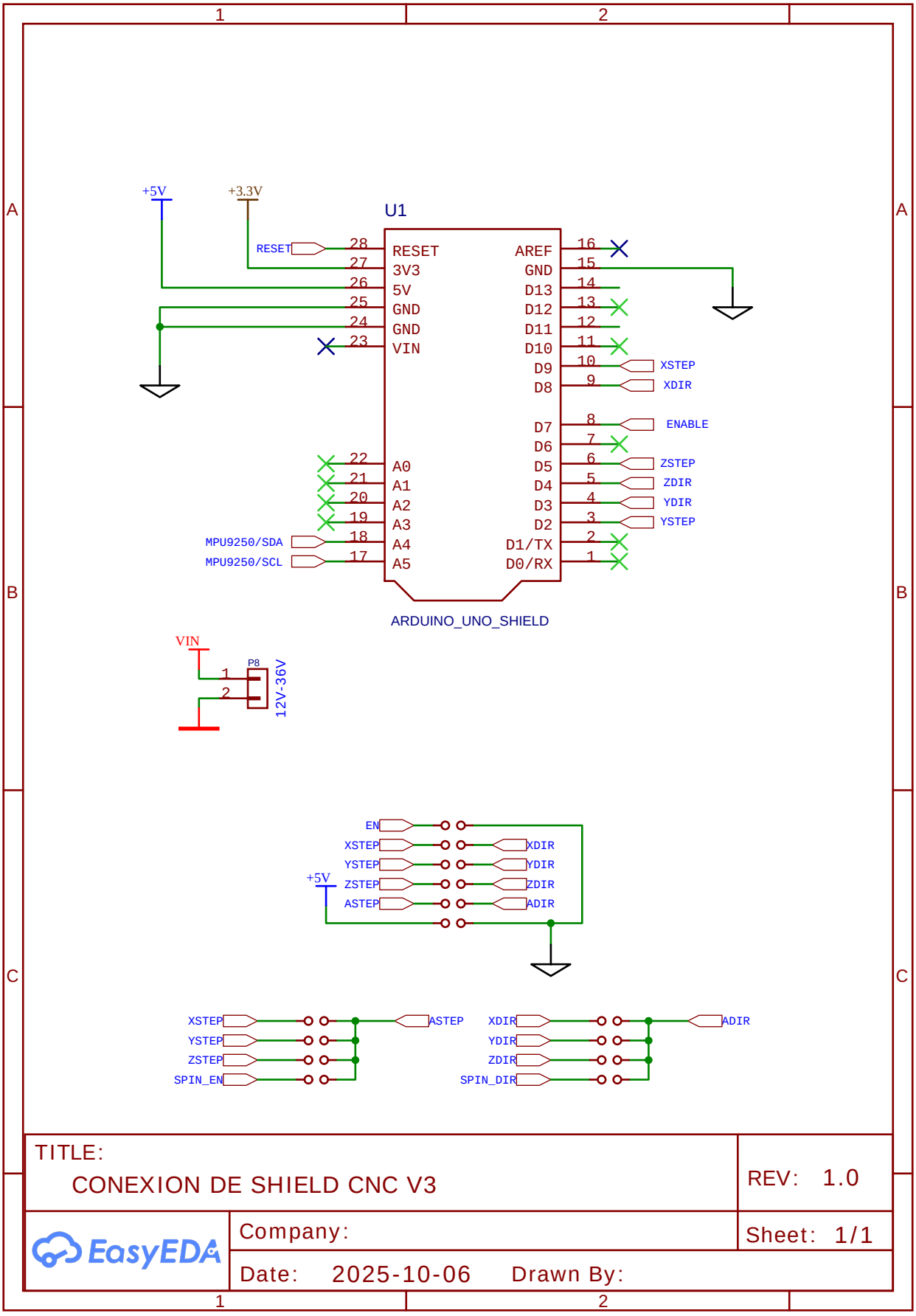


Figura 12: **Conexión general del robot**


1
2
A
B
C
+5V
+3.3V
U1
RESET
28
27
26
25
24
23
RESET
3V3
5V
GND
GND
VIN
AREF
GND
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1/TX
D0/RX
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
XSTEP
XDIR
ENABLE
ZSTEP
ZDIR
YDIR
YSTEP
22
21
20
19
18
17
A0
A1
A2
A3
A4
A5
MPU9250/SDA
MPU9250/SCL
ARDUINO_UNO_SHIELD
VIN
P8
1
2
12V-36V
EN
XSTEP
YSTEP
ZSTEP
ASTEP
XDIR
YDIR
ZDIR
ADIR
+5V
XSTEP
YSTEP
ZSTEP
SPIN_EN
ASTEP
XDIR
YDIR
ZDIR
SPIN_DIR
ADIR
TITLE:
CONEXION DE SHIELD CNC V3
REV: 1.0
EasyEDA
Company:
Sheet: 1/1
Date: 2025-10-06
Drawn By:

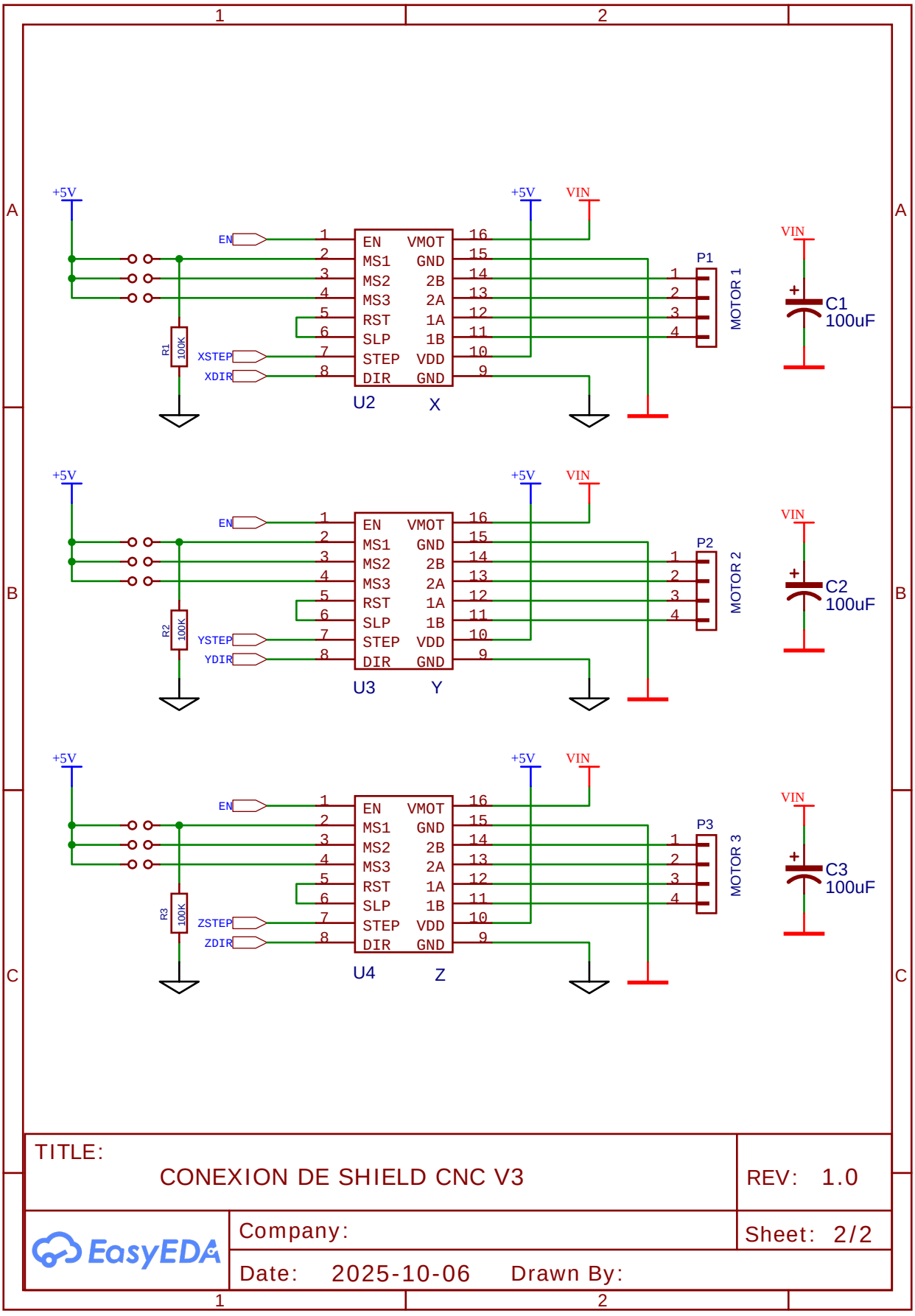


Figura 14: **Esquemático de control para tres motores paso a paso (ejes X, Y, Z).**