Universidad Autónoma de Ciudad Juárez.

Extensión multidisciplinaria ciudad universitaria

Ingeniería en Mecatrónica

Curso: Diseño mecatrónico

Diseño de un robot cartesiano ciberfísico con integración hardware in the loop

Autores:

Adan Rodrigo De la Cruz Osorio 192637

Daniela Yazmin Tarango Montes 201153

Jose Orlando Martinez Gonzalez 201471

Bernardo David Burciaga Medina 201489

Docente:

Dr. Francesco José García Luna

Fecha de entrega: 12 de mayo del 2025
Ciudad Juárez, Chih.

Índice

1. Introducción **1**

2. Objetivos **1**

2.1. Objetivo General . . . 1

2.2. Objetivos Específicos . . . 1

3. Marco teórico **2**

3.1. Robots cartesianos . . . 2

3.2. HIL . . . 2

3.3. ROS2 . . . 3

3.4. Gazebo . . . 4

3.5. URDF y SDF . . . 4

3.6. EDA . . . 5

3.7. Interfaz gráfica . . . 5

4. Metodología **6**

4.1. Concepto y planificación . . . 6

4.1.1. Descripción del robot . . . 6

4.1.2. Especificaciones . . . 7

4.1.3. Requerimientos . . . 9

4.1.4. Limitaciones . . . 9

4.1.5. Diagrama de actividades . . . 10

4.2. Diseño del robot . . . 11

4.2.1. Diseño electrónico . . . 11

4.2.2. Diseño 3D . . . 12

4.3. Simulación del robot . . . 14

4.3.1. Instalación de ROS2 y Gazebo . . . 14

4.3.2. Creación del entorno de trabajo . . . 14

4.3.3. Visualización del robot . . . 16

4.3.4. Simulación del movimiento . . . 16

4.4. Construcción del robot . . . 16

4.5. Interfaz gráfica . . . 18

4.6. Pruebas . . . 18

4.6.1. Pruebas de comunicación 19
4.6.2. Pruebas de movimiento 19
4.6.3. Observaciones generales 20

5. Resultados **20**
5.1. Resultados del diseño 3D 20
5.2. Resultados de la simulación 21
5.3. Resultados de la construcción 22
5.4. Resultados de la interfaz gráfica 23
5.5. Resultados de las pruebas 23

6. Conclusiones **24**

7. Referencias **25**

A. Anexo 1: Diagrama de actividades **27**

B. Anexo 2: Esquema eléctrico **28**

C. Anexo 3: Planos de piezas **29**

D. Anexo 4: Red de tópicos **53**

E. Anexo 5: Script de instalación de ROS2, Gazebo y dependencias **54**

F. Anexo 6: Códigos del paquete carobot **58**

1. Introducción

En este proyecto se observara el diseño de un robot cartesiano ciberfísico, que permite la ejecución de movimientos de las tres dimensiones. Para la realización de pruebas sin contar con la construcción física completa del robot, fue implementada la técnica de Hardware in the Loop (HIL), mediante la cual el comportamiento del sistema es simulado en un entorno virtual mientras se establece comunicación con el hardware real.

A lo largo del trabajo fueron mostrados los pasos que fueron elaborados, desde el diseño mecánico hasta la programación y simulación del robot. Una valiosa oportunidad para aprender y aplicar conocimientos de diseño mecatrónico fue proporcionada gracias a la realización de este proyecto. El diseño de las piezas del robot fue realizado en SolidWorks, mientras que el entorno de simulación fue configurado utilizando ROS2 y Gazebo. El código, los modelos y los recursos generados fueron puestos a disposición en un repositorio de GitHub y en una carpeta compartida en la nube, con el fin de facilitar su consulta por otros interesados en el tema.

- `https://github.com/adnksharp/cartesian-ppp-ros`
- `https://alumnosuacj.sharepoint.com/:f:/s/Section_445919-equipo3/EisrVS2E7jBPuNGVBnNSQIMBEK5nDnH8iA342m-mQfGQ1A?e=gtPAZ7`
- `https://alumnosuacj.sharepoint.com/:f:/s/Section_445919-equipo3/Ei5hTYRpv0NGlv9GVdSte4EB6db__T7Ev7hyr4ijszJugA?e=J9gBm8`

2. Objetivos

2.1. Objetivo General

- Diseñar un robot cartesiano ciberfísico con integración hardware in the loop.

2.2. Objetivos Específicos

- Diseñar el modelo de un robot cartesiano.
- Simular el robot cartesiano en un entorno virtual.
- Ensamblar el robot cartesiano utilizando componentes mecánicos y electrónicos.
- Programar el nodo de comunicación entre el robot y el entorno simulado.

3. Marco teórico

3.1. Robots cartesianos

Los robots cartesianos, también conocidos como robots de coordenadas rectangulares, se caracterizan por moverse en tres ejes lineales mutuamente perpendiculares: X, Y y Z. Este tipo de robots utiliza actuadores lineales para moverse a lo largo de cada eje, lo que les proporciona una gran precisión y simplicidad mecánica. Son comúnmente utilizados en aplicaciones de manufactura automatizada como impresión 3D, corte CNC, ensamble y manipulación de objetos. Su estructura modular y su facilidad de programación los convierten en una excelente opción para fines educativos y de investigación.

Los robots cartesianos como el de Osmanović, Dedić, Čosić, Trakić, y Bečirović están compuestos por tres actuadores lineales electromecánicos, un sistema de control y un software de soporte. El autor de este robot cartesiano utiliza el mismo método de conversión de movimiento rotacional en translacional a través de husillos roscados. El autor también menciona que el sistema de control incluye un lazo proporcional-integral (PI) para el control de velocidad y un lazo proporcional (P) para el control de posición [1].

Proyectos como el de Dena y Delgado combinan un robot cartesiano con un brazo robótico, lo que permite tener un sistema de cinco grados de libertad. El robot diseñado por el autor utiliza la IDE de Arduino para realizar tareas repetitivas [2].

3.2. HIL

La simulación de HIL(por sus siglas en inglés, Hardware-in-the-loop) es un tipo de simulación en tiempo real que se utiliza para probar el diseño de un controlador. Esta simulación muestra cómo responde el controlador en tiempo real ante un estímulo virtual realista. También se puede utilizar HIL para determinar si el modelo de un sistema físico (planta) es válido [3]. En la simulación de HIL, se utiliza un equipo en tiempo real como una representación virtual del modelo de planta y una versión real del controlador.

La simulación de HIL (Hardware-in-the-loop) es un tipo de simulación en que se utiliza para probar el diseño de un controlador. Esta simulación muestra cómo responde el controlador ante un estímulo virtual realista. También se puede utilizar HIL para determinar si el modelo de un sistema físico (planta) es válido [3]. El uso de HIL permite realizar pruebas en un entorno controlado y seguro, lo que reduce el riesgo de daños al hardware real.

3.3. 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 [4].

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 [5].

Los proyectos de ROS2 se organizan en paquetes, que 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 [6].

Existe una gran cantidad de paquetes disponibles para ROS2, que abarcan una amplia variedad de aplicaciones y funcionalidades. Uno de estos paquetes es `ros_gz_sim`, que permite la integración de ROS2 con el simulador Gazebo. Este paquete proporciona lanzadores y scripts para facilitar la comunicación entre ROS2 y Gazebo, lo que permite simular robots y entornos de manera efectiva [7].

El paquete `ros_gz_bridge` es una herramienta que permite la comunicación bidireccional entre ROS2 y Gazebo. Facilita la transferencia de datos entre ambos sistemas. Este paquete se encarga de crear puentes entre mensajes de ROS2 y Gazebo. Entre los mensajes que se pueden transferir se encuentran los de tipo `JointState` y `Float64`, que son utilizados para representar el estado de las articulaciones y valores de punto flotante de 64 bits, respectivamente. Estos mensajes son interpretados por Gazebo como tipos de tipo `Model` y `Double` respectivamente [8].

3.4. Gazebo

Gazebo es una colección de librerías de software de código abierto diseñadas para simplificar el desarrollo de aplicaciones de alto rendimiento. Su audiencia principal son los desarrolladores, diseñadores y educadores en robótica. Sin embargo, Gazebo ha sido estructurado para adaptarse a diferentes casos de uso. Cada librería dentro de Gazebo tiene dependencias mínimas, lo que permite su uso en tareas que van desde la resolución de transformaciones matemáticas hasta la simulación y gestión de procesos [9].

Gazebo cuenta con una gran variedad de complementos y herramientas que permiten simular diferentes aspectos de un robot. Por ejemplo, el complemento `gz-sim-joint-position-controller-system` permite controlar la posición de las articulaciones de un robot en Gazebo usando un controlador PID para alcanzar la posición deseada. Este complemento es útil para simular el comportamiento de un robot en un entorno virtual antes de implementarlo en el mundo real [10].

Para publicar el estado de las articulaciones de un robot en Gazebo, se puede utilizar el complemento `gz-sim-joint-state-publisher-system`. Este complemento permite enviar información sobre la posición, velocidad y esfuerzo de las articulaciones del robot a tópicos de Gazebo. Esto es útil para monitorear el estado del robot y realizar ajustes [11].

Estos complementos son solo algunos ejemplos de las muchas herramientas y funcionalidades que Gazebo ofrece para la simulación de robots. La flexibilidad y extensibilidad de Gazebo lo convierten en una opción popular para investigadores y desarrolladores en el campo de la robótica. Sumado a esto, el uso de ROS2 permite una integración fluida entre el software de simulación y el hardware real, lo que facilita la creación de sistemas robóticos complejos y eficientes.

3.5. URDF y SDF

Tanto URDF como SDF son lenguajes de etiquetas XML utilizados para describir la geometría y la cinemática de los robots. URDF (por sus siglas en inglés, Unified Robot Description Format) es un formato de descripción robótica unificado que se utiliza principalmente en la comunidad de ROS para modelar sistemas multibody como brazos robóticos y robots animatrónicos [12]

Por otro lado, SDF (por sus siglas en inglés, Simulation Description Format) es un formato XML desarrollado originalmente para el simulador Gazebo, diseñado para describir objetos y entornos para simuladores de

robots, visualización y control [13]. Ambos formatos son ampliamente utilizados en la simulación y control de robots.

La herramienta `solidworks_urdf_exporter` permite exportar proyectos de SolidWorks a URDF creando un paquete similar al de ROS [14]. Esta herramienta es de gran utilidad ya que permite exportar el modelo 3D de un robot diseñado en SolidWorks a un formato que puede ser utilizado en simulaciones de ROS y Gazebo sin tener que crear un modelo URDF o SDF desde cero.

3.6. EDA

La automatización del diseño electrónico (EDA, por sus siglas en inglés) es un conjunto de herramientas y técnicas utilizadas para diseñar circuitos electrónicos. Estas herramientas permiten a los ingenieros crear esquemas eléctricos, diseñar placas de circuito impreso (PCB) y simular el comportamiento de circuitos antes de fabricarlos. La EDA es esencial en la industria electrónica moderna, ya que permite reducir el tiempo y los costos de desarrollo, así como mejorar la calidad y la fiabilidad de los productos electrónicos [15].

KiCad es un conjunto de software de código abierto para la automatización del diseño electrónico (EDA). KiCad incluye un editor de esquemas, un editor de PCB y un visor 3D. El editor de esquemas permite crear esquemas eléctricos y verificar el diseño con un simulador SPICE integrado y un verificador de reglas eléctricas. El editor de PCB es lo suficientemente accesible para facilitar el diseño de PCBs simples, pero también lo suficientemente potente para diseños complejos. El visor 3D permite inspeccionar fácilmente la PCB para verificar el ajuste mecánico y previsualizar el producto terminado [16].

KiCad es una herramienta muy útil para diseñar circuitos electrónicos y placas de circuito impreso. Además, al ser de código abierto, KiCad es accesible para cualquier persona interesada en el diseño electrónico, lo que lo convierte en una herramienta valiosa para la educación y la investigación.

3.7. Interfaz gráfica

La interfaz gráfica de usuario (GUI, por sus siglas en inglés) es una parte fundamental de cualquier sistema robótico moderno. Proporciona una forma intuitiva y visual de interactuar con el robot, permitiendo a los usuarios controlar y monitorear su funcionamiento. En este proyecto, se utilizó el framework Neutralinojs para crear una GUI que se comunica con el robot a través de un WebSocket. Neutralinojs es un framework de código abierto que permite crear aplicaciones de escritorio multiplataforma utilizando tecnologías web como HTML,

CSS y JavaScript. Su principal ventaja es que no requiere un servidor web para funcionar, lo que lo hace ligero y fácil de implementar [**?**].

El WebSocket es un protocolo de comunicación que permite la comunicación bidireccional entre un cliente y un servidor a través de una conexión persistente. En este proyecto, se utilizó un WebSocket para establecer una conexión entre la GUI y los topicos de ROS2. Esto permite enviar y recibir mensajes para interactuar con el robot de manera sencilla. [**?**]

4. Metodología

4.1. Concepto y planificación

4.1.1. Descripción del robot

El concepto de este proyecto surge de la necesidad de diseñar un sistema robótico simple, eficiente y adaptable que permita realizar movimientos en tres dimensiones. Se optó por un robot cartesiano debido a su facilidad de control, su estructura mecánica directa y su alta precisión en tareas de posicionamiento, siendo ideales para aplicaciones educativas, de investigación y de manufactura ligera.

Desde el inicio, el proyecto se planteó con los siguientes criterios de diseño:

- **Simplicidad mecánica:** Seleccionar mecanismos sencillos como tornillos sin fin y varillas lisas para facilitar el ensamble y el mantenimiento.
- **Bajo costo:** Utilizar componentes de fácil acceso como perfiles de aluminio, motores con encoder económicos y piezas impresas en PLA para reducir el presupuesto sin comprometer el desempeño.
- **Compatibilidad HIL:** Asegurar que el diseño permita su integración con entornos virtuales como Gazebo usando ROS2, para facilitar pruebas y simulaciones sin necesidad de montar todo el sistema físico inicialmente.
- **Modularidad:** Diseñar el sistema de forma que sus componentes puedan ser fácilmente reemplazados, ajustados o mejorados en futuras versiones.

Para la planificación inicial, el proyecto se dividió en las siguientes fases principales:

1. **Diseño conceptual:** Definir las características principales del robot y seleccionar los componentes clave.
2. **Diseño detallado:** Modelar las piezas en 3D y desarrollar el diseño electrónico (PCB, esquemas de conexión).

3. **Simulación:** Crear el entorno virtual utilizando ROS2 y Gazebo para validar el modelo antes del ensamblaje físico.

4. **Construcción:** Ensamblar la estructura mecánica, montar los sistemas electrónicos y realizar pruebas de integración.

5. **Pruebas y ajustes:** Verificar el correcto funcionamiento del sistema tanto en simulación como en hardware, realizando ajustes en los controladores de movimiento.

4.1.2. Especificaciones

El robot cartesiano ha sido diseñado buscando un equilibrio entre simplicidad, precisión y funcionalidad. A continuación, se detallan sus especificaciones junto con una breve explicación de su función en el sistema:

- **Dimensiones:** 390 mm x 390 mm x 345 mm.
  Espacio físico ocupado por el robot, con el fin de que sea compacto y fácil de manipular en entornos escolares o de laboratorio.

- **Peso:** 8.5 kg aproximadamente.
  Determinado por los materiales estructurales (aluminio, motores, piezas impresas), lo que permite movilidad sin necesidad de herramientas especializadas.

- **Carga máxima:** 0.25 kg.
  Suficiente para aplicaciones educativas, pruebas de manipulación o transporte de pequeños objetos sin comprometer la estructura.

- **Velocidad máxima:** 5 mm/s.
  Velocidad controlada que favorece los movimientos automatizados.

- **Precisión:** 1.5 mm.
  Limitada por el paso del tornillo sin fin y la resolución del encoder, adecuada para tareas de posicionamiento básico.

- **Rango de movimiento:** 200 mm en cada eje.
  Definido por la longitud útil de los tornillos sin fin y las guías lisas, permitiendo desplazamientos simétricos en el volumen de trabajo.

- **Ejes:** 3 ejes lineales (X, Y, Z).
  Configuración cartesiana clásica, que permite una cinemática directa simple y fácilmente controlable mediante ROS2.

- **Motores:** 3 motores con encoder JGA25-371.

  Motores de corriente directa con retroalimentación de posición, seleccionados por su buena relación costo-desempeño.

- **Controladores:** 2 módulos L298.

  Permiten el control bidireccional de los motores con suficiente capacidad de corriente para el sistema.

- **Microcontrolador:** 1 placa de desarrollo ESP32-S3.

  Unidad de control principal, con soporte para Wi-Fi, Bluetooth y comunicación serial con ROS2.

- **Sensores:** 3 encoders.

  Instalados en cada motor, proveen información de posición para la estimación del estado del sistema.

- **Alimentación:** 1 fuente de alimentación de 12V 2.5A.

  Capaz de energizar todos los módulos electrónicos y motores simultáneamente sin caídas de voltaje.

- **Conectores:** Conectores JST.

  Facilitan una conexión segura y ordenada de los componentes eléctricos en la PCB.

- **Estructura:** 12 barras de aluminio extruido de 45x45 mm.

  Forman el esqueleto del robot, proporcionando rigidez, facilidad de montaje y modularidad.

- **Soportes:** 6 soportes SK8.

  Mantienen alineadas las varillas lisas que guían los movimientos lineales del sistema.

- **Chumaceras:** 6 chumaceras KP08.

  Proveen soporte rotacional a los tornillos sin fin, reduciendo la fricción.

- **Tornillos:** 3 tornillos sin fin de 8x300 mm.

  Responsables de transformar el giro de los motores en desplazamiento lineal preciso.

- **Varillas:** 3 varillas lisas de 8x300 mm.

  Sirven como guías mecánicas para los carros móviles, asegurando estabilidad en el movimiento.

- **Tuercas:** 3 tuercas de latón T8.

  Trabajan junto con los tornillos sin fin para transmitir el movimiento lineal a los ejes.

- **Acopladores:** 3 acopladores elásticos flexibles de 5x8x25 mm.

  Conectan los motores a los tornillos sin fin, compensando ligeras desalineaciones mecánicas.

- **Piezas impresas:** Varias piezas en PLA.
  Soportes personalizados fabricados con impresión 3D, utilizados para facilitar el montaje de partes críticas.
- **LEDs:** 4 LEDs SMD blancos 5730.
  Indicadores visuales de estado del sistema, distribuidos en la PCB.
- **Resistencias:** 4 resistencias SMD 1206 de 220 Ω.

4.1.3. Requerimientos

Los siguientes requerimientos definen las funcionalidades mínimas que el robot cartesiano debe cumplir para asegurar su correcta operación. En primer lugar, debe permitir el movimiento independiente en tres ejes (X, Y, Z), lo cual es fundamental para realizar desplazamientos controlados en cada eje por separado, esencial para tareas de posicionamiento en espacios tridimensionales.

Además, se requiere una capacidad de carga máxima de 0.25 kg, ideal para experimentar con elementos pequeños y simular procesos industriales a escala. El robot debe operar a una velocidad máxima de 5 mm/s, una velocidad baja adecuada para entornos educativos y pruebas que requieren una observación detallada de su comportamiento. Se busca garantizar una precisión mínima de 1.5 mm en sus movimientos, necesaria para aplicaciones como la inspección o la manipulación básica.

Es crucial que el sistema tenga comunicación con el entorno simulado en Gazebo, integrándose con ROS2 y siendo capaz de enviar y recibir datos, lo que permitirá realizar simulaciones HIL (hardware-in-the-loop). Finalmente, el diseño del robot debe ser modular y reajustable, facilitando el montaje, desmontaje o modificación de sus componentes para futuras pruebas o mejoras, aprovechando la flexibilidad del perfil de aluminio y las piezas impresas.

4.1.4. Limitaciones

Estas limitaciones establecen los márgenes físicos y funcionales dentro de los cuales el robot debe operar. El peso total no debe exceder los 8.5 kg, un límite que busca mantener el robot liviano para facilitar su transporte, manipulación y montaje, además de minimizar la carga sobre los motores. Las dimensiones máximas del robot no deben superar los 390 mm x 390 mm x 545 mm, asegurando su compatibilidad con el espacio de

trabajo disponible y facilitando su integración en estaciones de prueba.

El rango de movimiento se ha limitado a 280 mm por eje, aunque el rango útil planificado es de 200 mm; este límite máximo se establece para evitar colisiones o desajustes. La carga máxima estructural se ha fijado en 1.5 kg, lo que significa que, aunque el robot está diseñado para operar con cargas livianas (0.25 kg), su estructura debe ser capaz de soportar hasta 1.5 kg de manera estática sin comprometer su integridad. Por último, para evitar posibles errores de control o fallos por sobreesfuerzo de los motores, la velocidad máxima absoluta del sistema no debe superar los 10 mm/s durante las pruebas o implementaciones futuras.

4.1.5. Diagrama de actividades

En el anexo A se presenta el diagrama de actividades del proyecto, el cual detalla las acciones clave a llevarse a cabo en cada etapa del desarrollo del robot cartesiano. Este diagrama abarca desde la planificación inicial y el diseño, pasando por la simulación y la construcción, hasta las fases de construcción y pruebas del sistema.

La figura 6 muestra este diagrama de actividades, el cual se puede desglosar de la siguiente manera. Inicialmente, se definió el concepto y la planificación inicial, donde se describieron las características y especificaciones del robot cartesiano, se establecieron los requerimientos y limitaciones del sistema, y se elaboró el diagrama de actividades general del proyecto.

La segunda fase se centra en el diseño del robot, abarcando tanto el diseño electrónico como el mecánico. Durante esta etapa, se diseñó la PCB del robot, se modelaron las piezas en 3D y se generaron los archivos de mallas y URDF necesarios para la simulación en Gazebo. Posteriormente, se llevó a cabo la simulación del robot cartesiano utilizando el entorno de ROS2, donde se creó el paquete de ROS2 y se configuraron los programas de simulación pertinentes.

La fase de construcción implica el montaje físico del robot cartesiano, utilizando los componentes mecánicos y electrónicos previamente diseñados. En esta etapa, se inició el ensamblaje del robot, la instalación de los motores y la conexión de los diversos componentes electrónicos. Finalmente, se encuentra la fase de pruebas, donde se verificó el correcto funcionamiento del robot en el entorno simulado, realizando los ajustes necesarios para asegurar su funcionamiento esperado en el entorno real.

4.2. Diseño del robot

4.2.1. Diseño electrónico

El diseño electrónico del robot cartesiano se llevó a cabo mediante el software KiCad. El esquema eléctrico resultante integra los módulos controladores de motores L298, las conexiones para los motores con encoder JGA25-371, la placa ESP32-S3, los LEDs indicadores y la fuente de alimentación de 12V. La placa de circuito impreso (PCB) fue diseñada específicamente para facilitar la conexión de los encoders de los motores a la placa ESP32-S3, asegurando además una alimentación adecuada para todo el sistema a través de un conector JST destinado a la fuente de 12V. El esquema eléctrico detallado se encuentra en el anexo B, figura 7.

El esquema electrónico se organiza en ocho secciones distintas, cada una representando un componente o subsistema del circuito. Estas secciones corresponden a la placa de desarrollo ESP32-S3, que incluye los pines de alimentación, tierra, los encoders, los controladores de motores y los LEDs; dos secciones dedicadas a los controladores de motores L298, detallando sus conexiones de alimentación, tierra, los motores y el micro-controlador; tres secciones para los motores con encoder JGA25-371, mostrando sus pines de alimentación y las conexiones de los encoders; una sección para los LEDs del sistema, incluyendo sus conexiones al micro-controlador y las resistencias limitadoras de corriente; y finalmente, una sección que representa la fuente de alimentación, con su conector BarrelJack para la entrada de 12V.

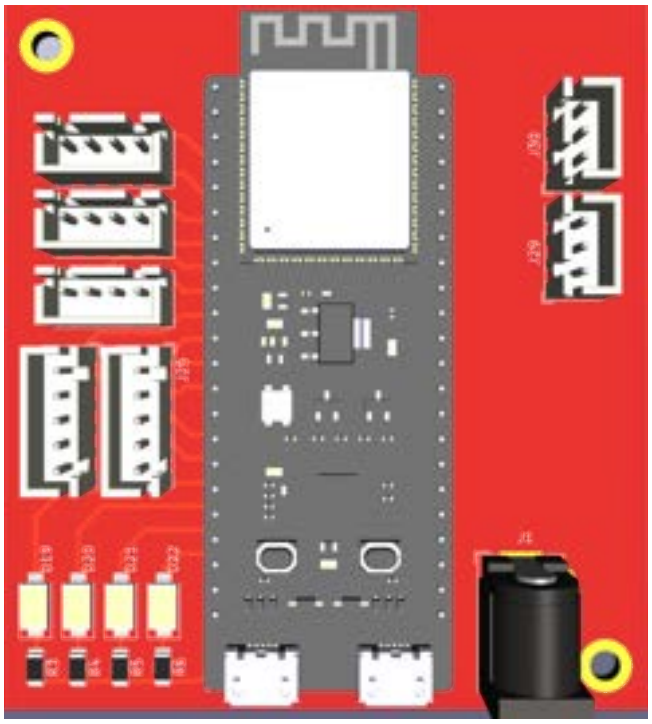


Figura 1: Diseño 3D de la PCB del robot cartesiano.

La figura 1 presenta el diseño de la PCB del robot cartesiano, mostrando la disposición de los componentes principales. En la parte central de la PCB se ubica la placa de desarrollo ESP32-S3. En la parte izquierda media superior se encuentran tres conectores JST-B4B-XH-A destinados a la conexión de los encoders de los motores. Un poco más abajo, en la parte izquierda media, se sitúan dos conectores JST-B5B-XH-A para las entradas de los controladores de motores L298.

En la parte derecha media superior de la PCB se encuentran dos conectores JST-B3B-XH-A para la alimentación de los controladores de motores L298. En la parte derecha inferior se ha dispuesto un conector BarrelJack para la conexión de la fuente de alimentación de 12V. Finalmente, en la parte izquierda inferior se ubican cuatro resistencias SMD 1206 de 220 Ω para limitar la corriente de los LEDs, junto con cuatro LEDs SMD blancos 5730, situados en la parte izquierda media inferior para indicar el estado del sistema.

4.2.2. Diseño 3D

El diseño tridimensional del robot cartesiano se llevó a cabo utilizando el software SolidWorks, con el objetivo de visualizar de forma precisa la disposición de los componentes antes de su construcción física. Esta etapa fue esencial para validar que todas las piezas encajaran correctamente, optimizar el espacio de trabajo y facilitar futuras integraciones de hardware.

La estructura principal fue modelada utilizando 12 perfiles de aluminio extruido de 45x45 mm, seleccionados por su rigidez, modularidad y facilidad de ensamblaje. Estos perfiles conforman una base cuadrada de 390 mm x 390 mm, proporcionando la estabilidad necesaria para soportar los movimientos de los tres ejes.

Cada uno de los tres ejes (X, Y, Z) fue diseñado de manera independiente:

- **Eje X:** Se modeló una varilla lisa de 8 mm de diámetro y 300 mm de largo, acompañada de un tornillo sin fin de las mismas dimensiones. El motor JGA25-371, montado en un soporte impreso en PLA, se acopla al tornillo mediante un acoplador flexible de 5x8x25 mm. El soporte del motor y las chumaceras KP08 fueron también diseñados e impresos en 3D para garantizar una alineación precisa.
- **Eje Y:** Similar al eje X, pero posicionado verticalmente. Se emplearon rodamientos SC8 para conectar el movimiento del eje anterior al nuevo eje, utilizando un adaptador diseñado en PLA que permite la unión de varillas y tuercas T8.
- **Eje Z:** Utiliza también un sistema de varilla y tornillo sin fin. Para este eje, se diseñaron adaptadores especiales que permiten la conexión segura entre los movimientos del eje Y y Z mediante elementos impresos en PLA.

Durante el proceso de modelado, se consideraron varios factores importantes:

- **Compatibilidad estructural:** Asegurar que todos los componentes mecánicos y electrónicos pudieran integrarse sin interferencias.
- **Modularidad:** Permitir el fácil desmontaje o reajuste de piezas individuales para facilitar el mantenimiento o la mejora del sistema.
- **Minimización de peso:** Utilizar soportes ligeros impresos en PLA para reducir la carga sobre los motores, manteniendo al mismo tiempo la rigidez necesaria.
- **Facilidad de manufactura:** Diseñar piezas que pudieran imprimirse en 3D de forma sencilla, considerando tolerancias de impresión y facilidad de ensamble.

El modelado en SolidWorks permitió también exportar las piezas en formato STL, que posteriormente se utilizaron para la creación de los archivos URDF necesarios para la simulación en Gazebo. Esta integración asegura que tanto el entorno virtual como el físico compartan dimensiones, pesos y geometrías consistentes, optimizando el desarrollo de la estrategia de control.

Cabe destacar que durante la fase de diseño se realizaron varios rediseños y adaptaciones, especialmente en los soportes de motores y la disposición de las chumaceras, con el fin de mejorar la robustez general del sistema y optimizar el ensamblaje final.

4.3. Simulación del robot

4.3.1. Instalación de ROS2 y Gazebo

La forma en que esta construido ROS2 permite instalarlo en función de la versión de Ubuntu que se tenga. Uno de las dificultades que se pueden presentar son las configuraciones necesarias antes de realizar la instalación como habilitar el paquete `Universe` o agregar el repositorio de ROS2 a la lista de repositorios de Ubuntu. Para automatizar la instalación de ROS2 se desarrolló un script de instalación que permite instalar ROS2, Gazebo, μROS y paquetes adicionales necesarios para la simulación del robot cartesiano. Este script se encuentra en el anexo E, listing 1.

El script se encarga de obtener la versión de Ubuntu instalada en el sistema y, dependiendo de la versión, instala la versión correspondiente de ROS2. El script también instala Gazebo según la versión de ROS2 instalada. Además, instala los paquetes adicionales necesarios para la simulación del robot cartesiano, como `urdf_launch`. Con ROS2 y Gazebo instalados, el script descarga el paquete de μROS y lo instala como un paquete de ROS2.

Finalmente, el script descarga la librería de ROS2 para Arduino y muestra un mensaje correspondiente a configuraciones recomendadas para facilitar el uso de ROS2. Los mensajes que muestra el script dependen del idioma del sistema operativo soportando, actualmente solo el español y el inglés. El script se ejecuta con el comando `$SHELL install-ros.sh`.

4.3.2. Creación del entorno de trabajo

Teniendo el archivo de URDF del robot cartesiano y los archivos STL generados por *solidworks_urdf_exporter*, se creó una carpeta para ser usada como espacio de trabajo para el robot cartesiano. La idea es usar ROS2 como la plataforma de integración de hardware y software, permitiendo la comunicación entre el robot físico y el entorno simulado. La red de tópicos de ROS2 desarrollada en este proyecto se encuentra en el anexo D, figura 32.

Usando `ros2 pkg create` [6] se creó un paquete para el robot cartesiano llamado **carobot**. Entre las carpetas

que contiene el paquete cabe mencionar las siguientes:

- **carobot**: carpeta principal del paquete.
  - **config**: contiene un archivo YAML con la configuración del robot generado por *solidworks_urdf_exporter*.
  - **launch**: contiene los archivos de lanzamiento del paquete.
  - **urdf/**: contiene el archivo URDF del robot.
    - **meshes/**: contiene las mallas STL del robot.
  - **rviz/**: contiene el archivo de configuración de RViz2.
  - **meshes/**: contiene las mallas STL del robot.
  - **src/**: contiene scripts de Python del paquete.

Dentro del paquete se modificó el archivo `CMakeList.txt` (anexo F, listing 2) para que compile el paquete correctamente. Se agregaron las carpetas `launch`, `urdf`, `config` y `rviz`; y los scripts de Python `pos_put.py` y `pos_get.py` a la lista de archivos a compilar.

Dentro de la carpeta launch se creó un archivo de lanzamiento llamado **display.launch.py** (anexo F, listing 6) para abrir Gazebo y RViz2 con la configuración del robot. Este archivo de lanzamiento se encarga de abrir Gazebo y RViz2, cargar el modelo del robot en ambos entornos y crear los puentes de comunicación entre los tópicos de Gazebo y ROS2. El archivo de lanzamiento se ejecuta con el comando `ros2 launch carobot display.launch.py` y se encarga de:

1. Abrir RViz2 con la configuración de la carpeta rviz y el modelo del robot.
2. Cargar el modelo del robot a un mundo vacio de gazebo con `ros_gz_sim create` [7].
3. Crear puentes de comunicación entre los tópicos de Gazebo y ROS2 usando `ros_gz_bridge` [8].
4. Enviar las posiciones de las juntas del robot en Gazebo a RViz2 usando los puentes creados.
5. Abrir Gazebo con el modelo del robot.

También se creo una interfaz de linea de comandos para enviar las posiciones deseadas de las juntas del robot a Gazebo en un archivo de Python `pos_put.py`. También se creó otro archivo con el fin de leer las posiciones de las juntas del robot en Gazebo y enviarlas a RViz2 (`pos_get.py`). Los archivos `pos_put.py` y `pos_get.py` se encuentran en el anexo F, listings 5 y 6 respectivamente.

4.3.3. Visualización del robot

El lanzador **display.launch.py** se ejecuta con el comando `ros2 launch carobot display.launch.py`. Este lanzador abre RViz2 y Gazebo, mostrando el modelo del robot en ambos entornos. En RViz2 se pueden observar las posiciones de las juntas del robot y en Gazebo se puede ver el modelo del robot en un mundo vacío. El lanzador como tal no tiene un mundo definido, por lo que se abre un mundo vacío.

A pesar de que el lanzador crea conexiones entre los tópicos de Gazebo y RViz2, no es posible mover el robot en Gazebo. Para esto es necesario enviar las posiciones de las juntas del robot a Gazebo ya sea publicando en los tópicos de Gazebo o mediante un script. En la siguiente sección se explica el rol de los scripts desarrollados para mover el robot en Gazebo y RViz2.

4.3.4. Simulación del movimiento

Para que Gazebo lea las posiciones de las juntas del robot y simule su movimiento, se agregó el plugin `gz-sim-joint-position-controller-system` al archivo URDF del robot. Este plugin se encarga de leer las posiciones de las juntas del robot y simular su movimiento en Gazebo [10]. El plugin se agrega al archivo URDF dentro de una etiqueta `gazebo`, 1 plugin por cada eje del robot. Dentro de la etiqueta `gazebo` también se agregó el plugin `gz-sim-joint-state-publisher-system` para enviar las posiciones de las juntas del robot en la simulación en Gazebo a RViz2 [11].

De esta forma, el script `pos_put.py` se encarga de enviar posiciones como datos de tipo `Double` a los tópicos de Gazebo. Estos tópicos son leídos por el plugin `gz-sim-joint-position-controller-system` y simulan el movimiento del robot en Gazebo. El plugin `gz-sim-joint-state-publisher-system` se encarga de enviar las posiciones de las juntas del robot como datos de tipo `JointState`. En otro script se juntan los datos de posición de las juntas y se envían a RViz2 en un solo tópico.

La etiqueta `gazebo` de el archivo URDF se encuentra en el anexo F, listing 3 pero todo el entorno de trabajo se encuentra en un repositorio de Github (`https://github.com/adnksharp/cartesian-ppp-ros`).

4.4. Construcción del robot

La construcción física del robot cartesiano se llevó a cabo en varias etapas, siguiendo el diseño 3D previamente modelado en SolidWorks y los esquemas electrónicos generados en KiCad. El proceso abarcó desde la adquisición de materiales, la fabricación de piezas, el ensamblaje mecánico y las conexiones eléctricas.

Adquisición de materiales. Para llevar a cabo la construcción, se adquirieron los siguientes componentes principales:

- **Materiales estructurales:** Perfiles de aluminio extruido de 45x45 mm, tornillería M6, ángulos metálicos de unión.
- **Componentes de movimiento:** Motores con encoder JGA25-371, varillas lisas de 8x300 mm, tornillos sin fin de 8x300 mm, tuercas de latón T8, acopladores flexibles 5x8x25 mm.
- **Componentes de soporte:** Chumaceras KP08, soportes SK8, rodamientos SC8.
- **Componentes electrónicos:** Placa de desarrollo ESP32-S3, módulos controladores L298, fuente de alimentación de 12V 2.5A, conectores JST, LEDs SMD 5730 y resistencias SMD 1206 de 220Ω.
- **Piezas impresas:** Soportes de motor, adaptadores para chumaceras, soportes para rodamientos y fijaciones, fabricados en PLA.

Algunos componentes auxiliares como cables, conectores, varillas roscadas adicionales y material de sujeción también fueron adquiridos para completar el ensamblaje.

Ensamblaje mecánico. El ensamblaje del robot inició con la construcción de la base cuadrada de 390 mm x 390 mm utilizando los perfiles de aluminio y ángulos metálicos. Posteriormente, se instalaron las varillas lisas y los tornillos sin fin en los perfiles superiores, asegurándolos mediante soportes SK8 y chumaceras KP08 fijadas a través de piezas impresas en 3D.

Cada motor JGA25-371 se montó en su respectivo soporte impreso y se acopló al tornillo sin fin correspondiente utilizando un acoplador elástico flexible. Las tuercas T8 fueron insertadas en carros móviles diseñados específicamente para permitir el desplazamiento a lo largo de las varillas lisas. El ensamblaje de los ejes se realizó de forma modular, primero ensamblando el eje X, luego montando sobre éste el eje Y, y finalmente integrando el eje Z sobre el conjunto anterior, respetando la disposición definida en el diseño 3D.

Conexiones eléctricas. La placa de circuito impreso (PCB) diseñada en KiCad fue ensamblada y soldada, integrando la ESP32-S3, los módulos L298, los LEDs indicadores y los conectores JST. Se realizaron pruebas de continuidad y verificación de conexiones para evitar errores eléctricos. Debido a un ajuste de último momento, se modificó la forma de montar la placa ESP32-S3, utilizando conectores tipo header para adaptarla correctamente a la PCB.

Finalmente, los motores fueron conectados a los controladores de motor y la PCB fue alimentada a través de una fuente externa de 12V, preparando el sistema para las primeras pruebas de movimiento y comunicación.

Dificultades encontradas. Durante el proceso de construcción se presentaron algunos desafíos:

- **Problemas de tolerancia en piezas impresas:** Algunas piezas impresas inicialmente no se ajustaban de manera óptima, lo que requirió retrabajos o ajustes en los modelos 3D.
- **Montaje de la ESP32-S3:** El tamaño real de la placa de desarrollo no coincidía exactamente con el diseño inicial de la PCB, por lo que se tuvo que modificar la instalación utilizando conectores adicionales.
- **Ajuste de varillas y tornillos:** Fue necesario verificar y ajustar manualmente la alineación de varillas lisas y tornillos sin fin para asegurar un movimiento suave en todos los ejes.

Gracias a la adecuada planificación y flexibilidad en el diseño, fue posible superar estos problemas y continuar con el avance del proyecto.

4.5. Interfaz gráfica

La interfaz gráfica de usuario (GUI) fue desarrollada utilizando Neutralinojs. Usando 'neu', el comando de línea de comandos de Neutralinojs, se creó un proyecto simple. Este proyecto se basa en un archivo HTML que contiene etiquetas para:

- **Estado de comunicación:** Leer el estado de comunicación de ROS2, el socket de comunicación, μROS y Gazebo.
- **Posicion de las juntas:** Leer la posición de las juntas del robot y mostrarlas en la GUI.
- **Comandos de movimiento:** Enviar comandos de movimiento al robot.
- **Tareas:** Leer el estado de las posiciones que el robot debe alcanzar.
- **Historial de movimiento:** Visualizar el historial de movimiento del robot.

Neutralinojs permite crear ejecutables para Windows, Linux y MacOS. El proyecto se encuentra en el repositorio `https://github.com/adnksharp/cartesian-ppp-ros`. La interfaz gráfica permite al usuario interactuar con el robot de manera sencilla, enviando comandos de movimiento y visualizando el estado del sistema.

4.6. Pruebas

Se procedió a realizar pruebas iniciales de funcionamiento, las cuales abarcaron tanto la comunicación entre nodos como el control de movimiento básico.

4.6.1. Pruebas de comunicación

Se realizaron pruebas para verificar la correcta comunicación entre el robot simulado en Gazebo y los nodos de ROS2. Estas pruebas consistieron en:

- Publicar datos de posiciones deseadas hacia los controladores de Gazebo mediante el paquete `ros_gz_bridge`.
- Suscribir nodos de ROS2 a los estados de las juntas simuladas en Gazebo y visualizar el resultado en RViz2.
- Verificar que los mensajes `JointState` y `Float64` se transmitieran correctamente a través de los tópicos configurados.

Durante estas pruebas se identificaron algunas dificultades menores, como configuraciones incorrectas en los nombres de las articulaciones en los archivos URDF, las cuales fueron corregidas para asegurar una correcta correspondencia entre el modelo simulado y el entorno ROS2.

4.6.2. Pruebas de movimiento

Posteriormente, se realizaron pruebas para validar el movimiento controlado del robot en el entorno de simulación. Estas pruebas incluyeron:

- Enviar comandos de posición individuales a cada eje (X, Y, Z) de manera independiente para observar su respuesta en Gazebo.
- Validar que los movimientos fueran reflejados en tiempo real en la interfaz de RViz2.
- Ajustar la ganancia de los controladores PID del plugin `gz-sim-joint-position-controller-system` a través de sintonización manual (prueba y error).

La sintonización inicial del PID permitió obtener un movimiento aceptable con las siguientes ganancias:

- $K_p = 68.21$
- $K_i = 0.56$
- $K_d = 34.14$

Aunque se logró un movimiento razonablemente suave, se detectaron pequeñas oscilaciones en el posicionamiento final, por lo que se planea realizar una futura sincronización basada en el modelo físico del robot para optimizar la respuesta dinámica.

4.6.3. Observaciones generales

Durante las pruebas se comprobó que el sistema es capaz de:

- Recibir y procesar comandos de movimiento.
- Simular correctamente la respuesta mecánica en Gazebo.
- Visualizar en tiempo real la retroalimentación de las posiciones articulares en RViz2.

Estos resultados preliminares validan la correcta integración entre el modelo virtual y el entorno de simulación, sentando la base para las siguientes fases de integración con el hardware físico (Hardware-in-the-Loop).

5. Resultados

Este capítulo presenta los resultados obtenidos durante el desarrollo parcial del robot cartesiano, evaluando el desempeño de las diferentes etapas de diseño, simulación y construcción.

5.1. Resultados del diseño 3D

El diseño 3D del robot permitió validar correctamente la disposición de los componentes estructurales, asegurando la compatibilidad dimensional y la modularidad deseada. Las piezas impresas en PLA se adaptaron adecuadamente a los perfiles de aluminio y a los motores, con pequeñas correcciones necesarias en tolerancias, principalmente en los soportes de motor y adaptadores de chumaceras. Los archivos de KiCad se encuentran en la nube:

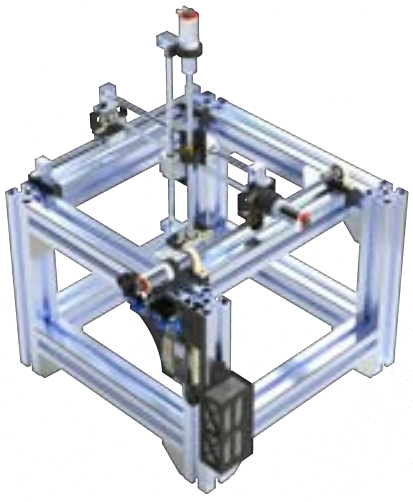
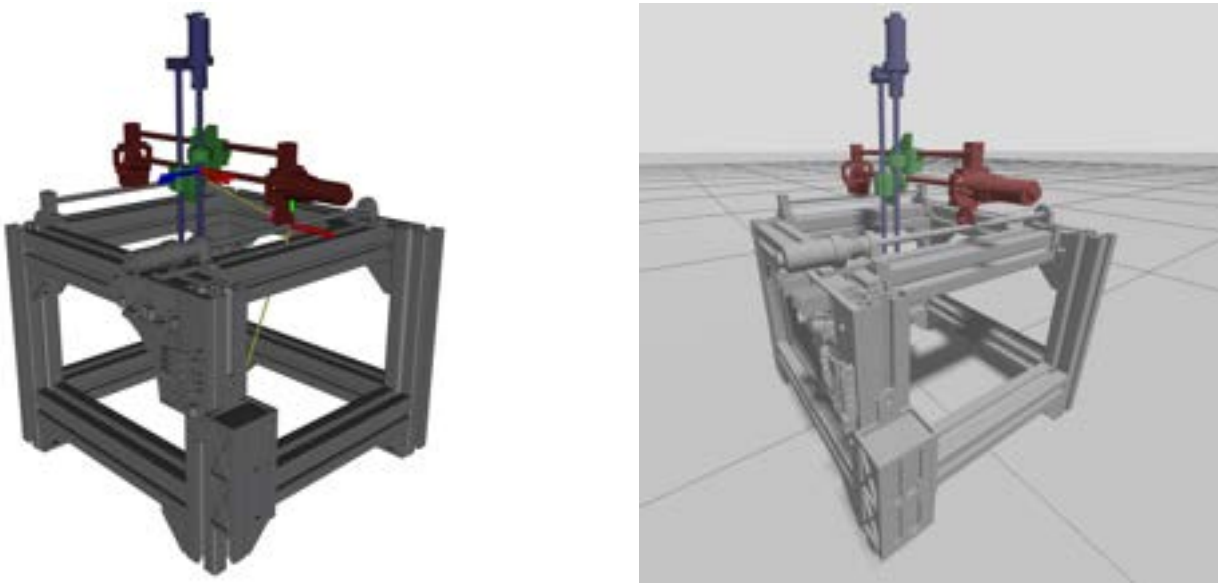


Figura 2: Diseño 3D del robot cartesiano realizado en SolidWorks.

Los planos de las piezas se encuentran en el anexo C y las piezas creadas se almacenaron en la nube:

https://alumnosuacj.sharepoint.com/:f:/s/Section_445919-equipo3/EisrVS2E7jBPuNGVBnNSQIMBEK5nDnH8iA342m-mQfGQ1A?e=gtPAZ7

5.2. Resultados de la simulación



(a) Visualización del robot en RViz2.

(b) Visualización del robot en Gazebo.

Figura 3: Visualización del robot en RViz2 y Gazebo.

La simulación del robot en Gazebo, controlada a través de ROS2, fue exitosa. Se logró visualizar el movimiento de los tres ejes (X, Y, Z) en RViz2, enviando comandos de posición individual a cada junta. La respuesta en la simulación fue satisfactoria, con desplazamientos estables y tiempo de respuesta adecuado.

No obstante, se detectaron ligeras oscilaciones en el posicionamiento final, atribuibles a la configuración inicial de los parámetros PID, lo que sugiere que una sintonización basada en modelos dinámicos podrá mejorar la respuesta en futuras versiones.

La figura 3a muestra la visualiaszación del robot en RViz2 y la figura 3b muestra la visualización del robot en Gazebo.

5.3. Resultados de la construcción

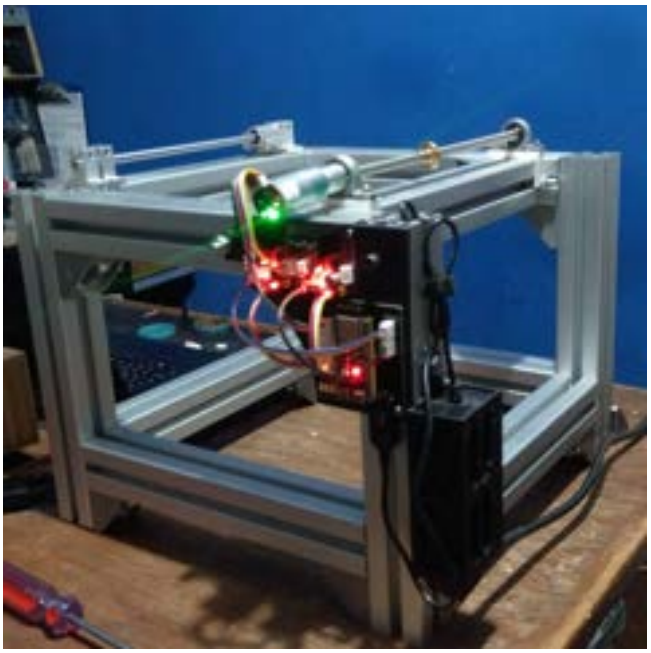


Figura 4: Construcción del robot cartesiano.

Se ensambló de manera exitosa la estructura base del robot (390 mm x 390 mm), los sistemas de movimiento de los tres ejes, y el montaje preliminar de los motores y sistemas de transmisión. Se integró la electrónica básica, incluyendo la PCB con la ESP32-S3 y los módulos controladores L298.

Durante el ensamblaje, se presentaron ajustes menores debidos a variaciones en la impresión 3D y diferencias entre el diseño teórico y las dimensiones reales de algunos componentes. Sin embargo, estos inconvenientes fueron solucionados de manera satisfactoria.

5.4. Resultados de la interfaz gráfica

Figura 5: Interfaz gráfica del robot cartesiano.

La interfaz gráfica desarrollada con Neutralinojs permite al usuario interactuar de manera sencilla con el robot cartesiano. La GUI muestra el estado de comunicación, las posiciones actuales de las juntas y permite enviar comandos de movimiento como se planeó inicialmente.

La interfaz se ejecuta como una aplicación independiente, facilitando la interacción con el robot sin necesidad de abrir múltiples terminales o entornos de desarrollo. La figura 5 muestra la interfaz gráfica en funcionamiento.

5.5. Resultados de las pruebas

En las pruebas de comunicación, se logró establecer una transmisión efectiva de comandos de posición y lectura de estados de las juntas entre ROS2 y Gazebo. La visualización en RViz2 permitió confirmar que los movimientos programados se reflejaban correctamente en el entorno simulado.

En las pruebas de movimiento, los controladores PID configurados permitieron movimientos suaves y controlados en cada eje. Aunque hubo ligeras oscilaciones, el robot pudo alcanzar las posiciones deseadas con una precisión aceptable considerando las ganancias de control iniciales. Estos resultados preliminares confirman que el sistema virtual del robot está operando de manera correcta, y que la transición hacia la integración hardware-in-the-loop será factible en las siguientes etapas del proyecto.

6. Conclusiones

Este documento representa un avance de proyecto tras un mes de desarrollo. Se ha logrado progresar en el modelado 3D, adquisición de materiales, diseño electrónico y simulación inicial del robot cartesiano. Aún queda trabajo por hacer, particularmente en la programación de controladores, integración HIL y pruebas funcionales. Sin embargo, los resultados hasta ahora son prometedores y sientan una base sólida para continuar con el proyecto.

7. Referencias

Referencias

[1] A. Osmanovic, B. Dedić, S. Čosić, E. trakić, and M. Bečirović, “Modeling and analysis of a cartesian coordinate robot,” 12 2023.

[2] J. A. Dena Aguilar, J. C. Delgado Flores, J. Acevedo Martínez, V. M. Velasco Gallardo, E. Zacarías Moreno, E. J. Martínez Delgado, and N. I. Escalante García, “Design, manufacture, and control motion of cartesian robot prototype with 5-dof robot arm for possible spinning applications at laboratory-level.” *Nova Scientia*, vol. 15, no. 30, pp. 1–24, May 2023. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=5d424927-2b03-3000-b83f-42f0a74a1aae

[3] MATLAB, “Conceptos básicos de la simulación de hardware-in-the-loop,” https://la.mathworks.com/help/simscape/ug/what-is-hardware-in-the-loop-simulation.html, accessed: 2025-04-06.

[4] 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. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[5] R. Automation, “Rviz2 - user manual,” https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html, accessed: 2025-03-15.

[6] O. Robotics, “Creating a package,” https://github.com/ros/solidworks_urdf_exporter, accessed: 2025-02-25.

[7] ——, “Ros + gazebo sim,” https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_sim, accessed: 2025-03-18.

[8] ——, “Bridge communication between ros and gazebo,” https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge, accessed: 2025-03-25.

[9] ——, “About gazebo,” https://gazebosim.org/about, accessed: 2025-03-09.

[10] ——, “Joint controllers,” https://gazebosim.org/api/sim/8/jointcontrollers.html, accessed: 2025-03-25.

[11] ——, “joint_state_publisher,” https://wiki.ros.org/joint_state_publisher, accessed: 2025-03-25.

[12] MATLAB, “Urdf primer,” https://la.mathworks.com/help/sm/ug/urdf-model-import.html, accessed: 2025-03-29.

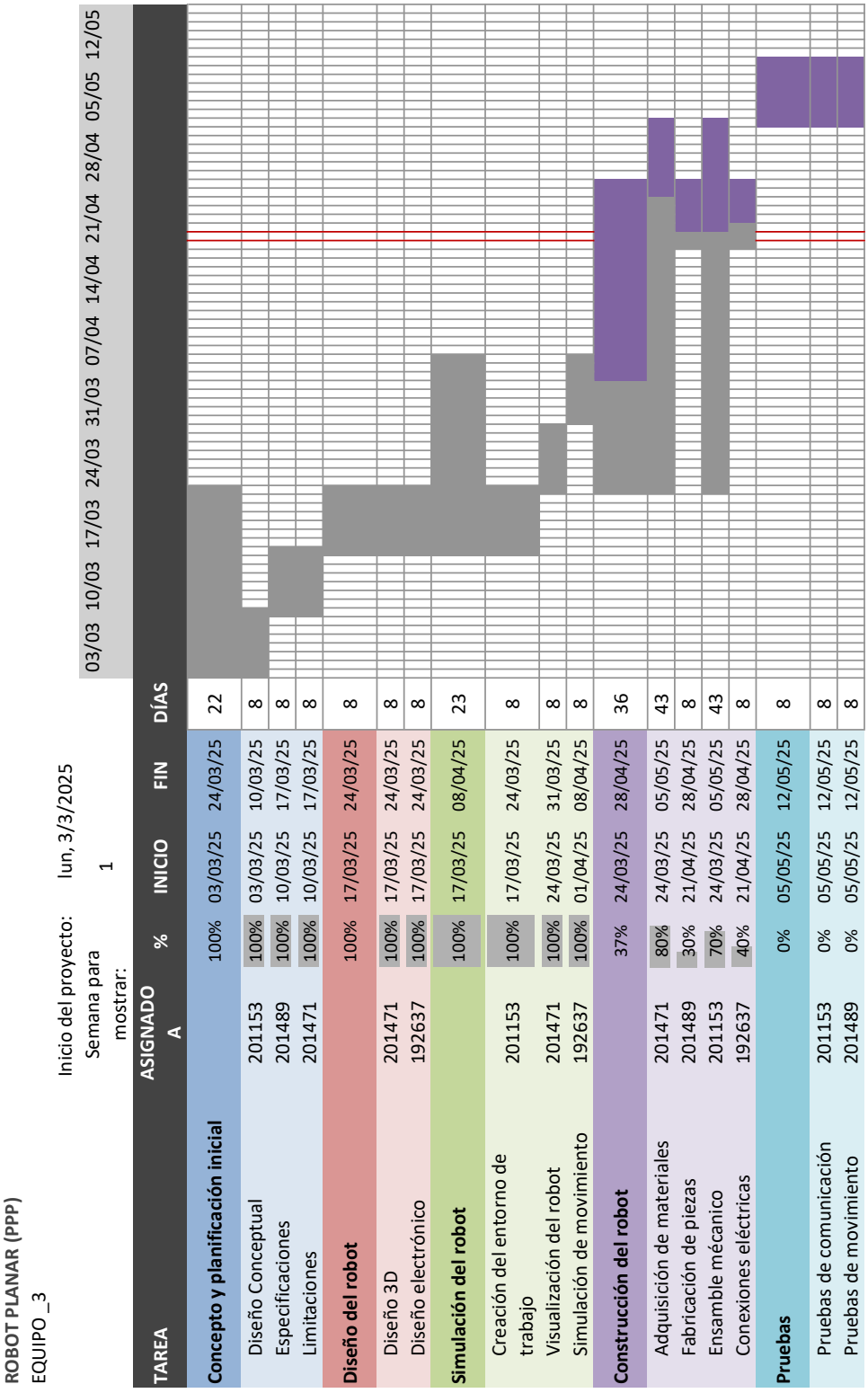
[13] O. S. R. Foundation, “What is sdformat,” http://sdformat.org, accessed: 2025-03-09.

[14] S. Brawner, “Solidworks to urdf exporter,” https://github.com/ros/solidworks_urdf_exporter, accessed: 2025-03-07.

[15] M. Birnbaum, *Essential electronic design automation (EDA)*. Prentice Hall Professional, 2004.

[16] KiCad, “Kicad eda - schematic capture and pcb design software,” https://www.kicad.org, accessed: 2025-04-02.

A. Anexo 1: Diagrama de actividades



TAREA	ASIGNADO A	%	INICIO	FIN	DÍAS
Concepto y planificación inicial		100%	03/03/25	24/03/25	22
Diseño Conceptual	201153	100%	03/03/25	10/03/25	8
Especificaciones	201489	100%	10/03/25	17/03/25	8
Limitaciones	201471	100%	10/03/25	17/03/25	8
Diseño del robot		100%	17/03/25	24/03/25	8
Diseño 3D	201471	100%	17/03/25	24/03/25	8
Diseño electrónico	192637	100%	17/03/25	24/03/25	8
Simulación del robot		100%	17/03/25	08/04/25	23
Creación del entorno de trabajo	201153	100%	17/03/25	24/03/25	8
Visualización del robot	201471	100%	24/03/25	31/03/25	8
Simulación de movimiento	192637	100%	01/04/25	08/04/25	8
Construcción del robot		37%	24/03/25	28/04/25	36
Adquisición de materiales	201471	80%	24/03/25	05/05/25	43
Fabricación de piezas	201489	30%	21/04/25	28/04/25	8
Ensamble mécanico	201153	70%	24/03/25	05/05/25	43
Conexiones eléctricas	192637	40%	21/04/25	28/04/25	8
Pruebas		0%	05/05/25	12/05/25	8
Pruebas de comunicación	201153	0%	05/05/25	12/05/25	8
Pruebas de movimiento	201489	0%	05/05/25	12/05/25	8

Figura 6: Cronograma de actividades del proyecto.

B. Anexo 2: Esquema eléctrico

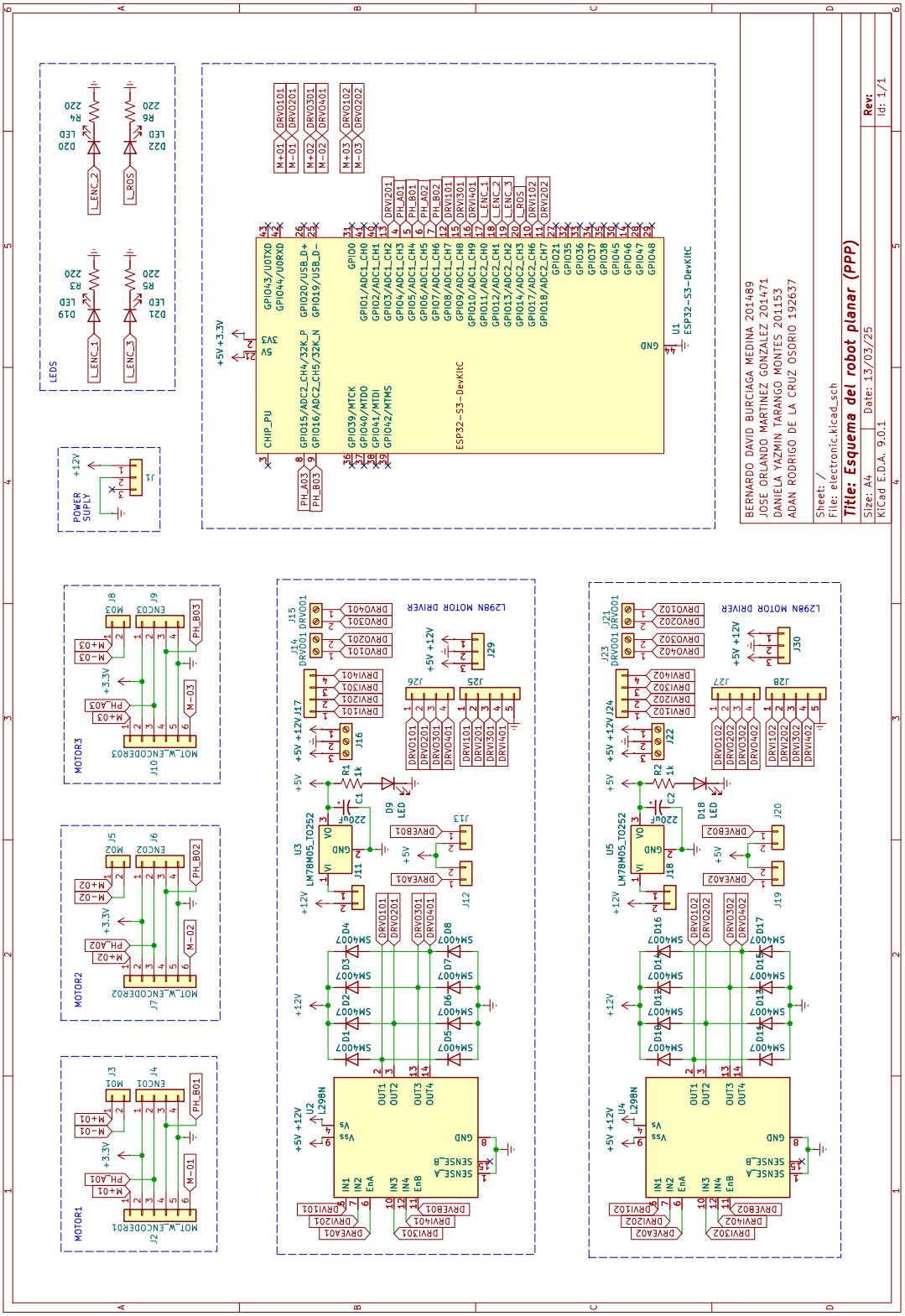


Figura 7: Esquema eléctrico del robot cartesiano.

C. Anexo 3: Planos de piezas

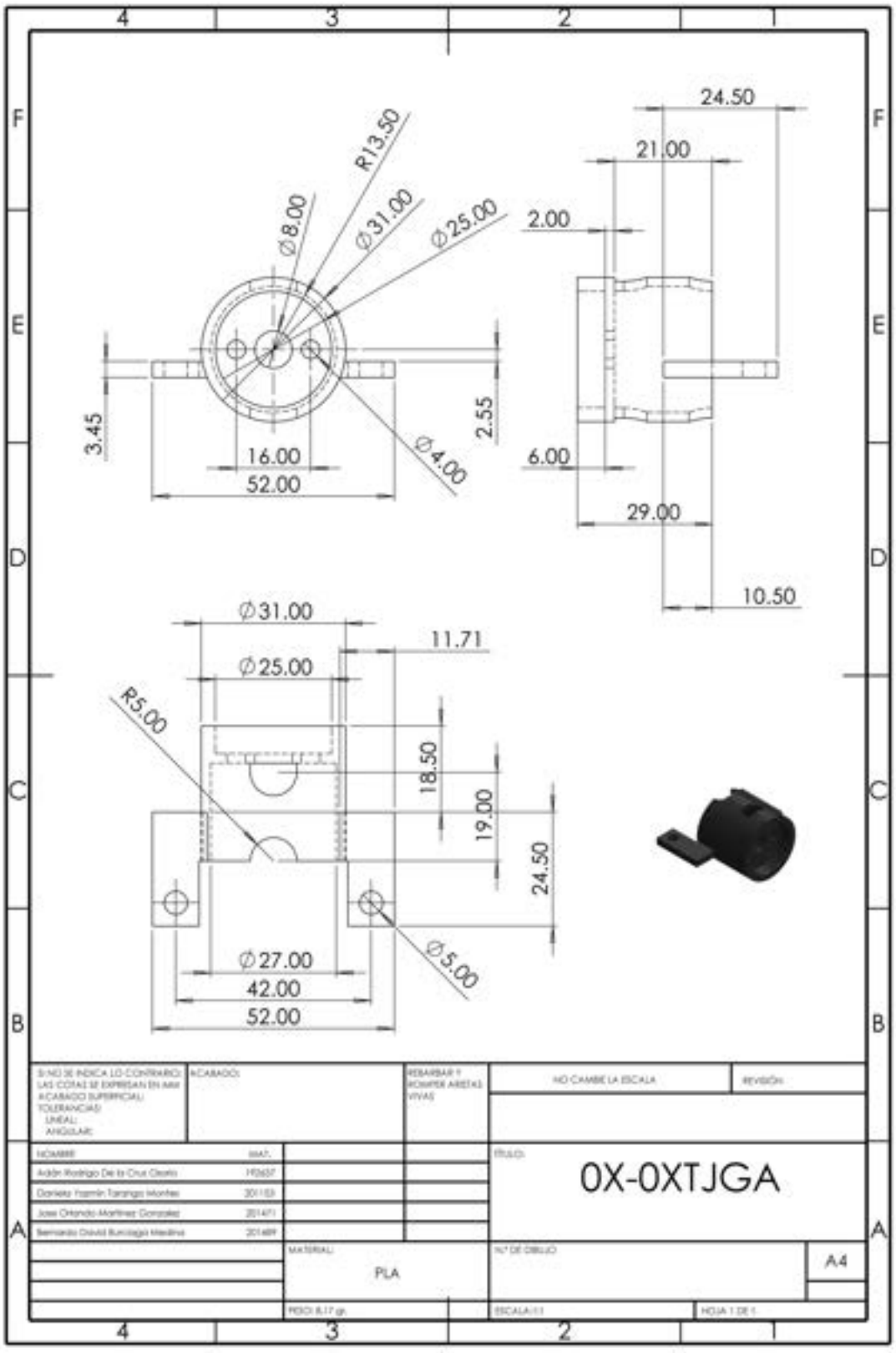


Figura 8: **0X-0XTJGA**: Soporte para conectar el motor al soporte **0X-KP08TSK08**.

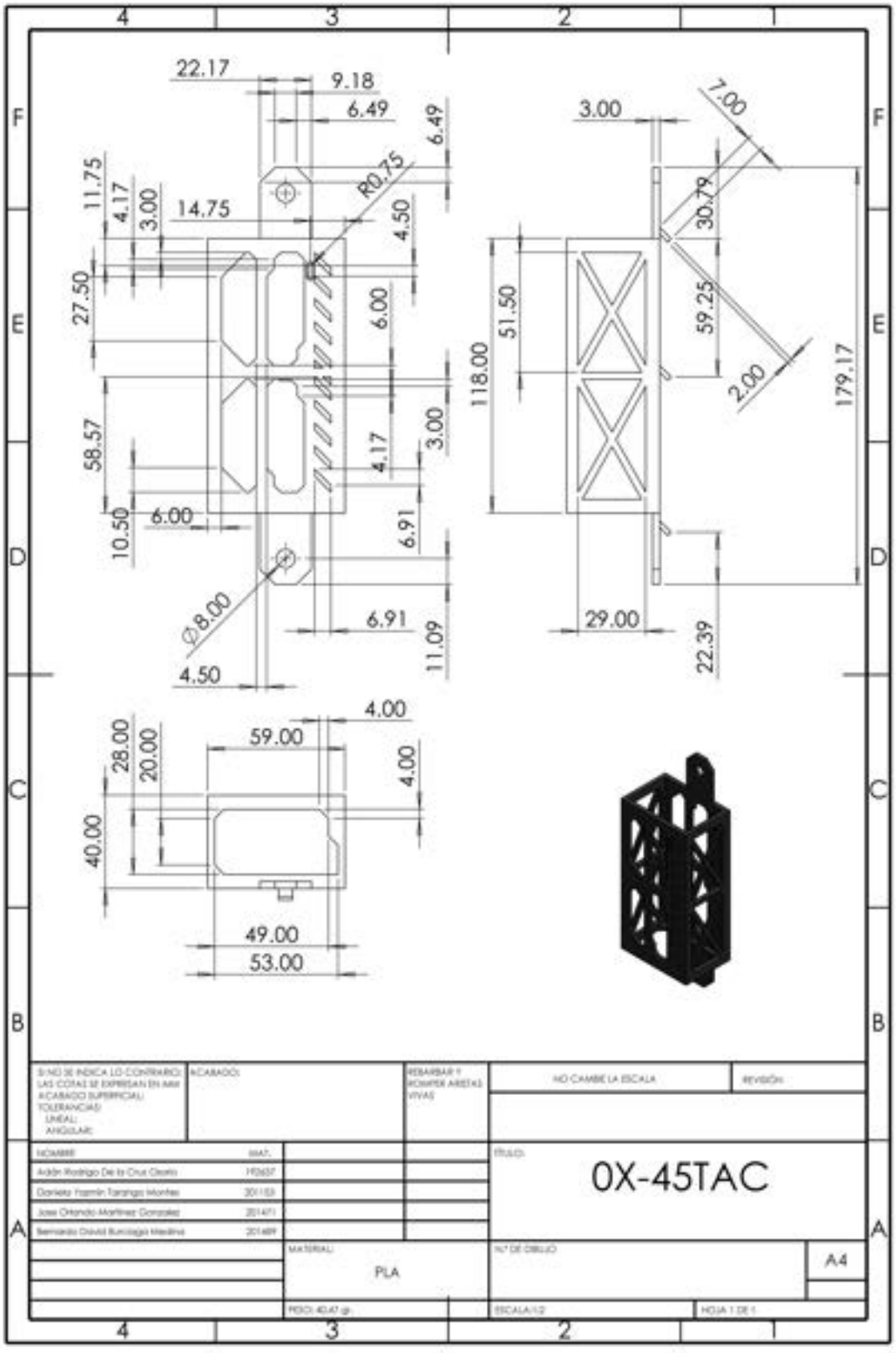


Figura 9: **0X-45TAC**: Soporte para conectar el adaptador de corriente AC **SW30W12** con el perfil de aluminio.

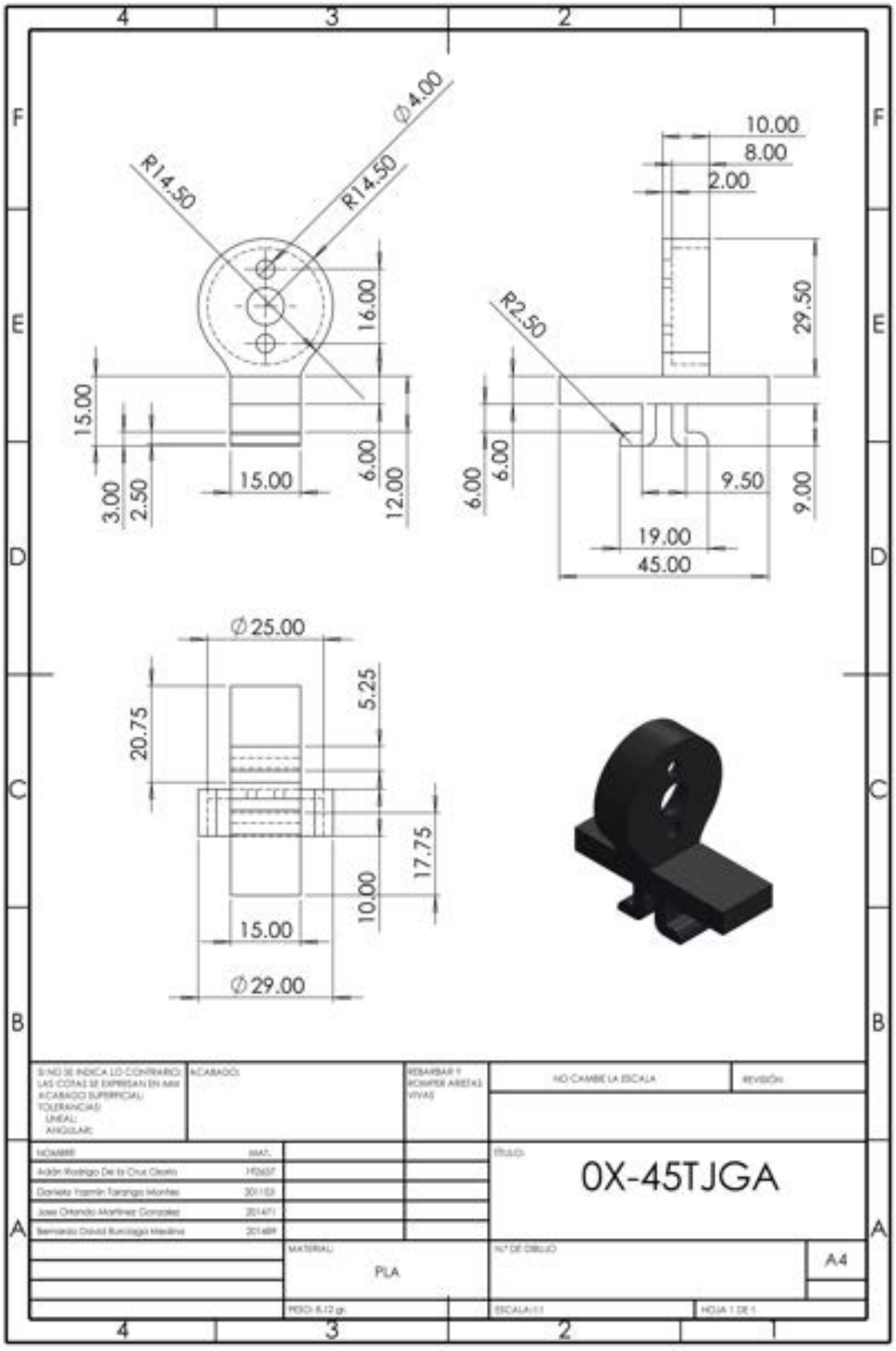


Figura 10: **0X-45TJGA**: Soporte para conectar el motor al perfil de aluminio.

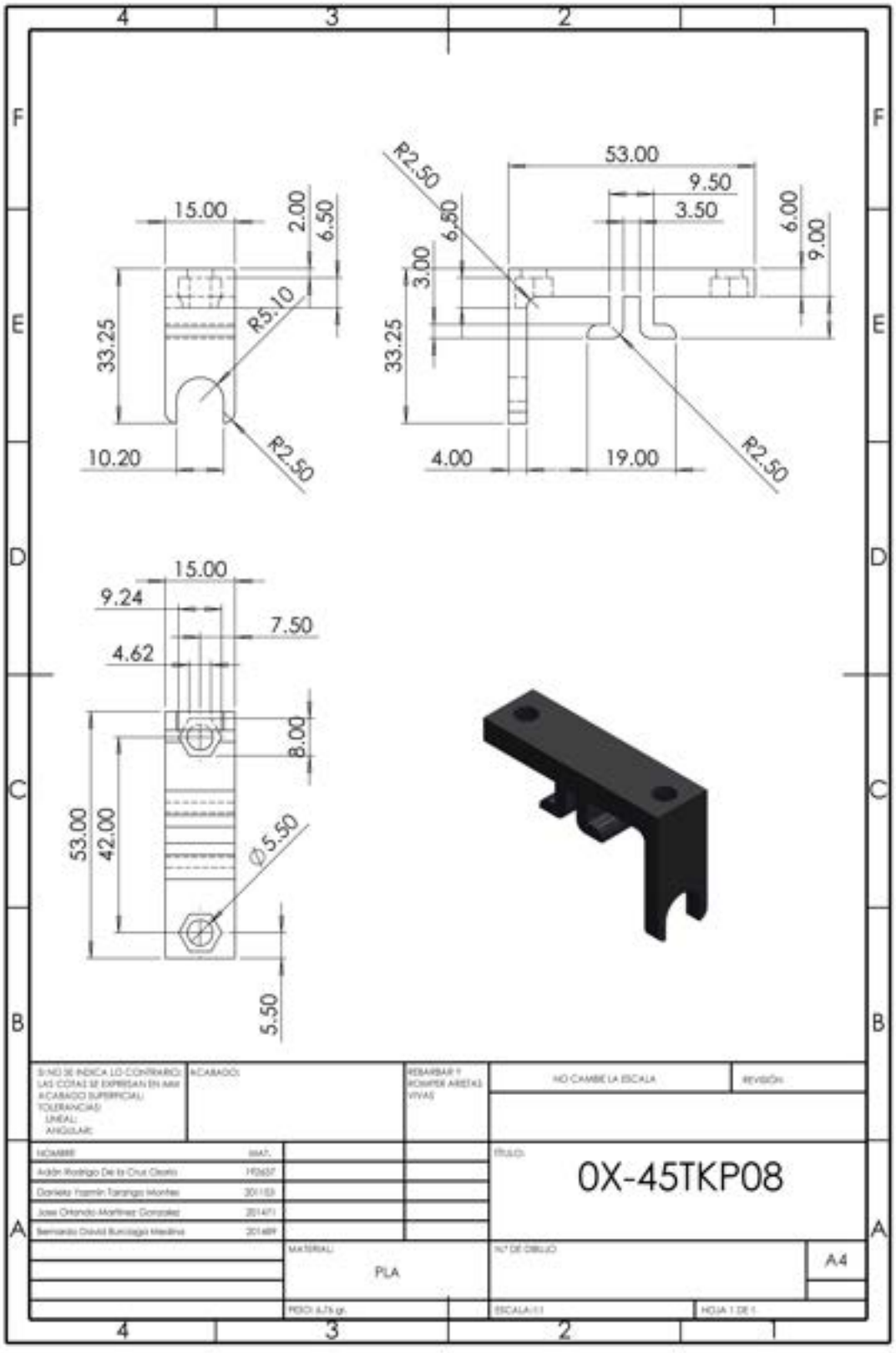


Figura 11: **0X-45TKP08**: Soporte para conectar la chumacera KP08 al perfil de aluminio.

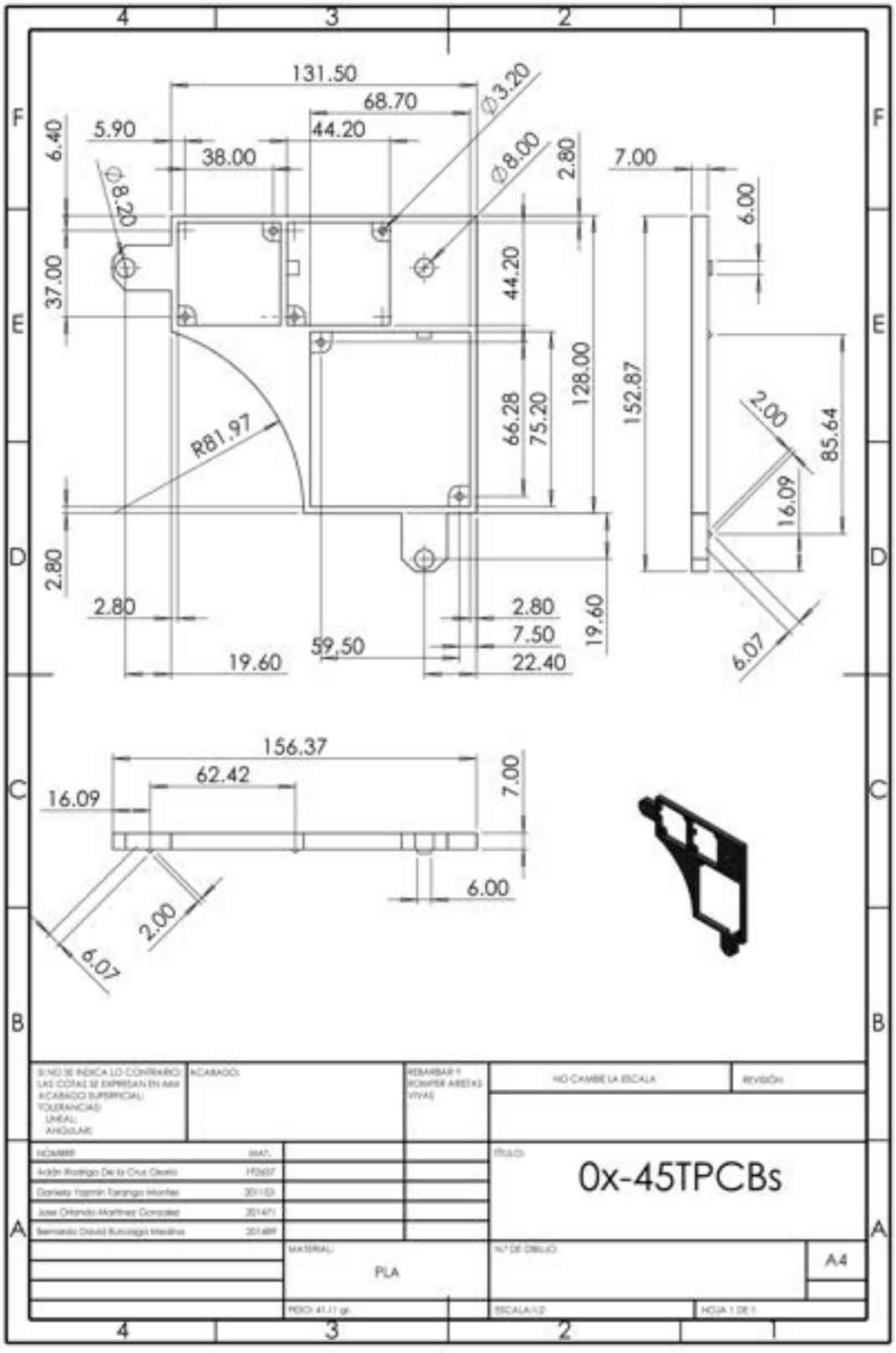


Figura 12: **0X-45TPCBs**: Soporte para conectar las PCBs al perfil de aluminio.

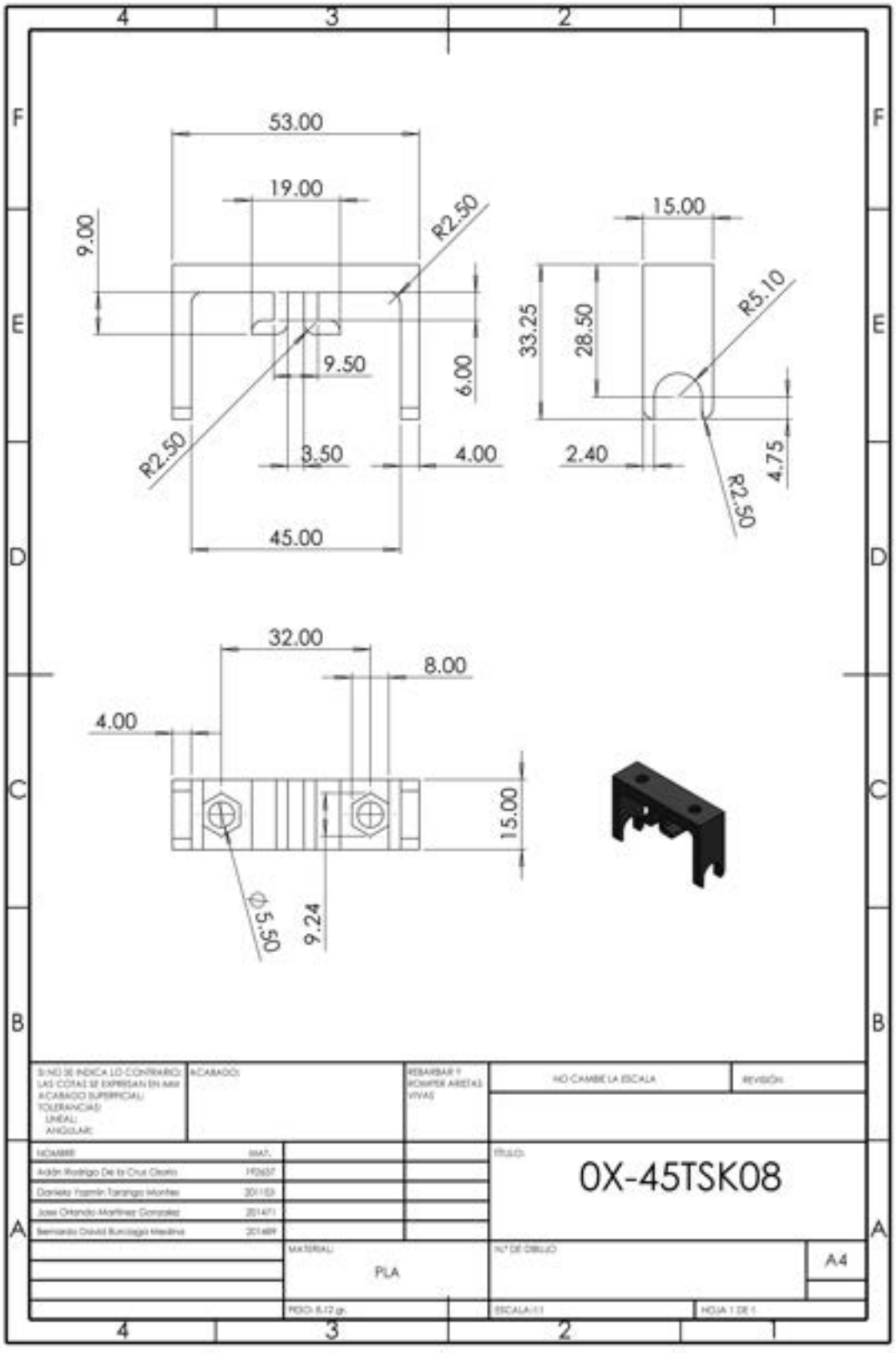


Figura 13: **0X-45TSK08**: Soporte para conectar el soport **SK8** al perfil de aluminio.

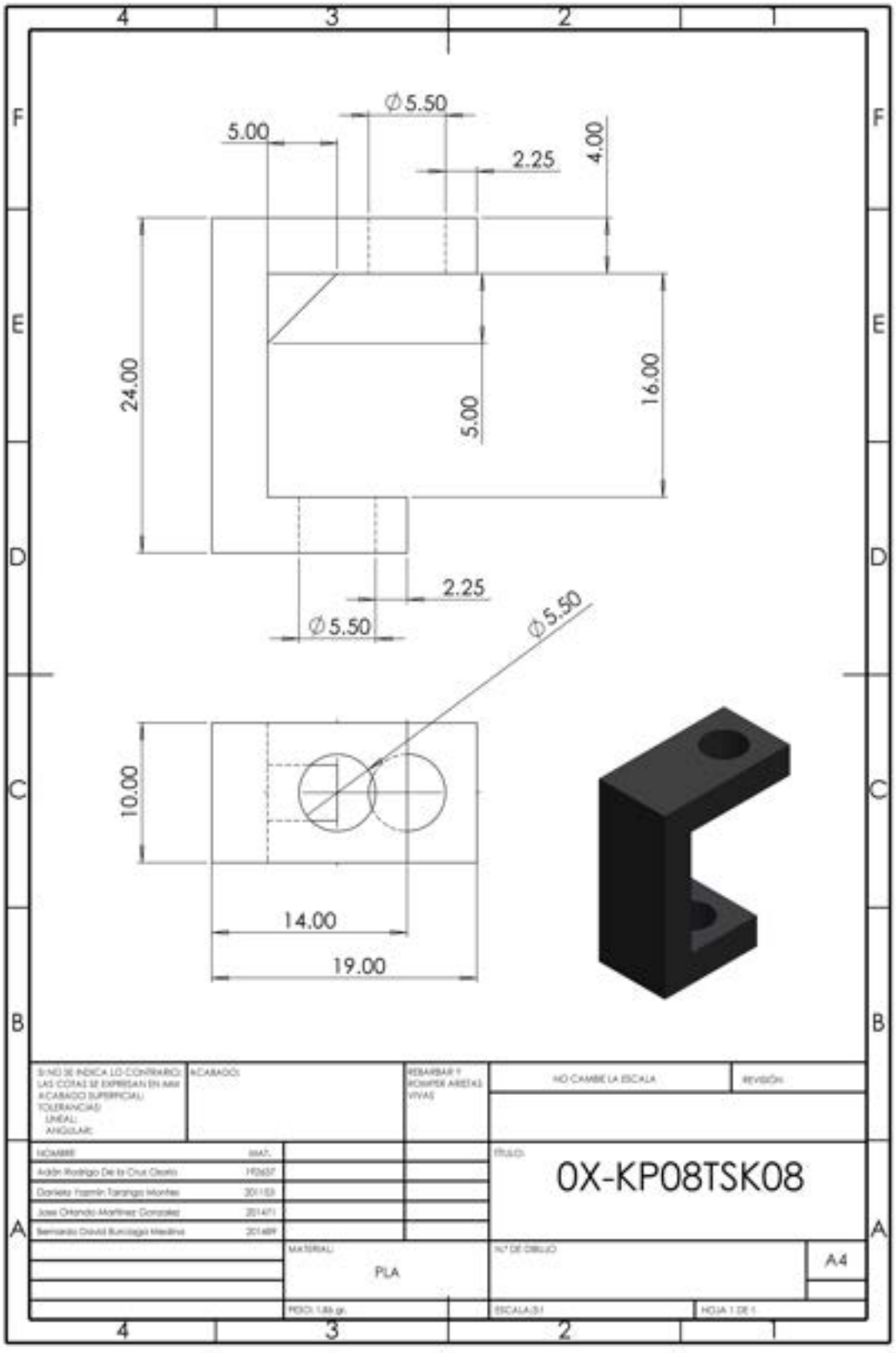


Figura 14: **0X-KP08TSK08**: Soporte para conectar la chumacera KP08 al soporte **SK8**.

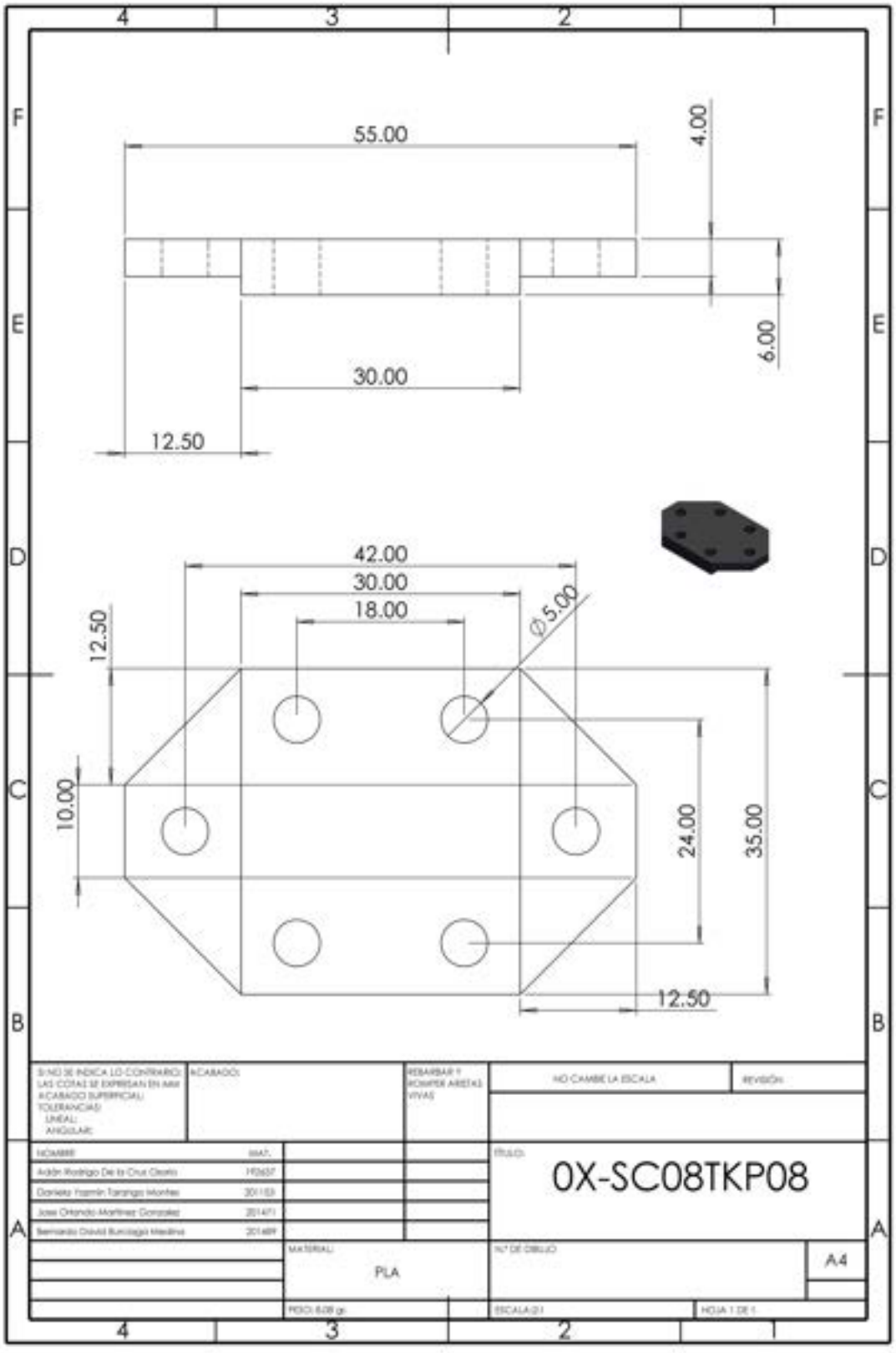


Figura 15: **0X-SC08TKP08**: Soporte para conectar el rodamiento **SC08UU** al soporte **KP08**.

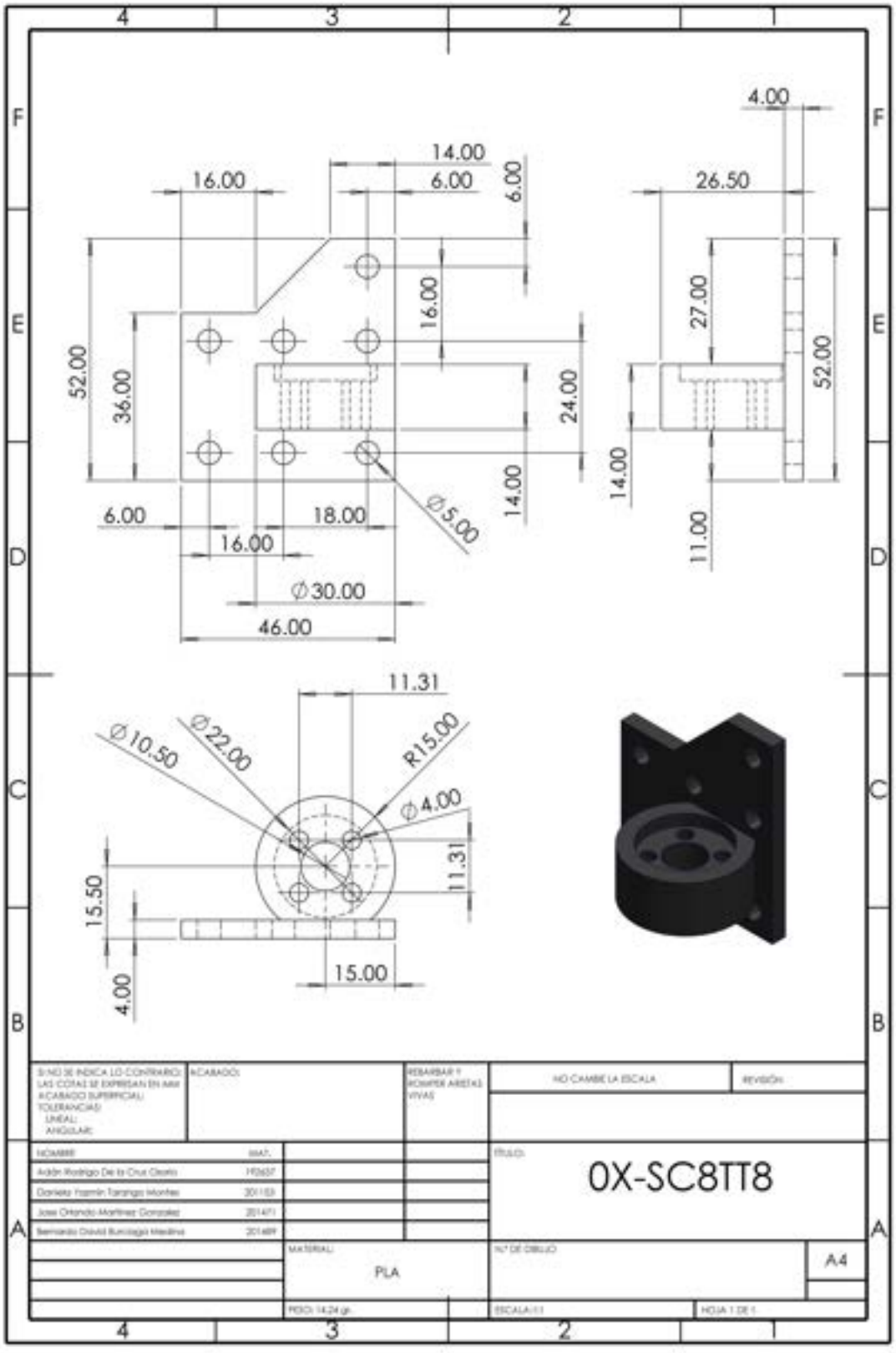


Figura 16: **0X-SC08TT8**: Soporte para conectar el rodamiento **SC08UU** a la tuerca Acme T8.

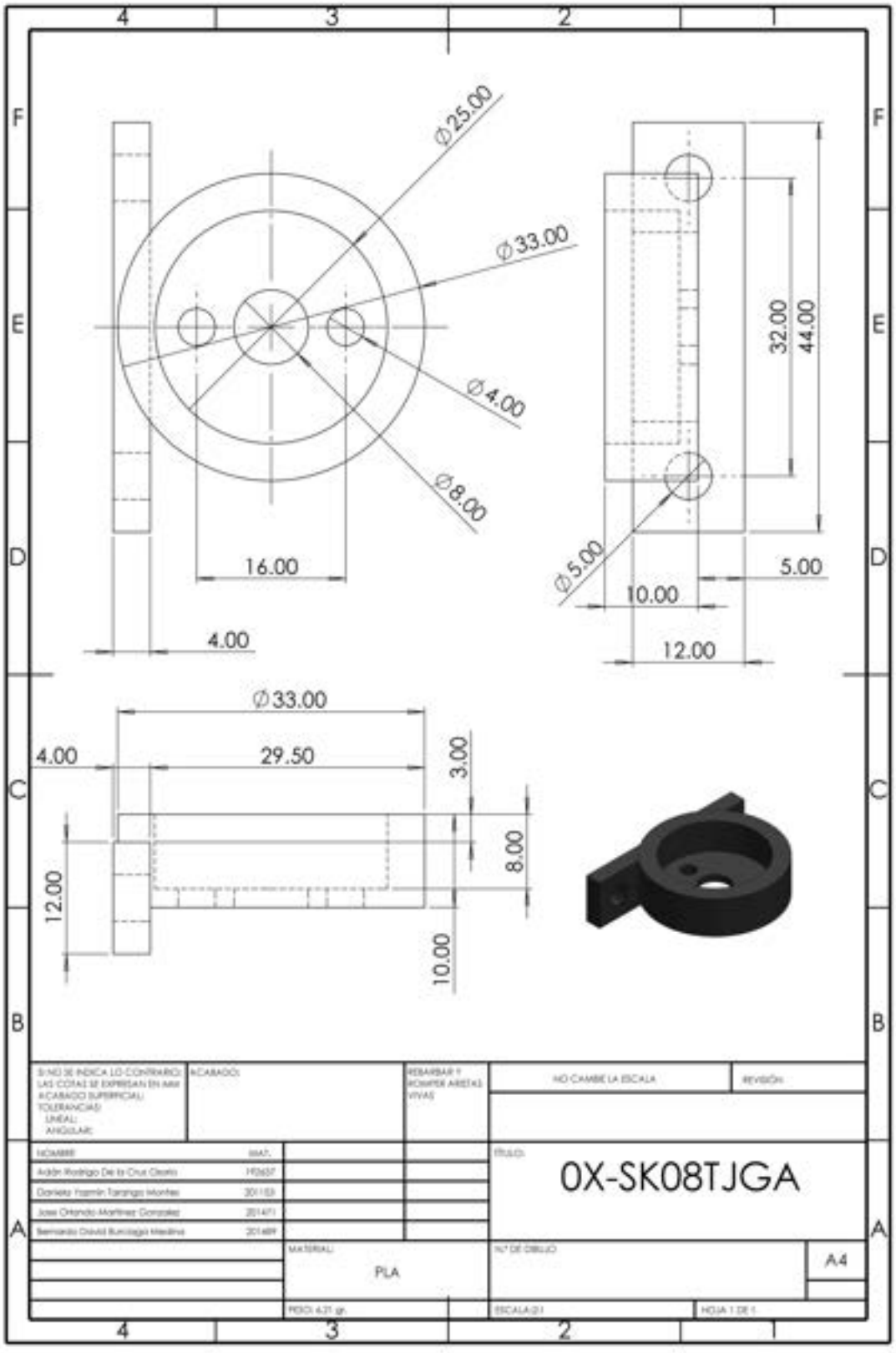


Figura 17: **0X-SK08TJGA**: Soporte para conectar el motor al soporte **SK8**.

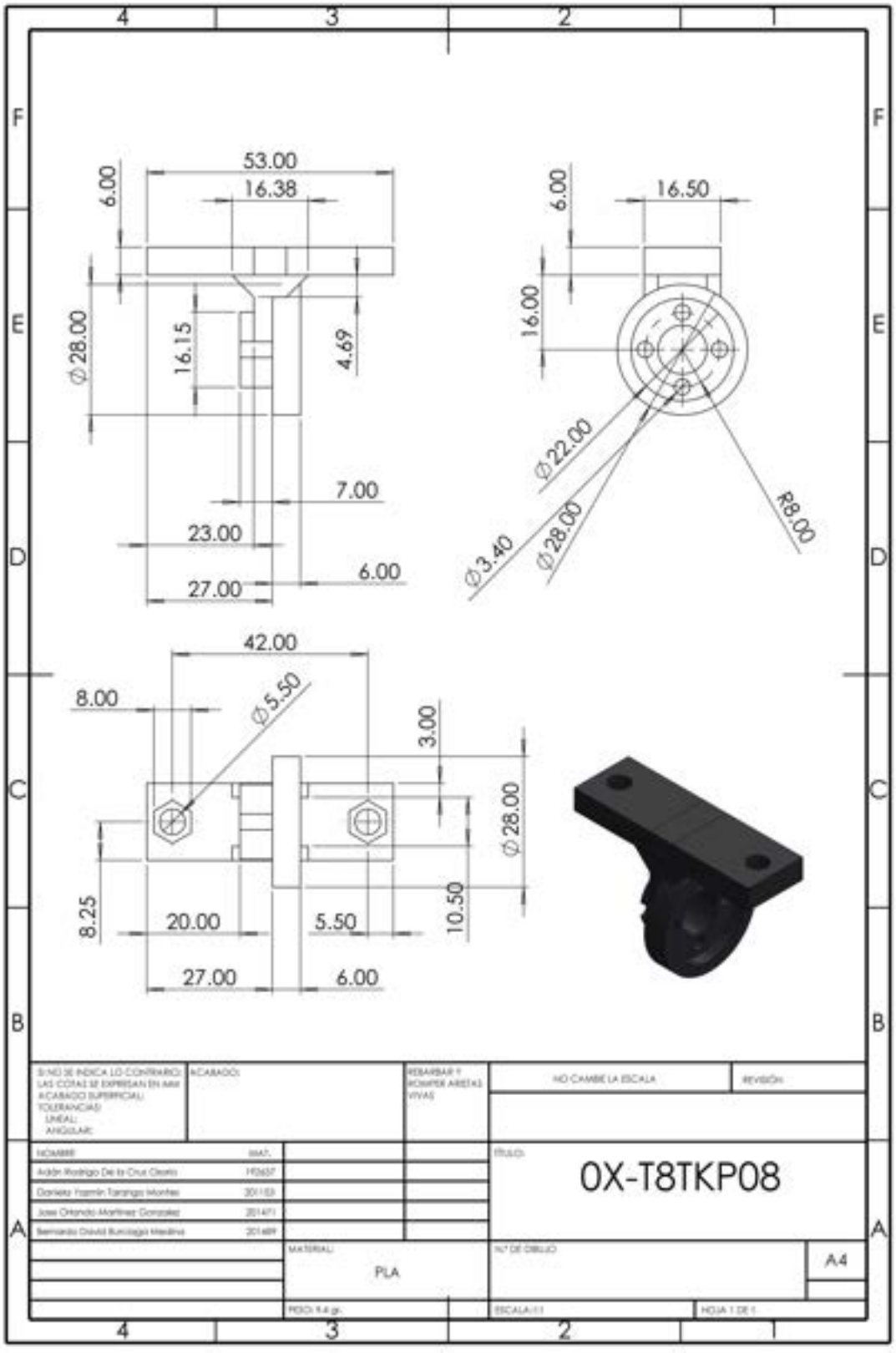


Figura 18: **0X-T8TKP08**: Soporte para conectar la tuerca Acme T8 a la chumacera KP08.

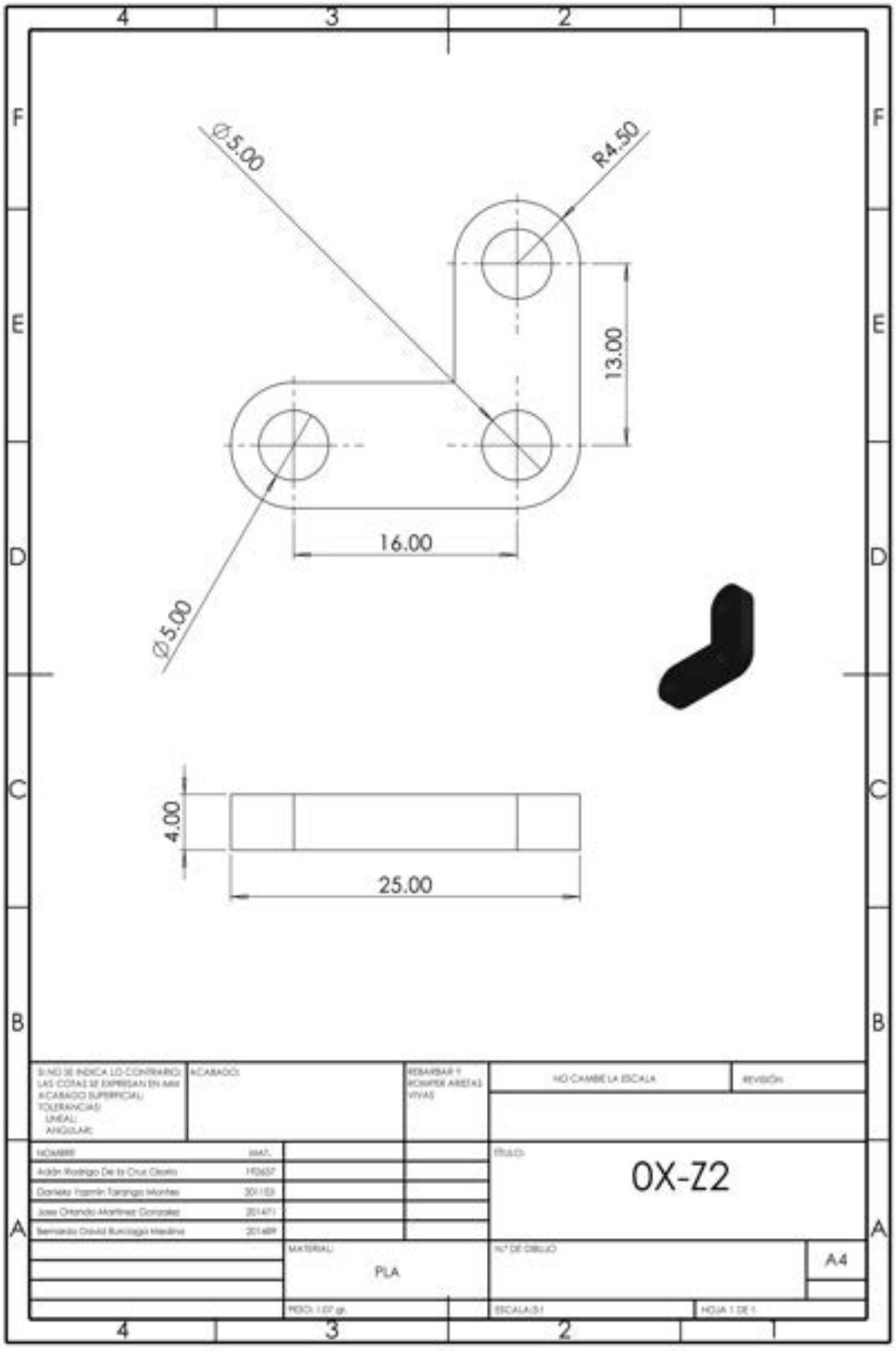


Figura 19: **0X-Z2**: Conector de piezas **0X-SC8TT8**.

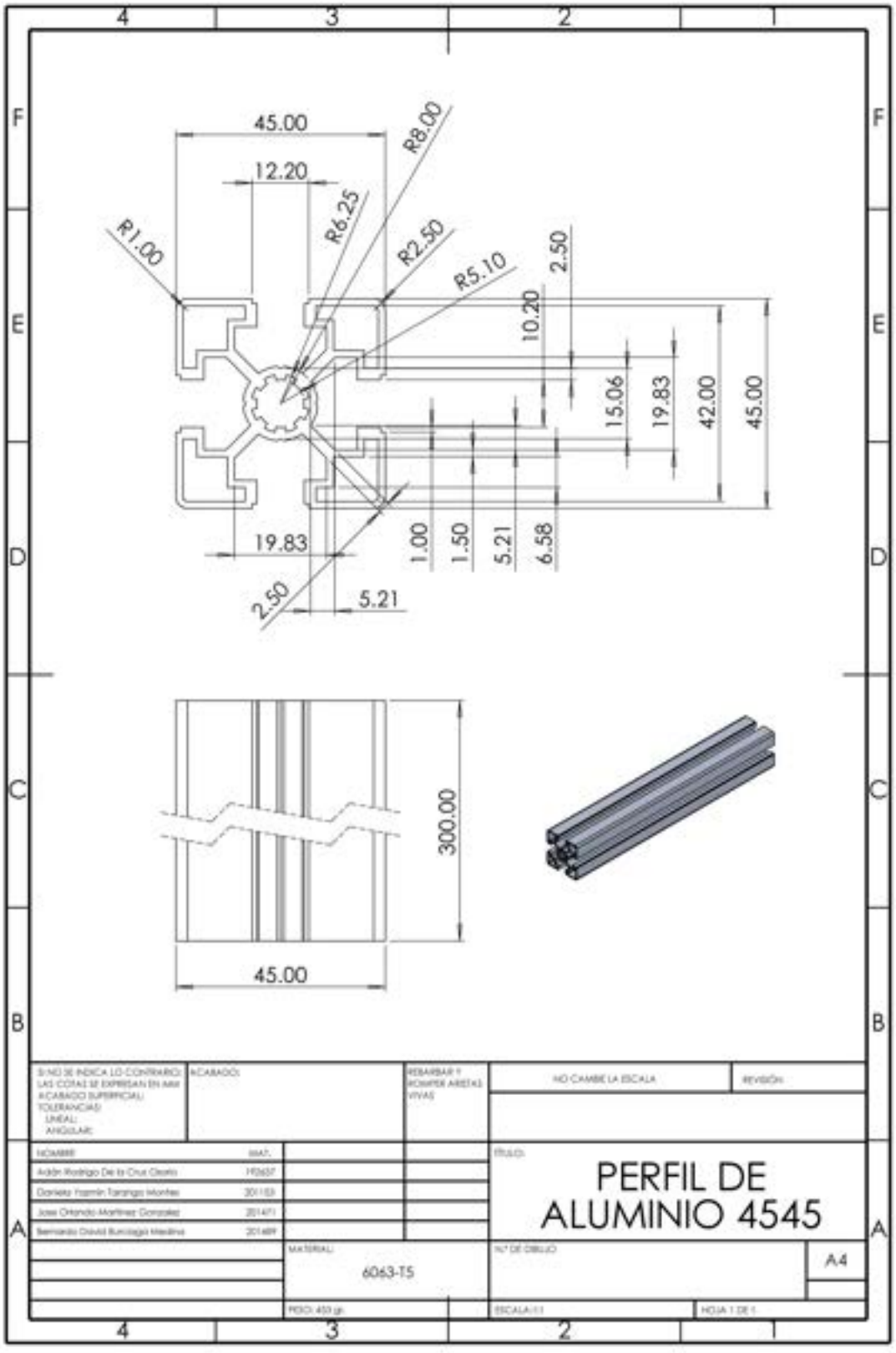


Figura 20: **4545**: Perfil de aluminio extruido de 45x45x300 mm.

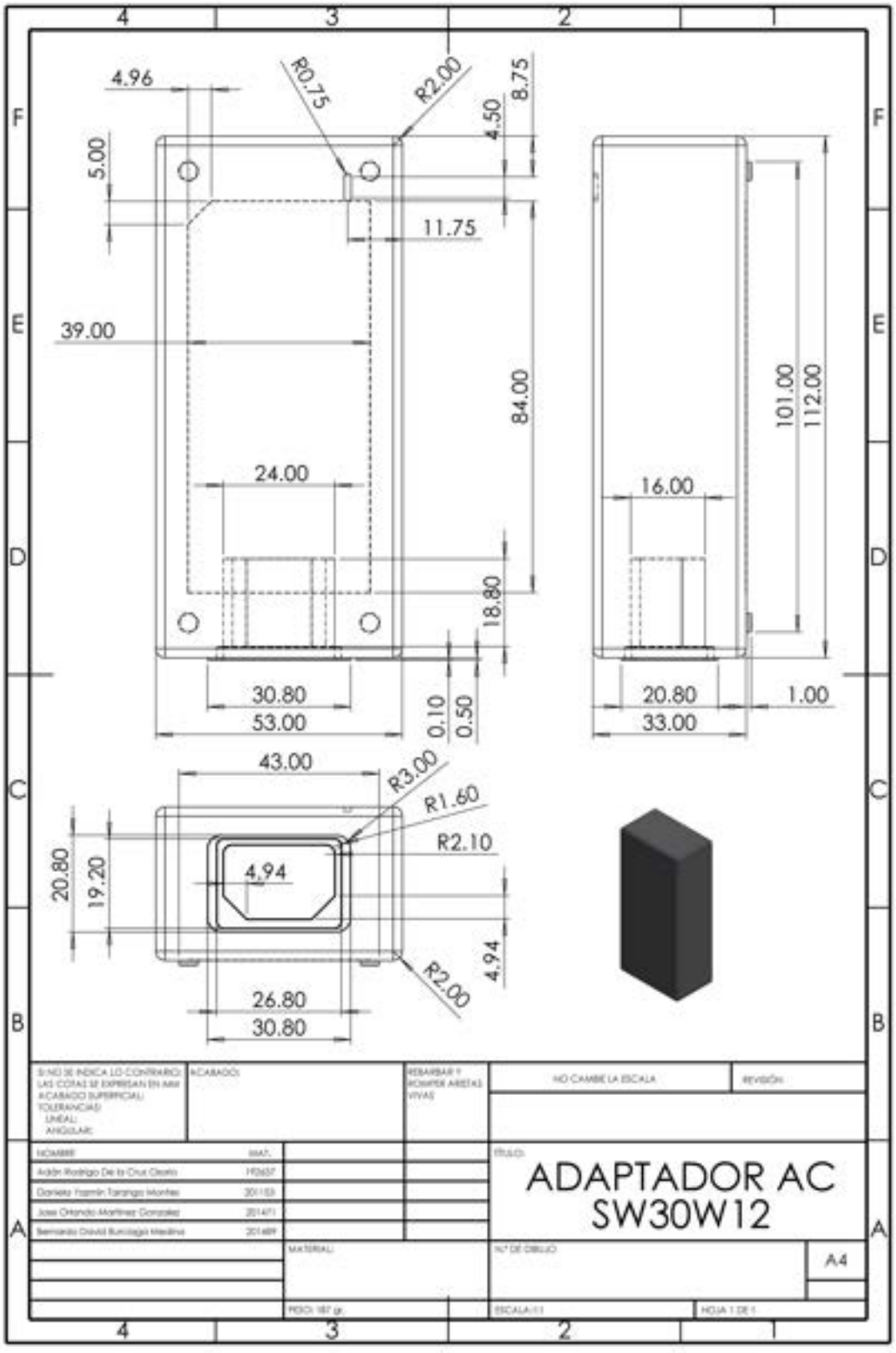


Figura 21: **AC Adapter**: Adaptador de corriente AC **SW30W12**.

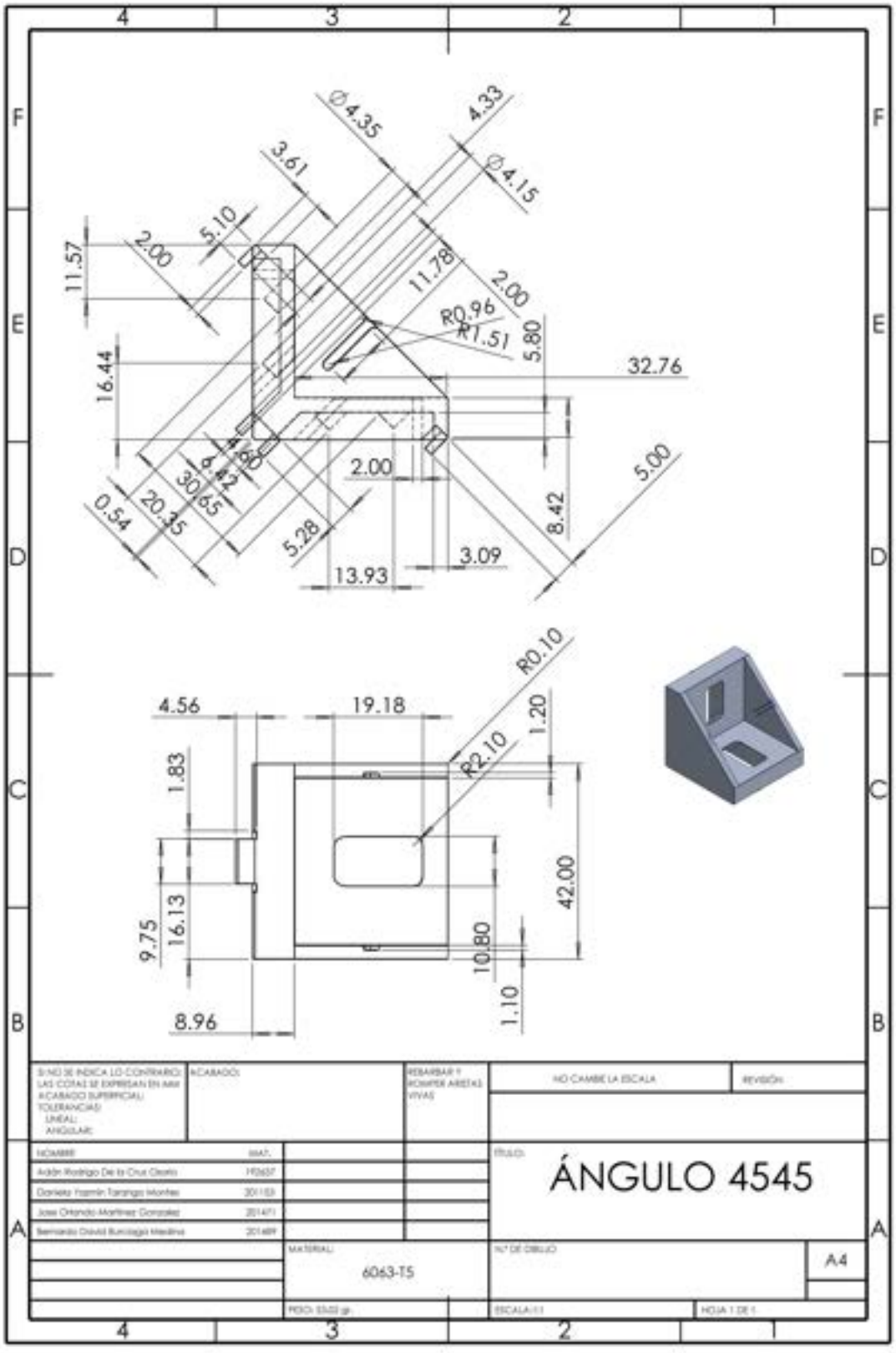


Figura 22: **angulo 4545**: Angulo de 45x45 mm para conectar el perfil de aluminio.

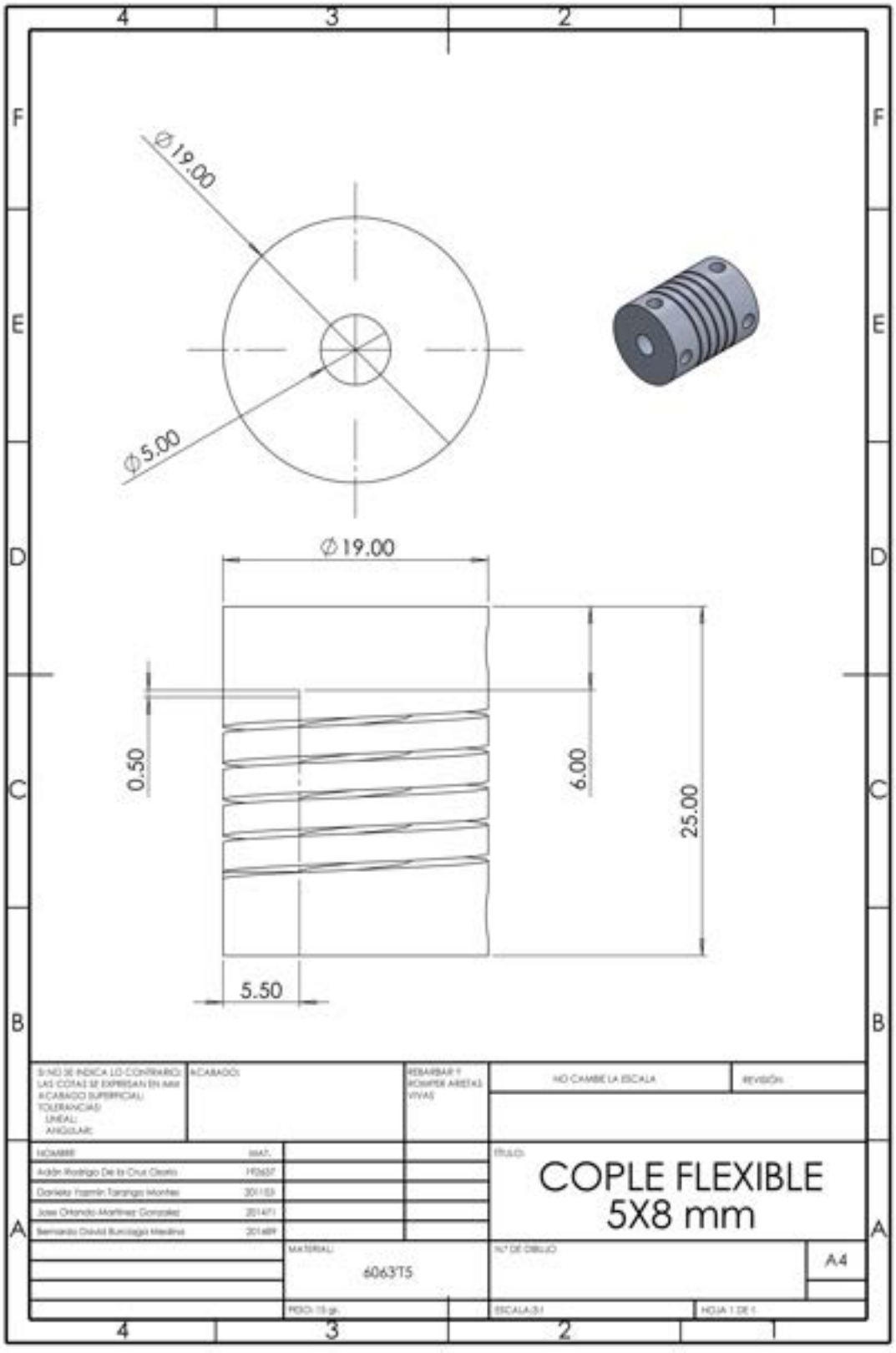


Figura 23: **Cople 5x8**: Cople flexible de 5x8x25 mm.

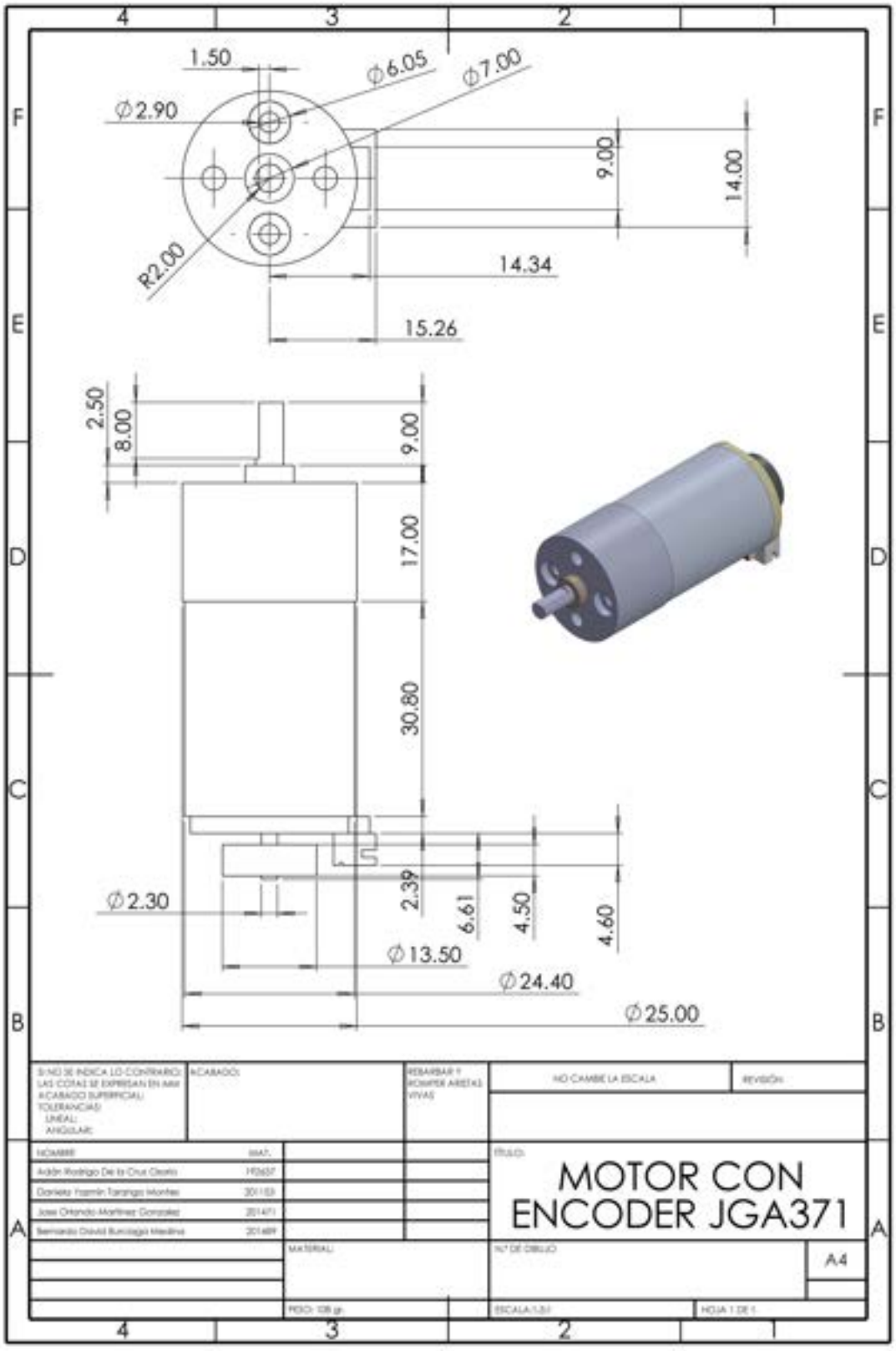


Figura 24: **Motor**: Motor con encoder JGA25-371.

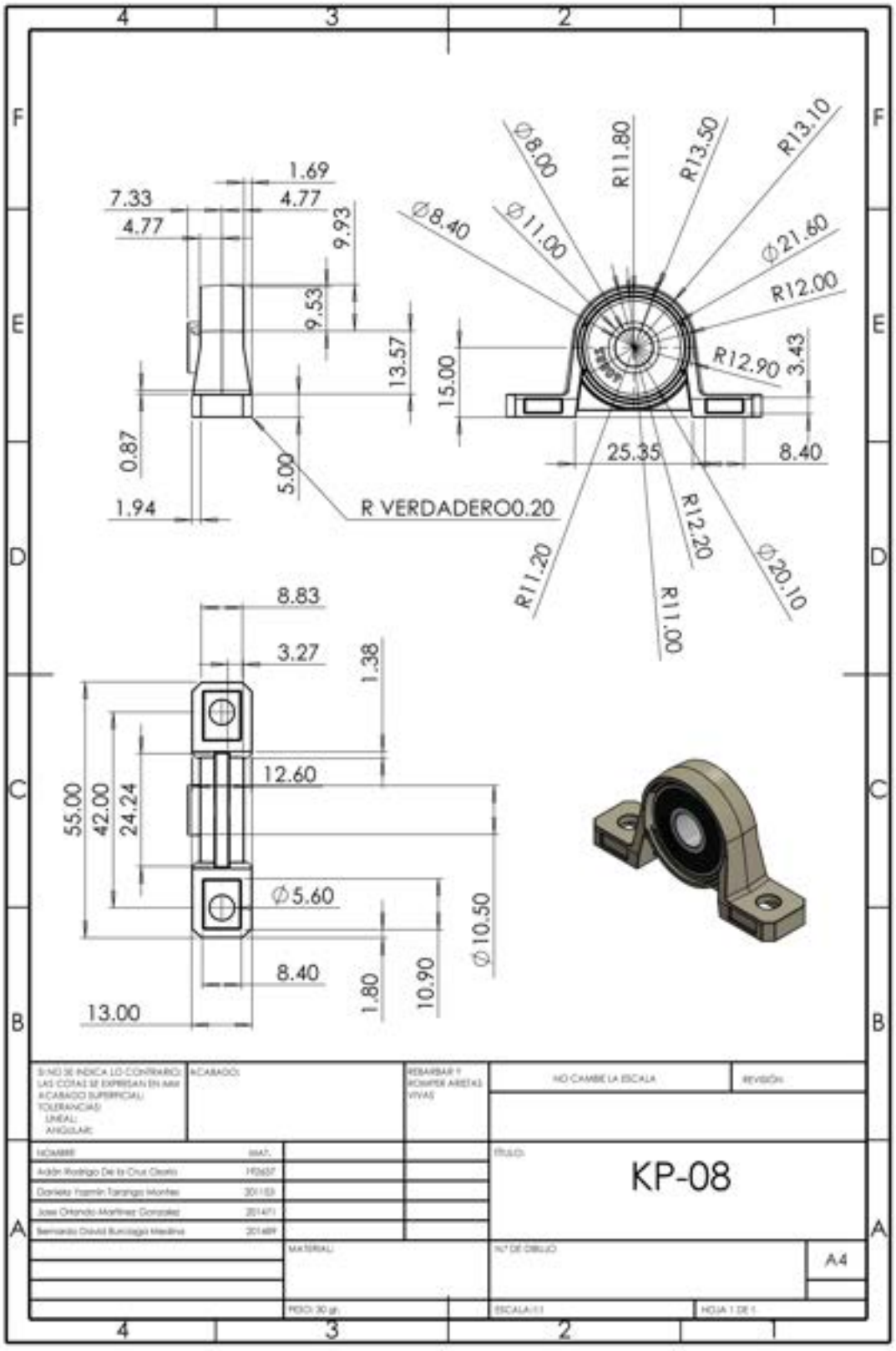


Figura 25: **KP08**: Chumacera KP08.

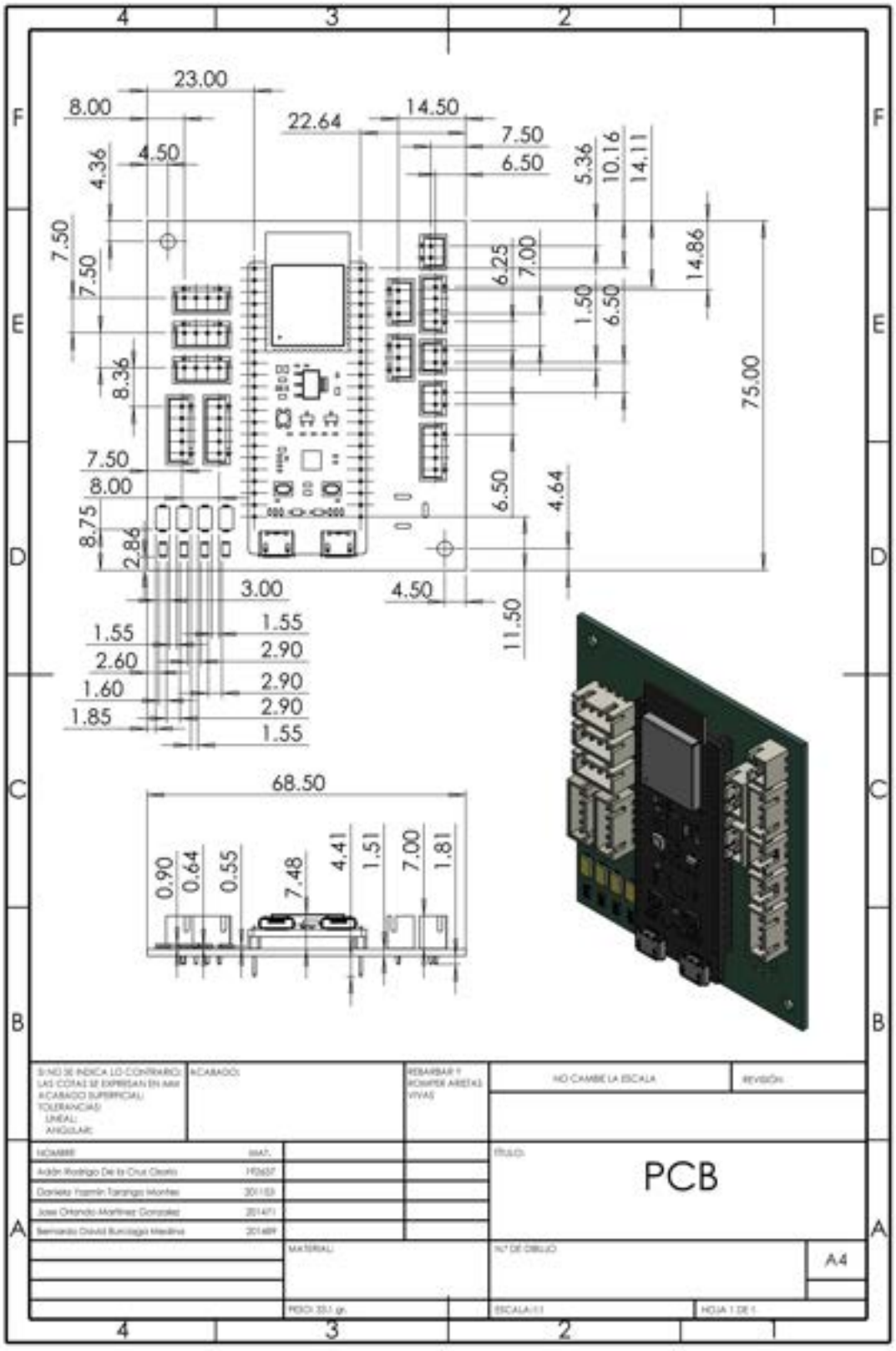


Figura 26: **PCB**: PCB generada en KiCad.

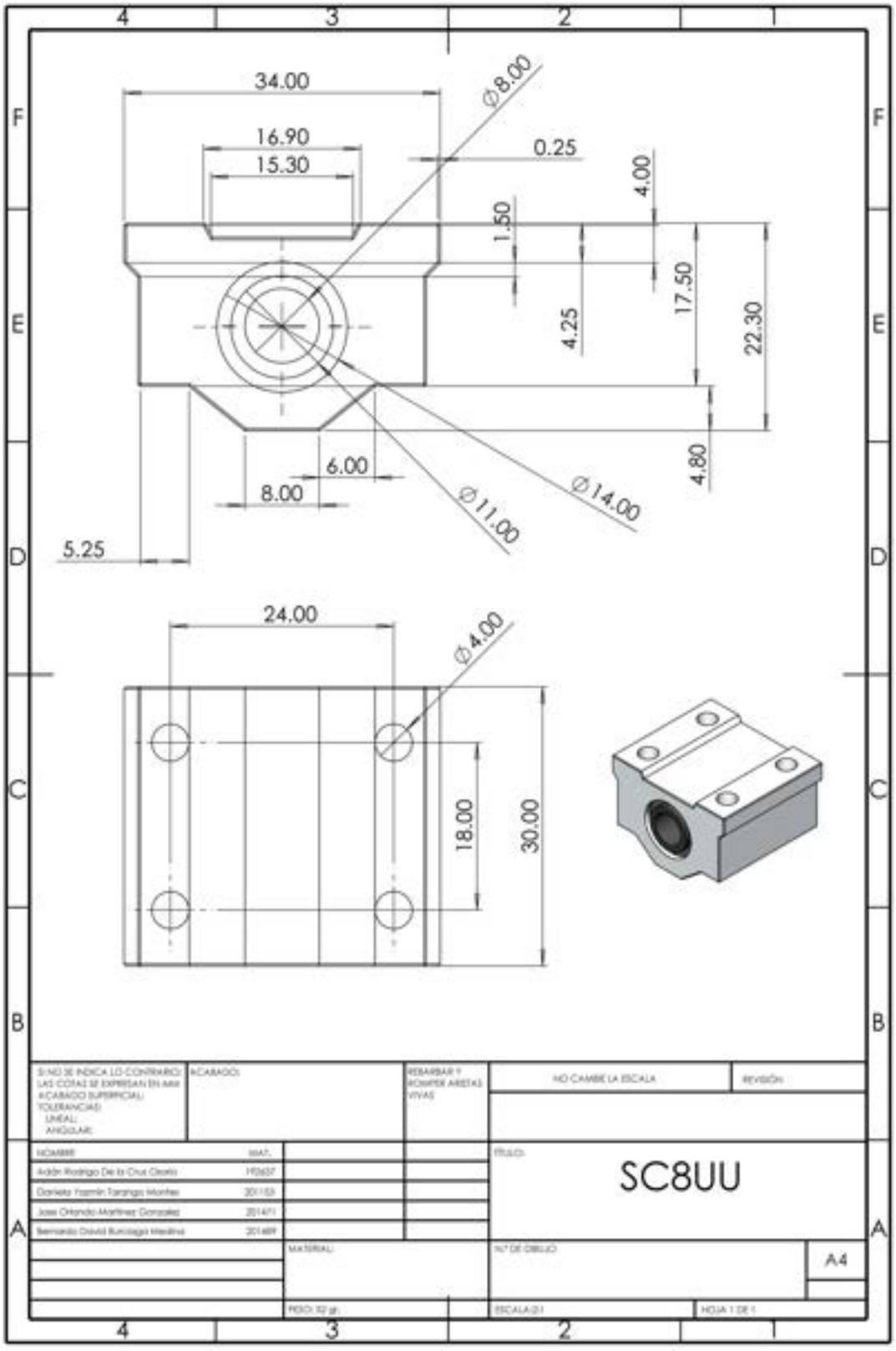


Figura 27: **SC8UU**: Rodamiento SC8UU.

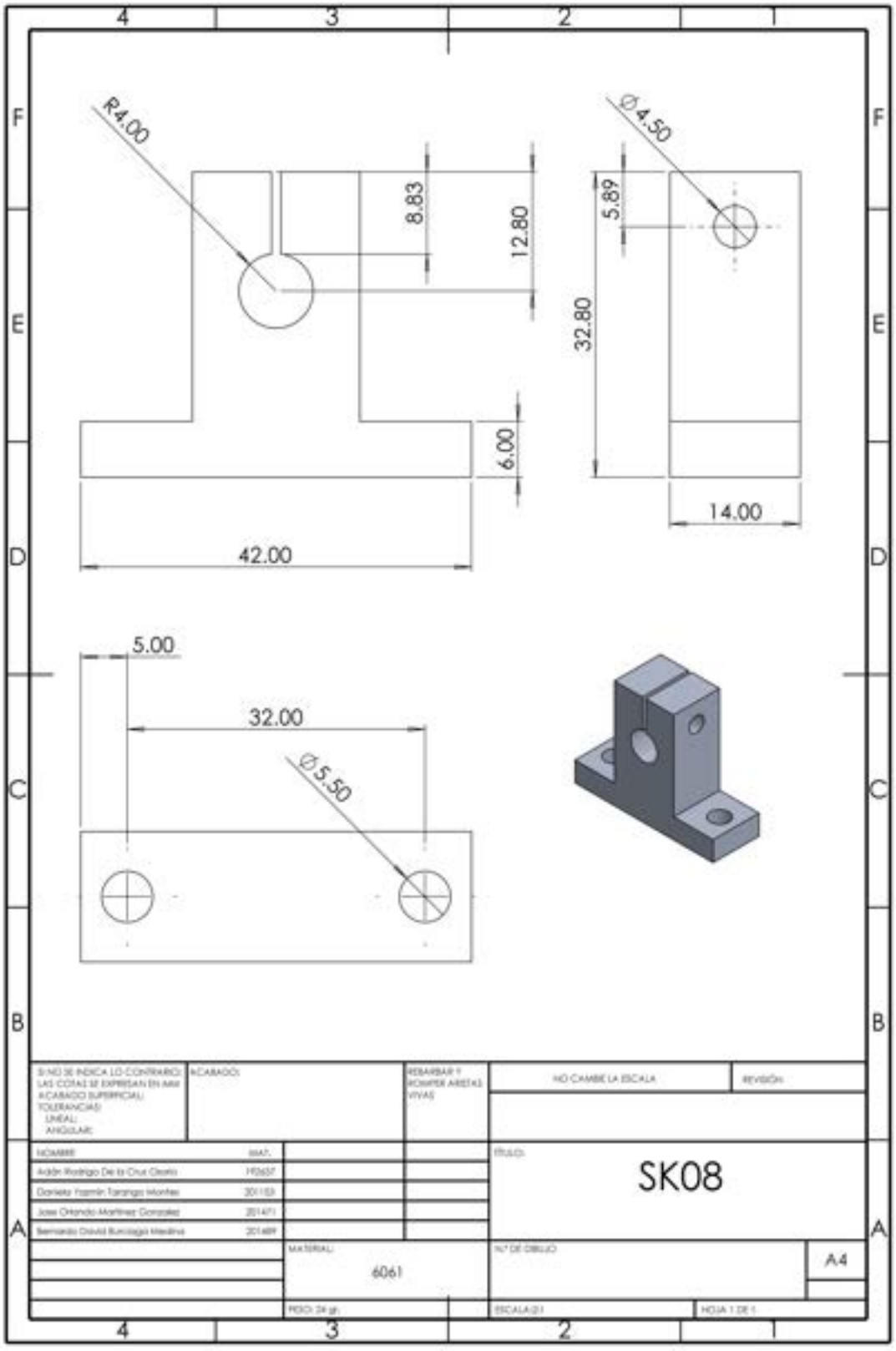


Figura 28: **SK8**: Soporte SK8.

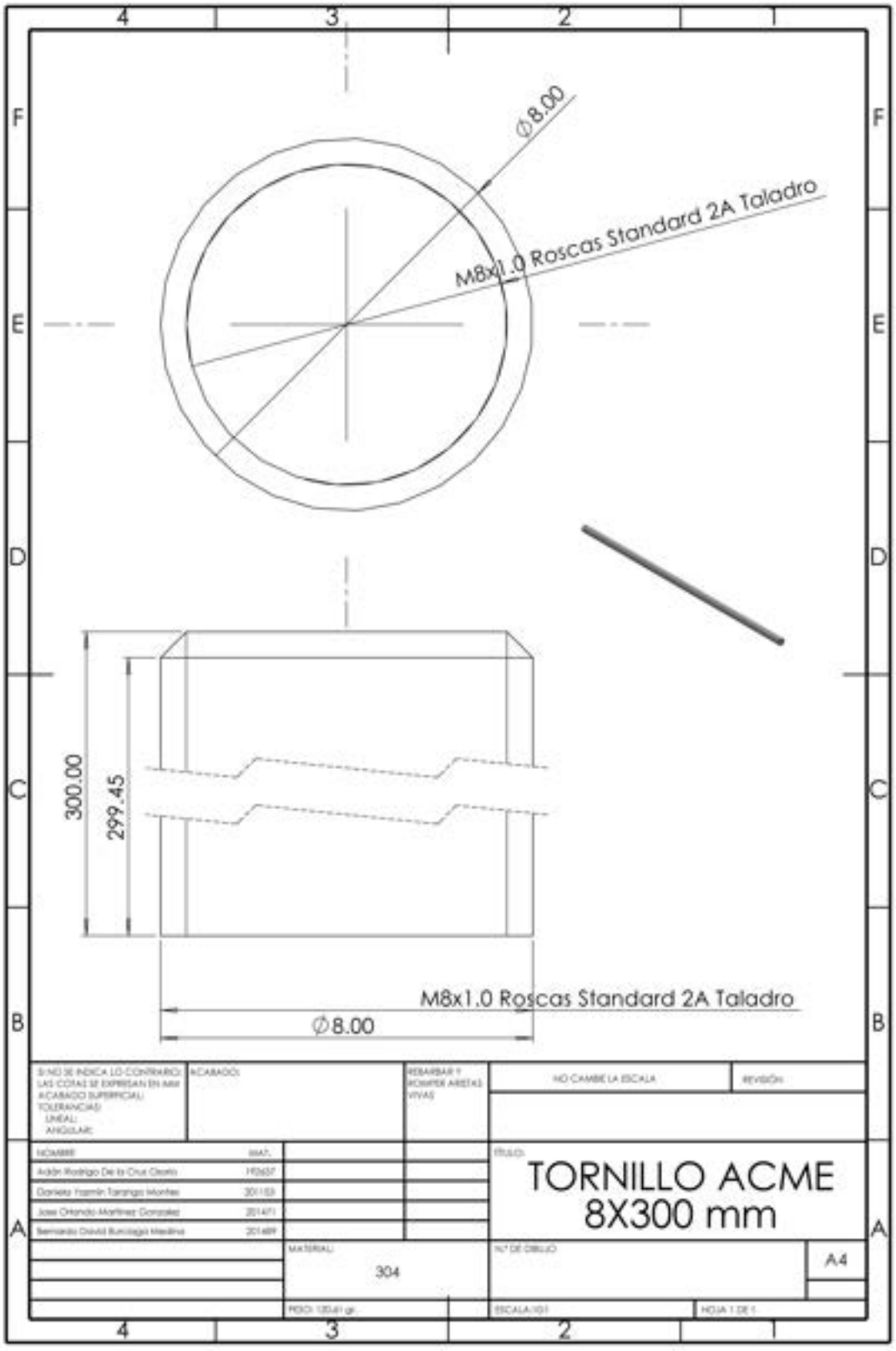


Figura 29: **TSF08**: Tornillo sin fin de 8x300 mm.

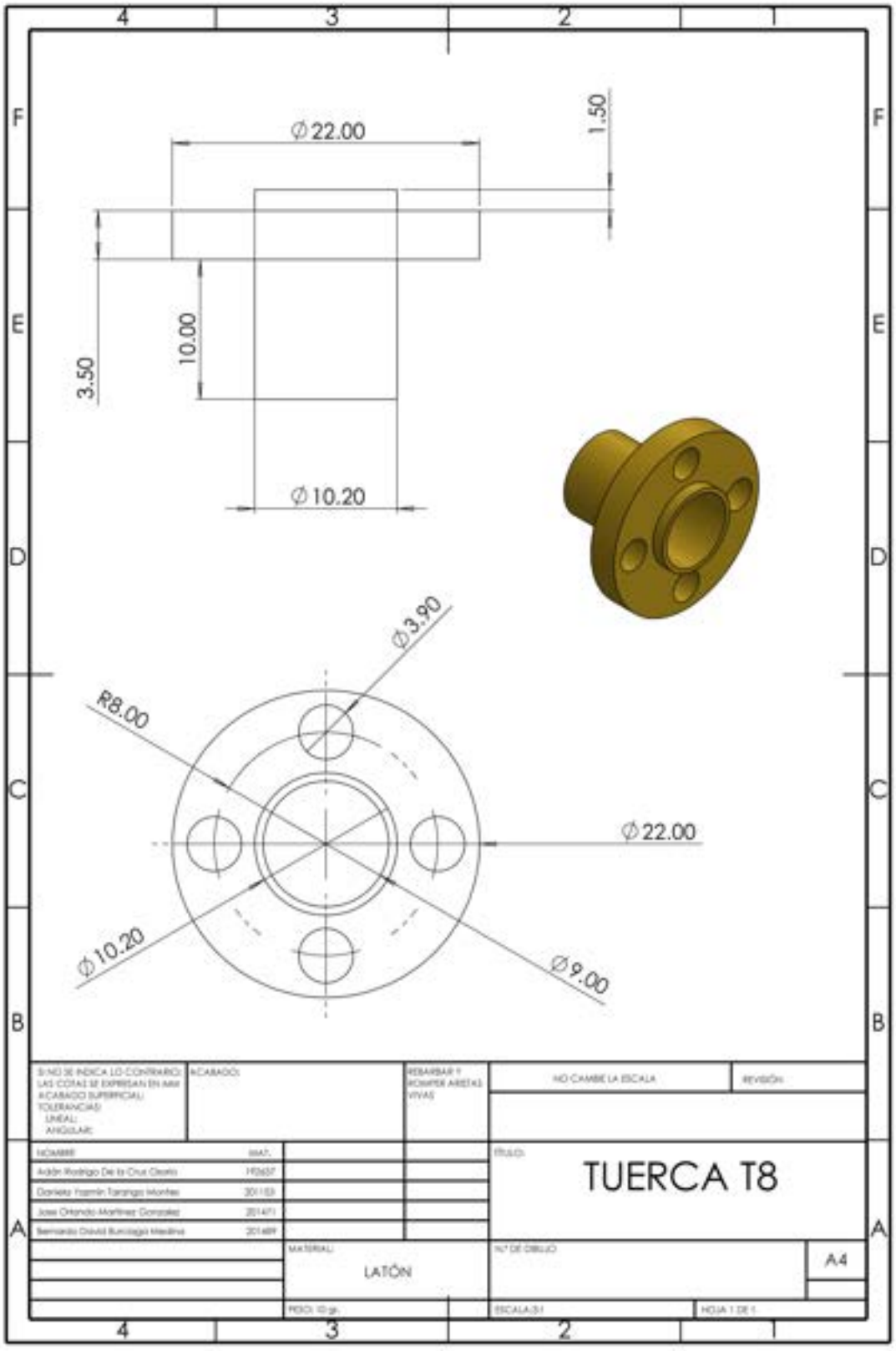


Figura 30: **T8**: Tuerca Acme T8.

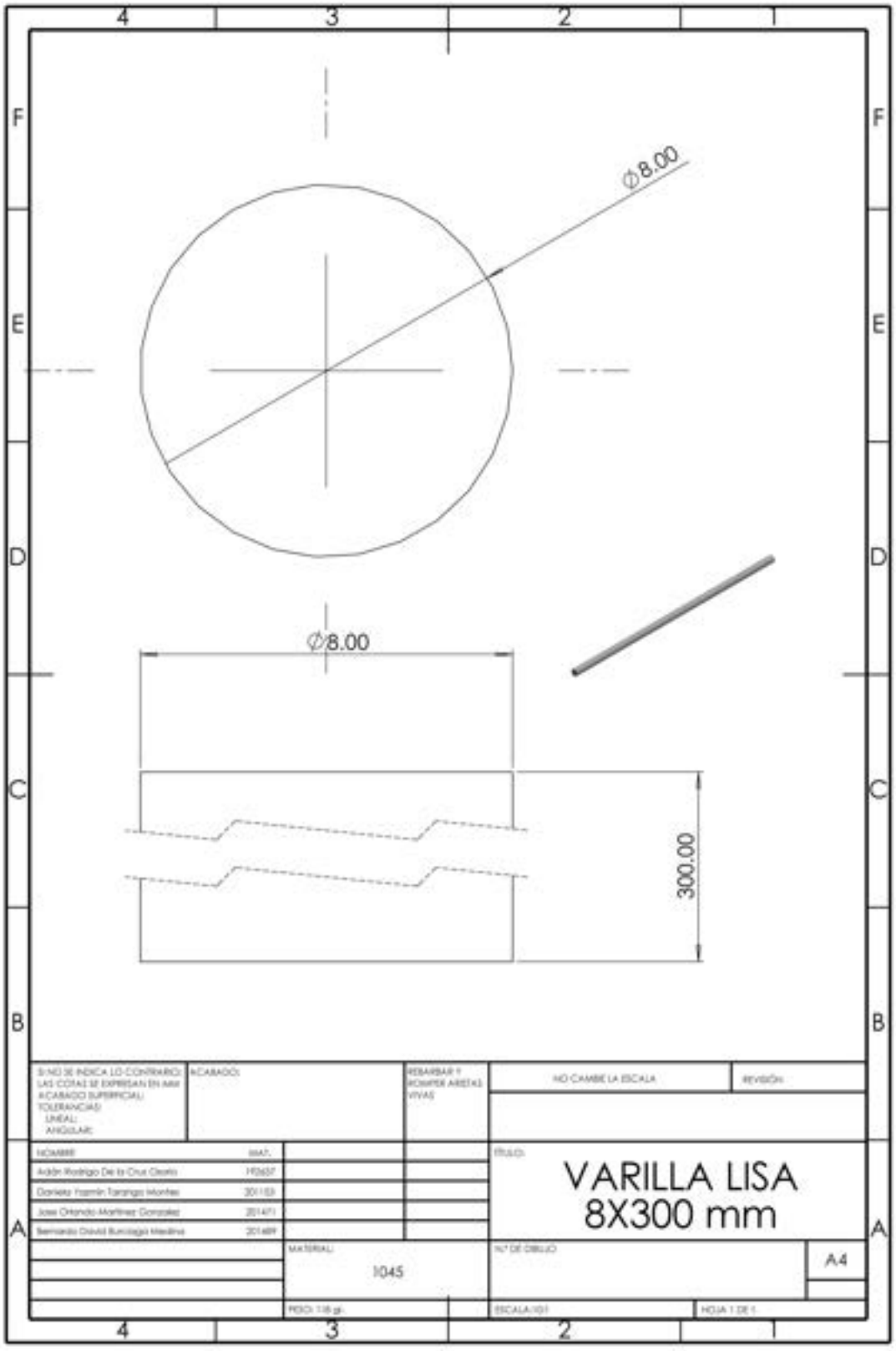


Figura 31: **L8**: Varilla lisa de 8x300 mm.

D. Anexo 4: Red de tópicos

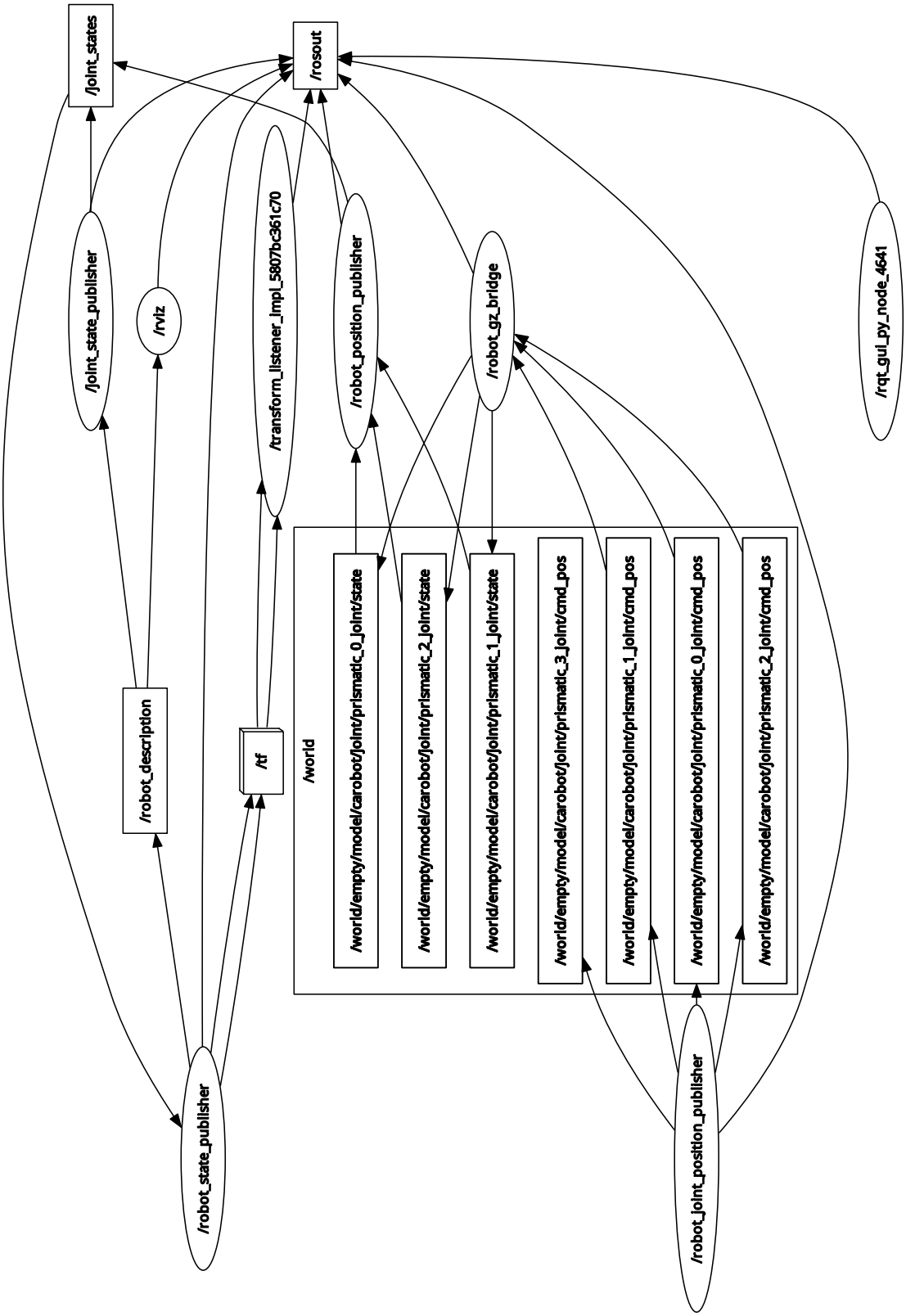


Figura 32: Red de tópicos del robot cartesiano.

E. Anexo 5: Script de instalación de ROS2, Gazebo y dependencias

```
#!/bin/bash
clear
SYS_LANG="./src/lang/${LANG:0:2}.sh"
if [ ! -f "$SYS_LANG" ]; then
  source ./src/lang/en.sh
else
  source "$SYS_LANG"
fi

DIST="$(. /etc/os-release && echo $PRETTY_NAME)"
printf "${MSG_TITLE}$(echo $DIST | awk -F ' ' '{print $1}')\n"

# for ubuntu distros
if [ "$(echo $DIST | grep -E 'Ubuntu|Debian|Linux Mint')" ]; then
  printf "${MSG_PKG_PRE}"
  sudo apt install software-properties-common lsb-release gnupg curl -y > /dev/null 2>&1
  printf "${MSG_TAB_PRE}"

  printf "${MSG_PKG_UNIV}"
  sudo add-apt-repository universe -y > /dev/null 2>&1
  printf "${MSG_TAB_UNIV}"

  printf "${MSG_PKG_KEYS}"
  sudo apt update > /dev/null 2>&1
  sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
      /usr/share/keyrings/ros-archive-keyring.gpg > /dev/null 2>&1
  sudo curl https://packages.osrfoundation.org/gazebo.gpg --output
      /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg > /dev/null 2>&1
  printf "${MSG_TAB_KEYS}"

  printf "${MSG_PKG_ADD}"
  echo "deb [arch=$(dpkg --print-architecture)
      signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
      http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME)
```

```
    main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null 2>&1
echo "deb [arch=$(dpkg --print-architecture)
    signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg]
    http://packages.osrfoundation.org/gazebo/ubuntu-stable $(lsb_release -cs) main" |
    sudo tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null 2>&1
printf "${MSG_TAB_ADD}"

printf "${MSG_PKG_UPD}"
sudo apt update > /dev/null 2>&1
printf "${MSG_TAB_UPD}"

printf "\n"
ROS_DISTROS=("beta3" "humble" "jazzy" "rolling" "galactic" "foxy" "dashing" "crystal"
    "ardent" "bouncy" "aero" "zesty" "xenial")
ROS_DIST="alpha1"
PKG_COUNT=50
for i in "${ROS_DISTROS[@]}"; do
  printf "${MSG_ROS_FIND} $i ${MSG_TAB_ROS}"
  ROS_CHECK="$(apt search ros-$i > /dev/null 2>&1 | wc -l)"
  if [ "$ROS_CHECK" -gt "$PKG_COUNT" ]; then
    ROS_DIST=$i
    break
  fi
done

printf "\n"
while IFS= read -r line; do
  if [[ "$line" == *"ros-"* ]]; then
    line=${line//ROS_DIST/$ROS_DIST}
    printf "${MSG_PKG_INSTALL} $line\n"
    sudo apt install -y $line > /dev/null 2>&1
  fi
done < ./src/deb-packages.txt

printf "\033[F\033[K\033[F\033[K"
printf "${MSG_ROS_INSTALL} $ROS_DIST"
```

```
ROS_SOURCE="/opt/ros/$ROS_DIST/setup.bash"
if [ -f "$ROS_SOURCE" ]; then
  printf "${MSG_TAB_ROS_OK}"
else
  printf "${MSG_TAB_ROS_FAIL}"
  exit 1
fi

if [[ "$ROS_DIST" == "rolling" ]]; then
  gz_print="ionic   "
  gz_install="gz-ionic"
elif [[ "$ROS_DIST" == "jazzy" ]]; then
  gz_print="harmonic"
  gz_install="gz-harmonic"
elif [[ "$ROS_DIST" == "humble" ]]; then
  gz_print="fortress"
  gz_install="ignition-fortress"
else
  gz_print="citadel "
  gz_install="ignition-citadel"
fi
printf "${MSG_GZ_INSTALL}${gz_print}${MSG_TAB_GZ}"
sudo apt install $gz_install -y > /dev/null 2>&1
printf "${MSG_GZ_INSTALLED}${gz_print}${MSG_TAB_GZ_OK}"

SSHELL="$(echo $SHELL | awk -F '/' '{print $NF}')"
ROS_DISTRO="$ROS_DIST"

source /opt/ros/$ROS_DISTRO/setup.$SSHELL

printf "${MSG_UROS_DOWNLOAD}"
rm -rf ~/uros-ws
mkdir -p ~/uros-ws
cd ~/uros-ws
git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git
    src/micro_ros_setup > /dev/null 2>&1
```

```
printf "${MSG_TAB_UROSDOWN}"

printf "${MSG_UROS_INSTALL}"
sudo apt update > /dev/null 2>&1 && rosdep update > /dev/null 2>&1
rosdep install --from-paths src --ignore-src -y > /dev/null 2>&1
colcon build > /dev/null 2>&1
source install/local_setup.$SSHELL
printf "${MSG_TAB_UROSINS}"

printf "${MSG_UROS_AGENT_CR}"
ros2 run micro_ros_setup create_agent_ws.sh > /dev/null 2>&1
printf "${MSG_TAB_UROSCR}"

printf "${MSG_UROS_AGENT_BLD}"
ros2 run micro_ros_setup build_agent.sh > /dev/null 2>&1
source install/local_setup.$SSHELL
printf "${MSG_TAB_UROSBLD}"

printf "${MSG_UROS_INO}"
rm -rf ~/Arduino/libraries/micro_ros_arduino
git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_arduino.git
    ~/Arduino/libraries/micro_ros_arduino > /dev/null 2>&1
printf "${MSG_TAB_UROSINO}"

printf "${MSG_FINISH} \e[93m~/.${SSHELL}rc\e[90m:\e[0m\n"
for i in $(seq 1 60); do
  printf "#"
done
printf "\nexport ROS_DOMAIN_ID=42\n"
printf "export ROS_VERSION=2\n"
printf "export ROS_PYTHON_VERSION=3\n"
printf "run-ros() {\n"
AD
printf "  export ROS_DISTRO=$ROS_DIST\n"
printf "  source /opt/ros/$ROS_DISTRO/setup.$SSHELL\n"
printf "  eval \"\$(register-python-argcomplete ros2)\"\n"
```

```
  printf "  eval \"\$(register-python-argcomplete ros2cli)\"\n"
  printf "  eval \"\$(register-python-argcomplete colcon)\"\n"
  printf "}\n"
  printf "run-uros() {\n"
  printf "  export ROS_DISTRO=$ROS_DIST\n"
  printf "  source ~/uros_ws/install/local_setup.$SSHELL\n"
  printf "  eval \"\$(register-python-argcomplete ros2)\"\n"
  printf "  eval \"\$(register-python-argcomplete ros2cli)\"\n"
  printf "  eval \"\$(register-python-argcomplete colcon)\"\n"
  printf "}\n"
  printf "alias ros-agent=\"ros2 run micro_ros_agent micro_ros_agent\"\n"
  for i in $(seq 1 60); do
    printf "#"
  done
  printf "\e[0m\n"
fi
```

Listing 1: Script de instalación

F. Anexo 6: Códigos del paquete carobot

```
cmake_minimum_required(VERSION 3.8)
project(carobot)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

install(
  DIRECTORY launch urdf config rviz
```

```
  DESTINATION share/${PROJECT_NAME}
)

install(PROGRAMS
  src/pos_get.py
  src/pos_put.py
  DESTINATION lib/${PROJECT_NAME}
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Listing 2: Archivo CMakeLists.txt del paquete

```
<?xml version="1.0" encoding="utf-8"?>
<gazebo>
  <plugin
    filename="gz-sim-joint-state-publisher-system"
    name="gz::sim::systems::JointStatePublisher">
    <joint_name>prismatic_0_joint</joint_name>
    <topic>world/empty/model/carobot/joint/prismatic_0_joint/state</topic>
  </plugin>
  <plugin
    filename="gz-sim-joint-state-publisher-system"
    name="gz::sim::systems::JointStatePublisher">
    <joint_name>prismatic_1_joint</joint_name>
```

```
    <topic>world/empty/model/carobot/joint/prismatic_1_joint/state</topic>
  </plugin>
  <plugin
    filename="gz-sim-joint-state-publisher-system"
    name="gz::sim::systems::JointStatePublisher">
    <joint_name>prismatic_2_joint</joint_name>
    <topic>world/empty/model/carobot/joint/prismatic_2_joint/state</topic>
  </plugin>
  <plugin
    filename="gz-sim-joint-position-controller-system"
    name="gz::sim::systems::JointPositionController">
    <joint_name>prismatic_0_joint</joint_name>
    <topic>world/empty/model/carobot/joint/prismatic_0_joint/cmd_pos</topic>
    <p_gain>68.21</p_gain>
    <i_gain>0.56</i_gain>
    <d_gain>34.14</d_gain>
  </plugin>
  <plugin
    filename="gz-sim-joint-position-controller-system"
    name="gz::sim::systems::JointPositionController">
    <joint_name>prismatic_1_joint</joint_name>
    <topic>world/empty/model/carobot/joint/prismatic_1_joint/cmd_pos</topic>
    <p_gain>68.21</p_gain>
    <i_gain>0.56</i_gain>
    <d_gain>34.14</d_gain>
  </plugin>
  <plugin
    filename="gz-sim-joint-position-controller-system"
    name="gz::sim::systems::JointPositionController">
    <joint_name>prismatic_2_joint</joint_name>
    <topic>world/empty/model/carobot/joint/prismatic_2_joint/cmd_pos</topic>
    <p_gain>68.21</p_gain>
    <i_gain>0.56</i_gain>
    <d_gain>34.14</d_gain>
  </plugin>
</gazebo>
```

Listing 3: Etiqueta gazebo del archivo URDF

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64

lim = [
        [-0.186, 0],
        [-0.184, 0],
        [0, 0.224]
    ]

class JointStatePublisher(Node):
    def __init__(self):
        super().__init__('robot_joint_position_publisher')

        self.publisher = [self.create_publisher(
            Float64,
            f'/world/empty/model/carobot/joint/prismatic_{i}_joint/cmd_pos',
            10
        ) for i in range(4)]

        timer_period = 0.1
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        msg = Float64()

        try:
            try:
                input_srt = input('>> ')
            except KeyboardInterrupt:
                self.get_logger().fatal(f'{"X # " * 20}')
                exit(0)
```

```
            pos = list(map(float, input_srt.split(',')))
            if len(pos) == 3:
                for i in range(3):
                    if pos[i] >= lim[i][0] and pos[i] <= lim[i][1]:
                        msg.data = pos[i]
                        self.publisher[i].publish(msg)
                self.get_logger().info(f'{pos}')
        except:
            pass

def main(args=None):
    rclpy.init(args=args)
    joint_state_enricher = JointStatePublisher()
    rclpy.spin(joint_state_enricher)
    joint_state_enricher.destroy_node()

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

Listing 4: Script de publicación de las posiciones de las juntas

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

names = ['prismatic_0_joint', 'prismatic_1_joint', 'prismatic_2_joint']
positions = [0.0, 0.0, 0.0]
velocities = [0.0, 0.0, 0.0]
efforts = [0.0, 0.0, 0.0]

class JointStatePublisher(Node):
    def __init__(self):
        super().__init__('robot_joint_position_listener')
        self.publisher_ = self.create_publisher(
            JointState,
            'joint_states',
```

```
            10)

        self.subscriber_ = [ self.create_subscription(
            JointState,
            f'/world/empty/model/carobot/joint/prismatic_{i}_joint/state',
            self.gz_callback,
            10) for i in range(3) ]

    def gz_callback(self, msg):
        global positions, velocities, efforts
        joint_state = JointState()
        joint_state.header.stamp = self.get_clock().now().to_msg()
        joint_state.name = names

        match msg.name[0]:
            case 'prismatic_0_joint':
                positions[0] = msg.position[0]
                velocities[0] = msg.velocity[0]
                efforts[0] = msg.effort[0]
            case 'prismatic_1_joint':
                positions[1] = msg.position[0]
                velocities[1] = msg.velocity[0]
                efforts[1] = msg.effort[0]
            case 'prismatic_2_joint':
                positions[2] = msg.position[0]
                velocities[2] = msg.velocity[0]
                efforts[2] = msg.effort[0]

        joint_state.position = positions
        joint_state.velocity = velocities
        joint_state.effort = efforts
        self.publisher_.publish(joint_state)
        self.get_logger().info(f'{positions}')

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

```
    joint_state_publisher = JointStatePublisher()
    rclpy.spin(joint_state_publisher)
    joint_state_publisher.destroy_node()
    rclpy.shutdown()

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

Listing 5: Script de lectura de las posiciones de las juntas

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription,
    ExecuteProcess
from launch.conditions import IfCondition, UnlessCondition
from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare

def generate_launch_description() -> LaunchDescription:
    ld = LaunchDescription()

    pkg_share = FindPackageShare(package='carobot')

    default_model_path = PathJoinSubstitution([pkg_share, 'urdf', 'model.urdf'])
    default_rviz_path = PathJoinSubstitution([pkg_share, 'rviz', 'urdf.rviz'])

    gui_arg = DeclareLaunchArgument(
        name='gui',
        default_value='false',
        choices=['true', 'false'],
        description='Flag to enable/disable the GUI'
    )
    ld.add_action(gui_arg)

    rviz_arg = DeclareLaunchArgument(
        name='rvizconfig',
        default_value=default_rviz_path,
```

```
        description='Path to the RViz configuration file'
    )
    ld.add_action(rviz_arg)

    model_arg: DeclareLaunchArgument = DeclareLaunchArgument(
        name='model',
        default_value=default_model_path,
        description='Path to the URDF model file'
    )
    ld.add_action(model_arg)

    jsp_node = Node(
        package='carobot',
        executable='pos_get.py',
        name='robot_position_publisher',
    )
    ld.add_action(jsp_node)

    ild: IncludeLaunchDescription = IncludeLaunchDescription(
        PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch',
            'display.launch.py']),
        launch_arguments={
            'urdf_package': 'carobot',
            'urdf_package_path': LaunchConfiguration('model'),
            'rviz_config': LaunchConfiguration('rvizconfig'),
            'jsp_gui': LaunchConfiguration('gui'),
            'use_gui': 'false',
        }.items(),
    )
    ld.add_action(ild)

    gz_create_node = Node(
        package='ros_gz_sim',
        executable='create',
        name='spawner',
        output='screen',
```

```
        arguments=[
            '-name', 'carobot',
            '-topic', '/robot_description',
        ]
    )

    ld.add_action(gz_create_node)

    gz_bridge_node = Node(
        package='ros_gz_bridge',
        executable='parameter_bridge',
        name='robot_gz_bridge',
        output='screen',
        arguments=[
            '/world/empty/model/carobot/joint/prismatic_0_joint/cmd_pos@
            std_msgs/msg/Float64]gz.msgs.Double',
            '/world/empty/model/carobot/joint/prismatic_0_joint/state@
            sensor_msgs/msg/JointState[gz.msgs.Model',
            '/world/empty/model/carobot/joint/prismatic_1_joint/cmd_pos@
            std_msgs/msg/Float64]gz.msgs.Double',
            '/world/empty/model/carobot/joint/prismatic_1_joint/state@
            sensor_msgs/msg/JointState[gz.msgs.Model',
            '/world/empty/model/carobot/joint/prismatic_2_joint/cmd_pos@
            std_msgs/msg/Float64]gz.msgs.Double',
            '/world/empty/model/carobot/joint/prismatic_2_joint/state@
            sensor_msgs/msg/JointState[gz.msgs.Model',

        ],
    )
    ld.add_action(gz_bridge_node)

    """
    # Agregar nodo conector entre joint_states y gz
    ld.add_action(Node(
        package='carobot',
        executable='republisher.py',
```

```
        name='joint_states_enricher',
        output='screen',
    ))
    """

    gz_sim_node = ExecuteProcess(
        cmd=['gz', 'sim', '-r', '-v', '4', 'empty.sdf'],
        output='screen'
    )
    ld.add_action(gz_sim_node)

    return ld
```

Listing 6: Lanzador del paquete carobot