UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un sistema de soldadura robotizado inteligente usando una cámara de profundidad y un robot colaborativo.

Trabajo Fin de Máster

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Górriz Aliaga, Abel

Tutor/a: Ivorra Martínez, Eugenio

CURSO ACADÉMICO: 2023/2024

Resumen

El presente trabajo aborda el diseño y desarrollo de un sistema de soldadura robotizado inteligente. Se utiliza una cámara de profundidad Intel RealSense 415D acoplada en la herramienta de un robot UR3e.

El sistema es capaz de identificar piezas de diversas formas y tamaños en un área específica y realizar una soldadura en dicha pieza.

Para lograr este objetivo se crea una nube de puntos del objeto tomando imágenes desde diferentes ángulos gracias al movimiento del robot. Una vez conocido el objeto y su posición en el espacio, se calculan trayectorias específicas para que el robot se acerque y simule la ejecución de una soldadura.

Para la programación y control del robot, así como para la manipulación y análisis de datos relacionados con la visión y la identificación de objetos, se emplea el entorno de desarrollo de Python, junto con el software de programación de robots RoboDK.

Summary

This work addresses the design and development of an intelligent robotic welding system. It utilizes an Intel RealSense 415D depth camera integrated into the tool of a UR3e robot. The system can identify pieces of various shapes and sizes in a specific area and perform welding on the identified piece.

To achieve this goal, a point cloud of the object is generated by capturing images from different angles through the movement of the robot. Once the object and its position in space are known, specific trajectories are calculated for the robot to approach and simulate the execution of a weld.

For the programming and control of the robot, as well as for the manipulation and analysis of data related to vision and object identification, the Python development environment is employed, along with the RoboDK robot programming software.

Agradecimientos

Quiero aprovechar este espacio para agradecer a todas las personas que han sido parte de mi viaje universitario. Primero, a mi familia, por su amor y apoyo incondicional que me ha impulsado en cada paso. También a mis profesores y mentores, que me han brindado su sabiduría y paciencia para crecer académicamente. Agradezco a mis compañeros de clase, Javier Gámir, Hipòlit Raigal y Marc Fontalba por los momentos compartidos y el apoyo mutuo durante los años de la carrera y el máster. Por último, agradezco a mi novia Patri por aguantarme y ayudarme en todo lo posible durante toda mi etapa universitaria.

Índice de documentos

DOCUMENTO Nº1: Memoria ***9***
DOCUMENTO Nº2:Planos ***123***
DOCUMENTO Nº3: Pliego de Condiciones ***137***
DOCUMENTO Nº4: Presupuesto ***145***

UNIVERSITAT POLITÉCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un sistema de soldadura robotizado inteligente usando una cámara de profundidad y un robot colaborativo.

DOCUMENTO Nº1: Memoria

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Górriz Aliaga, Abel
Tutor/a: Ivorra Martínez, Eugenio
CURSO ACADÉMICO:2023-2024

Índice de contenidos

1. OBJETIVO 13

2. ANTECEDENTES 14

2.1. Soldadura Robótica 14

2.2. Visión por Computador 14

2.3. Robots Colaborativos 15

2.4. Visión por computador y robótica 15

3. ESTUDIO DE NECESIDADES 16

4. PLANTEAMIENTO DE SOLUCIONES ALTERNATIVAS Y JUSTIFICACIÓN DE LA SOLUCIÓN ADOPTADA 18

5. DESCRIPCIÓN DETALLADA DE LA SOLUCIÓN 19

5.1. Preparación de la estación 19

5.1.1. Diseño de la realidad en RoboDK 19
5.1.2. Diseño del soporte para la cámara 19
5.1.3. Camera-Hand-Calibration 22

5.2. Identificación del objeto y procesado 24

5.2.1. Procesamiento de imágenes 25
5.2.2. Segmentación del objeto 28

5.3. Cálculo de trayectorias y ejecución 29

5.3.1. Posicionamiento de la herramienta del robot 29
5.3.2. Puntos en objeto de dos planos 31
5.3.3. Targets en objeto de 2 planos 32
5.3.4. Puntos en objeto de cinco planos 34
5.3.5. Targets en objeto de 5 planos, interior de la caja 34
5.3.6. Targets en objeto de cinco planos, exterior de la caja 36
5.3.7. Realización de trayectorias 37

5.4. Método ICP 38

5.5. Interacción RoboDK-UR3e y distintas configuraciones y posibilidades 41

5.5.1. Interfaz de configuración 41
5.5.2. Interfaz de aplicación 43
5.5.3. Interfaz final del movimiento 44

5.6. Seguridad 45

6. JUSTIFICACIÓN DETALLADA DE LA SELECCIÓN O DIMENSIONAMIENTO DE ELEMENTOS 48

6.1. Elementos alternativos 48

6.1.1. Robot ABB IRB 140 48

6.2. Elementos seleccionados 49

6.2.1. Robot colaborativo UR3e 49
6.2.2. Intel Realsense D415 49
6.2.3. Quick changer tool OnRobot 50

7. PRUEBAS Y RESULTADOS 51

7.1. PRUEBAS EN SIMULACIÓN 51
7.1.1. Intersección de dos planos 51
7.1.2. Interior de una caja 55
7.1.3. Exterior de una caja 59
7.1.4. Método ICP 63
7.2. PRUEBAS EN EL LABORATORIO 68
8. CONCLUSIONES 74
9. BIBLIOGRAFÍA 75
ANEXO 1. RELACIÓN DEL TRABAJO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE DE LA AGENDA 2030 77
ANEXO 2. FICHAS TÉCNICAS DE LOS ELEMENTOS 79
ANEXO 3. CÓDIGO IMPLEMENTADO 89

1. OBJETIVO

El objetivo del proyecto es la creación de una estación de soldadura o cualquier otro tipo de trabajo, como de pintura o aplicación de adhesivo sobre un objeto con unan forma determinada.

Para la creación de la célula de trabajo se utilizará el robot colaborativo UR3e del fabricante Universal Robots.

La identificación de los objetos se realizará mediante la cámara de profundidad Intel RealSense 415D, del fabricante de circuitos integrados Intel Corporation. Dicha cámara estará integrada en el robot de forma que se moverá juntamente con él.

Una vez determinada la posición y orientación del objeto se calculará las trayectorias que deberá seguir el robot para realizar la aplicación deseada.

Se van a desarrollar 4 tipos de aplicación:

- Intersección de dos planos
- Interior de una caja
- Exterior de una caja
- Objeto usando ICP (Iterative Closest Point)

Para la programación y control del robot, así como para la manipulación y análisis de datos relacionados con la visión por computador y la identificación de objetos, se emplea el entorno de desarrollo de Python y la librería Open3D para el procesamiento de imágenes 3D, junto con el software de programación de robots RoboDK.

2. ANTECEDENTES

Para crear un contexto actual del proyecto, conviene conocer aspectos básicos como, la soldadura robótica, la utilidad de los robots colaborativos y el concepto de visión artificial.

2.1. Soldadura Robótica

La soldadura robótica es una versión avanzada de la soldadura automatizada, esta implica realizar el proceso de soldadura mediante robots, controlados por programas reprogramables para adaptarse a proyectos específicos y ofrecer mayor adaptabilidad.

El uso de tecnología robótica permite obtener resultados precisos y rápidos, con menos desperdicio y mayor seguridad. Los robots llegan a lugares que, por otros medios, serían inaccesibles y pueden realizar líneas de soldadura y soldaduras complicadas y precisas más rápidamente que con la soldadura manual.

Se utiliza en industrias del metal, pesada y automotriz para soldaduras por puntos y láser, ofreciendo alta productividad y eficiencia. Es adecuada para producciones en masa, pero también adaptable a tiradas pequeñas o únicas, manteniendo una alta rentabilidad.

2.2. Visión por Computador

La visión artificial o por computador engloba tanto aplicaciones industriales y no industriales en donde gracias a la combinación de hardware y software se obtienen datos e información útil basándose en la captura y el procesamiento de imágenes para reconocer y analizar características específicas de dichas imágenes como formas, colores, texturas y contornos. La visión por computador se usa en diversas áreas, como la robótica, la automatización industrial, la seguridad, la medicina, entre otros. Algunas de las técnicas utilizadas en la visión artificial incluyen el reconocimiento de patrones, el aprendizaje automático, la segmentación de imágenes, la detección de bordes, la extracción de características y la fusión de imágenes.

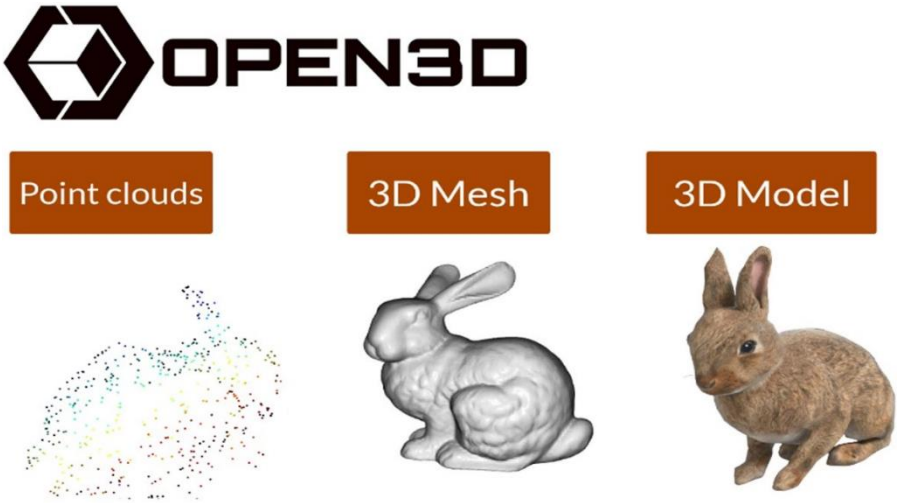


Fig 1: Distintas representaciones de un conejo

Open3D es una biblioteca de código abierto que soporta el desarrollo de software para datos en 3D. La librería contiene un conjunto de estructuras de datos y algoritmos cuidadosamente seleccionados tanto en C++ como en Python, además de estar altamente optimizado.

Open3D ha sido utilizado en varios proyectos de investigación y se despliega activamente en la nube, gracias también a las contribuciones de la comunidad de código abierto.

En el proyecto se van a usar mayoritariamente nubes de puntos o point clouds (pcd), para reconstruir el objeto y partir de ello extraer información necesaria para la ejecución de la acción. También se utilizarán las mallas o mesh, estas sirven para unir los puntos de una pcd y poder crear un objeto.

2.3. Robots Colaborativos

Fig 2: Robot UR3e

Los robots colaborativos o cobots tienen la finalidad de automatizar procesos industriales con el propósito de optimizar dichos procesos. Estos robots son un tipo de robot industrial, articulado de pequeño tamaño muy novedosos debido a su factor colaborativo, por el cual podemos encontrarlos trabajando en con junto con el personal humano sin que ello ponga en riesgo su seguridad.

Estos robots son ideales para realizar tareas repetitivas y evitar lesiones en los trabajadores, así como para trabajar en industrias que requieran altas temperaturas, materiales tóxicos, entre otros. La seguridad de este tipo de robots es gracias a los sensores integrados que les permiten responder a estímulos externos y evitar accidentes. Además, estos robots pueden trabajar como una extensión de otros equipos o asistir a los trabajadores en sus tareas diarias.

2.4. Visión por computador y robótica

En la actualidad gracias a la Industria 4.0 se está optando por la digitalización de las líneas de producción con elementos como la realidad virtual, inteligencia artificial, internet de las cosas y Big Data.

Las innovaciones tecnológicas han permitido que gracias a la unión de visión artificial y robots colaborativos hoy en día estos puedan realizar tareas impensables hasta ahora revolucionando el trabajo en sectores industriales.

Las ventajas de la visión artificial en robots incluyen garantizar la trazabilidad de los procesos de producción, un guiado automático de los brazos robóticos, rapidez y constancia. Las aplicaciones de la visión artificial incluyen pick & place, empaquetado y paletizado, control de calidad y montaje, y son ideales para sectores como el de automoción, alimentario, electrónica y tecnología, entre otros.

3. ESTUDIO DE NECESIDADES

El objetivo del proyecto es desarrollar una estación de trabajo automatizada utilizando el robot colaborativo UR3e y la cámara IntelRealSense 415D. La estación deberá ser capaz de realizar diversas tareas como son el reconocimiento de objeto y su posterior soldadura. La programación y control del sistema se realizarán con Python integrado en el software RoboDK.

A continuación, se detallará las necesidades y capacidad que debe brindar los elementos seleccionados, así como las características de los objetos para las distintas aplicaciones.

- Robot Colaborativo UR3e:

El robot debe tener capacidad para moverse con una alta precisión a la hora de realizar las tareas de soldadura.

El robot tiene una carga útil máxima de 3kg y un alcance de 500mm, por lo que la herramienta no puede ser más pesada que 3kg y los objetos deben estar dentro del rango de alcance del robot.

El robot colaborativo es seguro para trabajar junto a humanos, empleando sistemas de detección de colisiones y paradas de emergencia, cumpliendo con las normativas de seguridad colaborativa ISO/TS 15066.

- Cámara Intel RealSense 415D:

La cámara debe estar montada y bien sujeta en robot, para permitir el movimiento de este con seguridad y tomar las capturas en las posiciones requeridas.

Necesidad de capturar tanto datos de profundidad como RGB con una alta resolución y precisión para el correcto reconocimiento de objetos y formas.

El sensor de profundidad funciona correctamente cuando el objeto a identificar se encuentra en el rango de 500 – 2000 mm del objetivo de la cámara.

- Objetos para la soldadura:

Para la aplicación de soldadura de dos planos se puede realizar sobre un objeto con diversas dimensiones, siempre dentro del alcance del robot y teniendo en cuenta la resolución de la cámara. Se debe cumplir que la intersección sea una línea recta.

Para la aplicación de soldadura de exterior de la caja se puede realizar a una gran variedad de dimensiones de esta, siempre dentro del alcance del robot y teniendo en cuenta la resolución de la cámara. Se debe cumplir que las intersecciones exteriores de la caja son líneas rectas.

Para la aplicación de soldadura de interior de la caja se puede realizar para una gran variedad de dimensiones de esta, siempre dentro del alcance del robot y teniendo en cuenta la resolución de la cámara, además se deberá tener en cuenta el grado de inclinación y la forma y grosor de la herramienta a la hora de realizar la soldadura. Se debe cumplir que las intersecciones interiores de la caja son líneas rectas.

Para la aplicación usando el algoritmo ICP se debe cumplir que el objeto sea lo más simple posible y que sea capaz el algoritmo de resolver el problema de correspondencia. Para cada objeto se deberán programar las trayectorias de forma individual.

- Software de Programación

Uso del software RoboDK por su capacidad de simular trayectorias de robots, posibilidad de uso de cámaras y la integración de Python para su posible programación.

Uso de Python por su versatilidad y gran cantidad de bibliotecas, incluyendo Open3D para el procesamiento de imágenes 3D y otras para el uso de Python en RoboDK.

4. PLANTEAMIENTO DE SOLUCIONES ALTERNATIVAS Y JUSTIFICACIÓN DE LA SOLUCIÓN ADOPTADA

La primera idea para el desarrollo de la estación fue colocar la cámara estática en posición cenital. De esta forma se podría identificar la posición y orientación de los objetos.

Fig 3: Cámara fija

El problema que supone esta solución depende de la complejidad del objeto, de esta forma solo es posible conocer información de la parte superior del objeto.

Otra solución es la utilización de más de una cámara y colocarlas en distintos puntos de la estación de trabajo. De esta forma se resolvería el problema de la primera solución, la perdida de información del objeto. Con motivo de abaratar costes esta opción se descarta, debido al alto precio de una cámara de profundidad.

Finalmente se ha optado por diseñar una herramienta para sostener la cámara con el robot, de esta forma la cámara puede tomar imágenes desde distintos puntos del entorno y obtener la información necesaria de los objetos.

Fig 4: Cámara integrada en el brazo

5. DESCRIPCIÓN DETALLADA DE LA SOLUCIÓN

5.1. Preparación de la estación

Para el desarrollo del sistema robotizado, primero se debe conocer el entorno donde se va a trabajar, con el fin de recrearlo en RoboDK y limitar el movimiento del robot para su seguridad y la de las personas. También se ha de diseñar la herramienta con la que se va a trabajar para poder sostener la cámara y simular la soldadura.

Una vez se haya diseñado el soporte y colocado en el robot se deberá realizar una calibración mano-cámara (camera-hand-calibration), necesaria para conocer con precisión la posición del ojo de la cámara respecto al robot.

5.1.1. Diseño de la realidad en RoboDK

Para el diseño de una estación en RoboDK se han escogido elementos proporcionados por el software y se han redimensionado para obtener las mismas medidas que en la realidad.

Se ha añadido un objeto de seguridad dentro de la estación el cual el robot no traspasara, esto se ha implementado con la idea de proteger el robot y la cinta del laboratorio. También se ha marcado en la mesa un área que marca el alcance aproximado del robot, para no colocar los objetos más lejos.

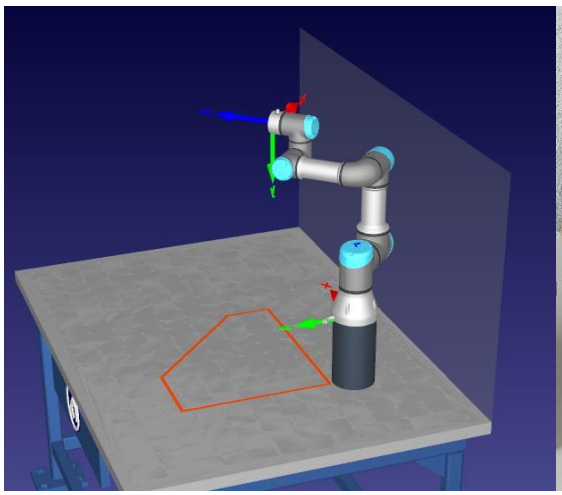


Fig 5: Estación de trabajo simulada

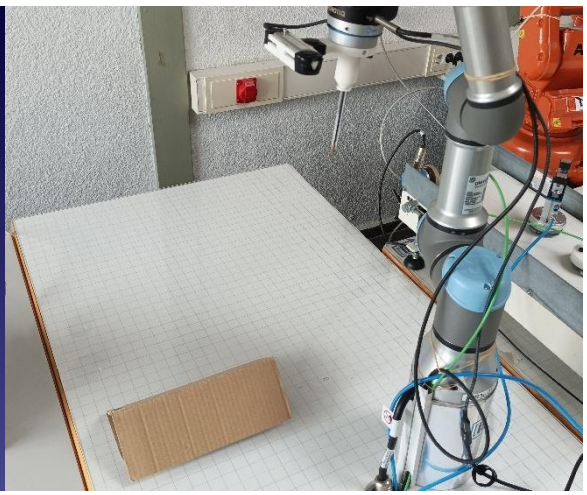


Fig 6: Estación de trabajo real

5.1.2. Diseño del soporte para la cámara

El diseño del soporte debe ser capaz de atornillarse al efector final del robot, sostener la cámara de forma firme y segura, y a modo de prueba para el proyecto, poder introducir un rotulador o lápiz para simular la soldadura.

Se ha pretendido utilizar el menor material posible para la impresión 3D, pero a su vez ser lo suficientemente robusto a los movimientos del robot.

El soporte consta de tres partes:

- Parte trasera
- Parte frontal
- Cavidad de la cámara

La parte trasera tiene las mismas medidas que el efector final del robot, en el caso de las pruebas en el laboratorio a un intercambiador de herramientas con las mismas medidas que el robot. Esta parte se encarga de anclarse al robot.

Fig 7: Parte trasera del soporte

La función de la parte frontal es dar posibilidad de introducir objetos. Esta cavidad es bastante ancha, ya que existe la posibilidad de diseñar unos acoples más precisos y únicos para el tipo y grosor del objeto a introducir.

Fig 8: Parte frontal del soporte

Fig 9: Tubo extensible del soporte

La cavidad de la cámara se ha diseñado a partir de las medidas de la cámara, dejando la parte frontal y trasera de la cámara al descubierto. Se ha diseñado con huecos para la correcta ventilación del sistema.

Fig 10: Espacio de la cámara

Debido a la falta de un tornillo para anclar con más seguridad la cámara, se ha diseñado un mecanismo de fácil uso simulando el tornillo.

Fig 11: Funcionamiento del "tornillo"

Fig 12: piezas simulando un tornillo

En la figura de la izquierda se muestra el funcionamiento del elemento que se introduce y hace a la perfección la función de un tornillo, manteniendo fija la cámara en el soporte.
Después se le añade cinta aislante a la parte donde sobresale la pieza negra, para asegurar que no se salga.

Finalmente, se le añade el intercambiador de herramientas y se coloca en el brazo robótico:

Fig 13: Intercambiador

Fig 14: Soporte colocado y montado en el robot

5.1.3. Camera-Hand-Calibration

Con la calibración mano-cámara, se obtendrán los parámetros intrínsecos de la cámara Intel RealSense D415 y la relación de la posición de la cámara con el cobot.

Los parámetros intrínsecos obtenidos proporcionan la información y características de la cámara que normalmente son únicos para cada cámara debido a tolerancias de fabricación. Estos parámetros conforman una matriz necesaria a la hora de la toma de datos de cualquier tipo, en este caso se usa para la creación de una nube de puntos a partir de la imagen capturada.

La relación entre la posición de la cámara y el robot forman otra matriz, esta es llamada matriz de parámetros extrínsecos ya que no dependen de la cámara, no como la anterior matriz que si dependen de ella. Dicha matriz se utiliza para posicionar la nube de puntos creada en el mundo teniendo como referencia la base del robot.

Para la realización de dicha calibración se necesita un patrón, en este caso un chessboard en el cual se conoce el número de cuadrados y su tamaño. Se ha elegido el tablero de ajedrez debido a que es un patrón regular y fácil de usar donde gracias a él se define las dimensiones del mundo. Una vez se comienza con el proceso el patrón no debe moverse, de lo contrario los datos capturados arrojaran un resultado erróneo.

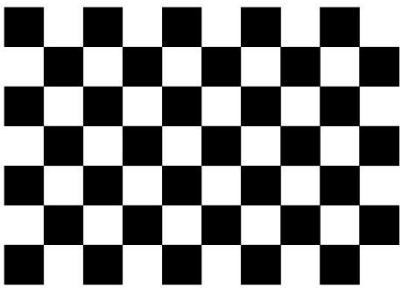


Fig 15: Patrón Chessboard utilizado

El proceso de calibración se divide en tres partes:

1. Toma de datos: El robot se mueve a varias posiciones y toma capturas de la cámara, también se registran el valor de las 6 articulaciones del robot. La cámara debe enfocar al patrón chessboard.

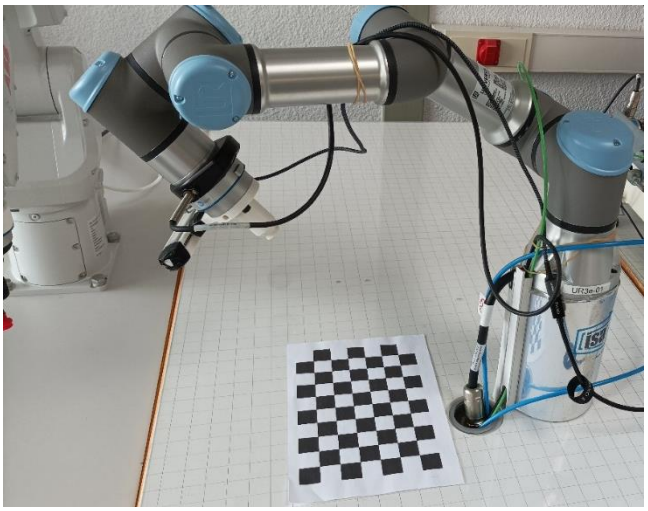


Fig 16: Robot orientado al patrón

2. Camera-Calibration: Se realiza una primera calibración solo con las imágenes para obtener la matriz de parámetros intrínsecos. Se utilizan algoritmos de calibración proporcionados por OpenCV que detectan las esquinas del tablero y calculan dichos parámetros.

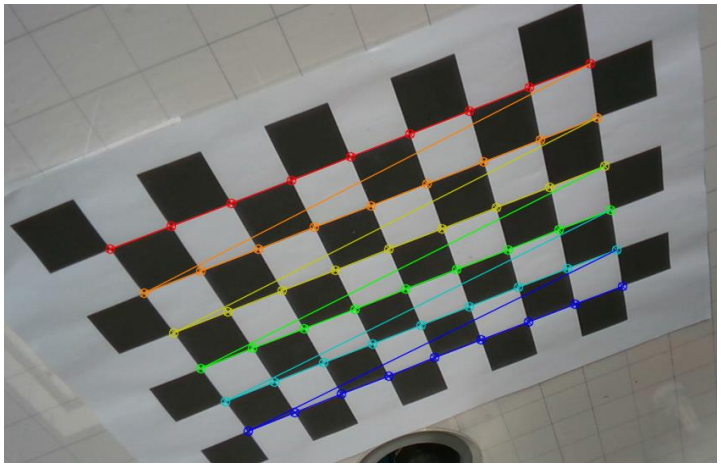


Fig 17: Puntos de interés de la imagen capturada

3. Camera-Hand-Calibration: En este último paso se usa las imágenes y los valores de articulaciones. De forma similar a la calibración anterior se usan algoritmos proporcionados por OpenCV que relacionan la información obtenida de la cámara con los valores de las articulaciones, obteniéndose la matriz de parámetros extrínsecos.

Tras realizar dicho proceso de calibración ya es posible crear nubes de puntos, gracias a la matriz de parámetros intrínsecos, y posicionarlas correctamente referenciadas a la base del robot, gracias a la matriz de parámetros extrínsecos.

5.2. Identificación del objeto y procesado

Para reconstruir el objeto se realizan distintas capturas desde varios puntos en el espacio para obtener toda la información del objeto, tanto forma como posición. Este problema se resuelve en tres bloques diferenciados:

1. Se realiza la captura de las imágenes de profundidad y la obtención de la posición de la cámara en el momento su ejecución
2. Se realiza un filtrado y transformación de las imágenes para formar el objeto con la mayor información posible.
3. el objeto se segmenta en planos, los cuales se utilizarán para el cálculo de las trayectorias.

Primero se han elegido 5 puntos en el espacio para realizar la captura, estos objetivos deben de ser alcanzables por el robot y deben permitir la correcta orientación del objetivo de la cámara hacia el centro de la mesa, donde se colocará el elemento representar. Se debe tener en cuenta que el sensor de profundidad de la cámara Intel RealSense D415 funcionan correctamente a partir de 50cm, por lo tanto, el objeto no puede ser demasiado grande ya que el robot tiene un corto alcance y no se realizaría correctamente la identificación.

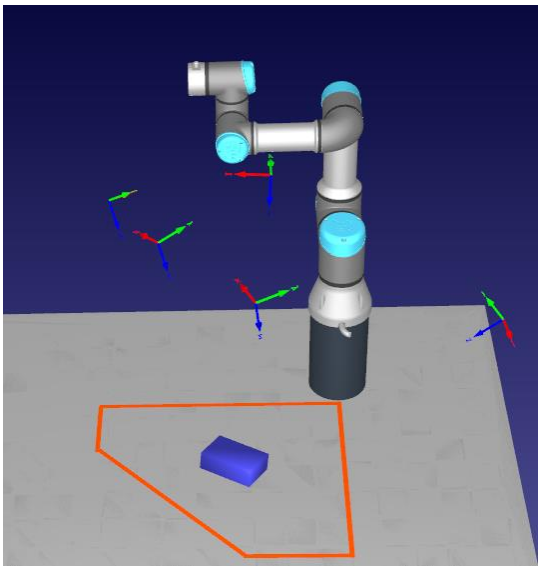


Fig 18: Targets en el espacio de trabajo

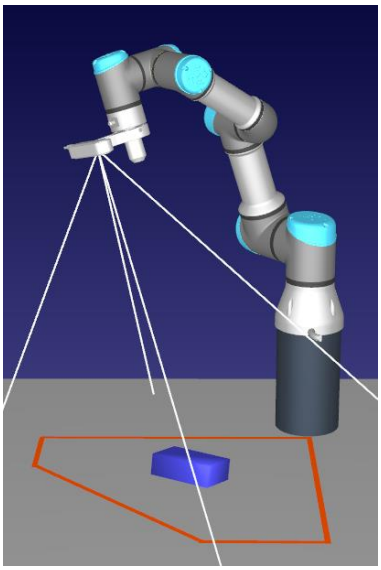


Fig 19: Campo de visión de la cámara

Una vez definidos los 5 targets que cumplan los requisitos y el objeto esté en posición, se comienza con la captura de las imágenes. El robot se coloca en cada target y se obtiene la información del sensor de profundidad.

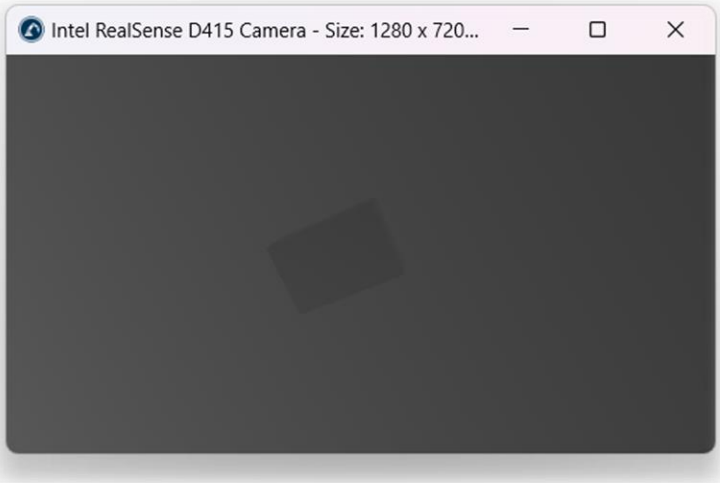


Fig 20: Imagen de la cámara

La última figura es una representación en simulador de lo que se vería con la cámara de profundidad, es una imagen en escala de grises, donde a más cerca esta el objeto se hace más oscuro.

Cabe mencionar que en dos de las cuatro aplicaciones solo se realiza una captura de imagen, ya que es suficiente para obtener la información necesaria de los objetos.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def get_image_and_camera_pose(...) :'*** *en el Anexo 3, página 99*

5.2.1. Procesamiento de imágenes

Con el procesamiento de las imágenes se pretende conseguir la reconstrucción del objeto y filtrar todo lo que no sea necesario.

Para cada imagen el primer paso es la creación de una nube de puntos con la información obtenida y los parámetros intrínsecos de la cámara. Después se aplica una transformación a la "point cloud" para posicionarla correctamente en el espacio, ya que cada imagen es tomada de distintos puntos y estas deben estar bien referenciadas. Se toma como origen la base del robot.

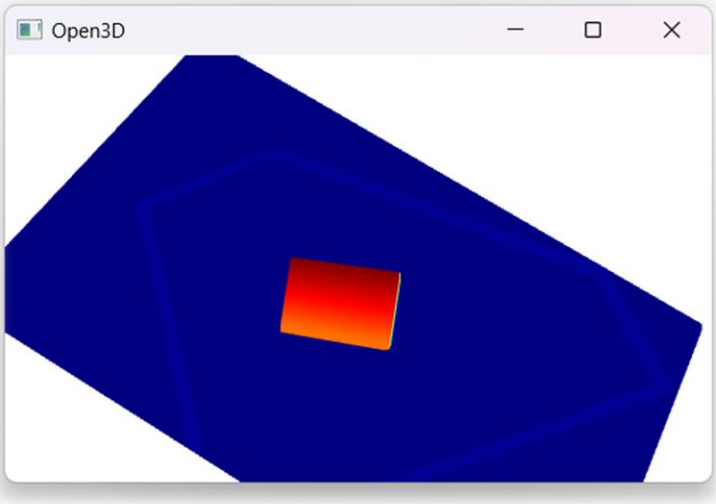


Fig 21: PCD sin filtrar

En la figura anterior se puede observar cómo quedaría una nube de puntos. En este instante también se ha obtenido información de la mesa y hasta incluso en ciertas imágenes aparece la base del robot u otros objetos cercanos.

Para resolver el problema se deben aplicar una serie de filtros para quedarnos con los puntos de interés.

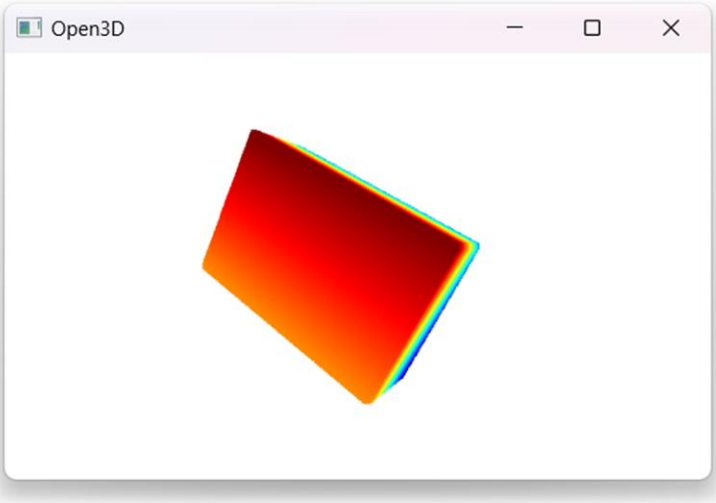


Fig 22: PCD filtrada

Ahora ya solo tenemos información del objeto. Este tipo de filtrado se basa en eliminar puntos que pasen cierta posición en el espacio XYZ, por ejemplo, el filtro más claro es la eliminación de la mesa.

Los objetos y los targets del robot tienen como origen la base del robot que está elevada 200mm sobre la mesa. Por lo tanto, para eliminar los puntos de la mesa se eliminan los puntos que estén por debajo de -200mm en el eje Z. De esta forma se puede delimitar una zona que se deberá mantener vacía y solo se podrá colocar el objeto a reconstruir.

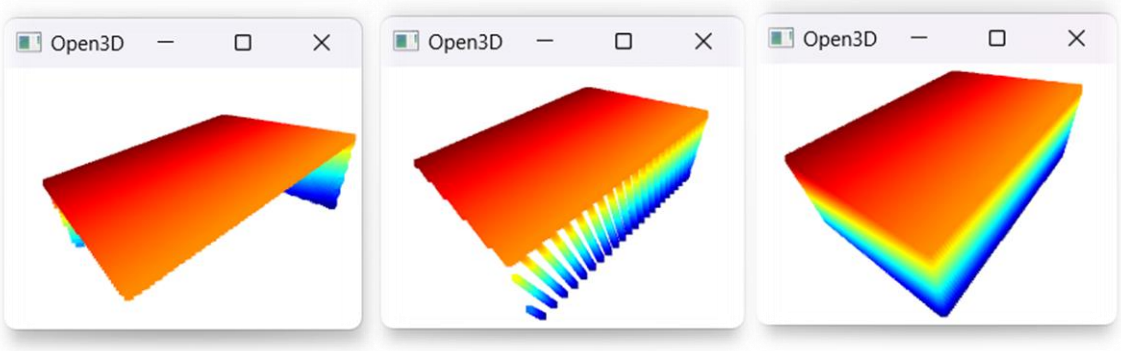


Fig 23: Reconstrucción del objeto con diferentes PCD

Gracias a la construcción de las diferentes nubes de puntos, luego su correspondiente transformación a la posición correcta y el filtrado adecuado se puede pasar a la suma de ellas. En la figura anterior se puede apreciar cómo se va obteniendo toda la información del objeto de interés a medida que se van añadiendo las pcd con distinta información.

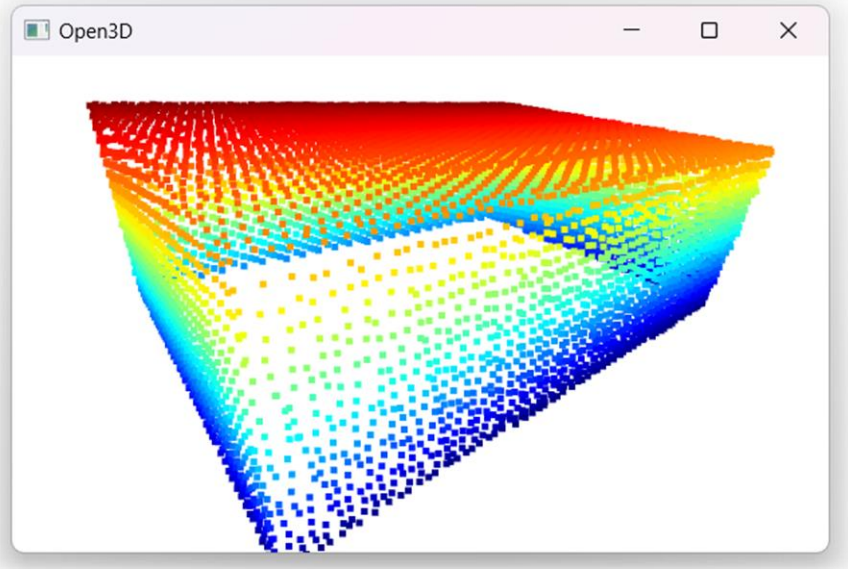


Fig 24: PCD final

Después de realizar un proceso de voxelización, una reducción de puntos de la nube, se obtiene la representación del objeto con información de todas sus caras, menos la inferior.

Este proceso de voxelizado se realiza para reducir los puntos, que implica una reducción del coste computacional, pero no una pérdida de la información que representa al objeto.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def get_point_cloud(...) :'*** *en el Anexo 3, página 100*

5.2.2. Segmentación del objeto

La forma que se ha empleado para segmentar planos es mediante el método ".segment_plane" de open3d. A partir de una nube de puntos, el método busca mediante iteración puntos que formen un plano en el objeto. Una vez encontrado devuelve los puntos del plano, así como los coeficientes (A, B, C, D) que define ese plano.

Para seguir con la búsqueda se deberá restar el plano obtenido al objeto inicial. De esta forma cuando se utilice el método de nuevo se obtendrá otro plano.

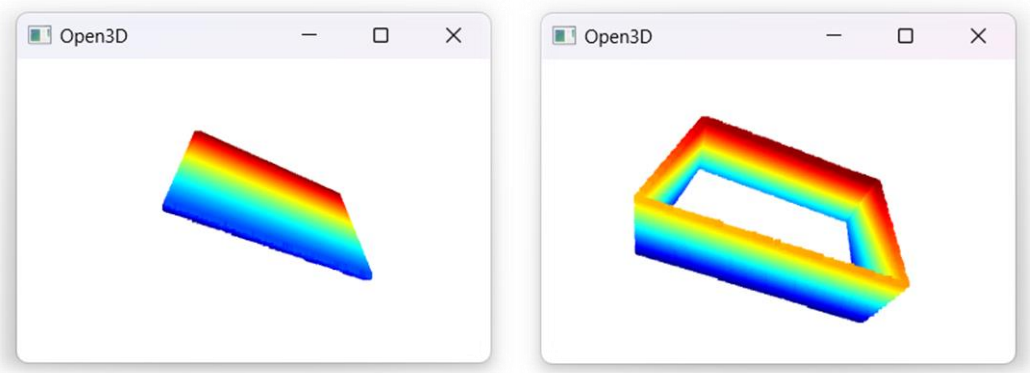


Fig 25: Detección del primer plano

En la figura anterior se muestra el primer plano detectado y su posterior eliminación.

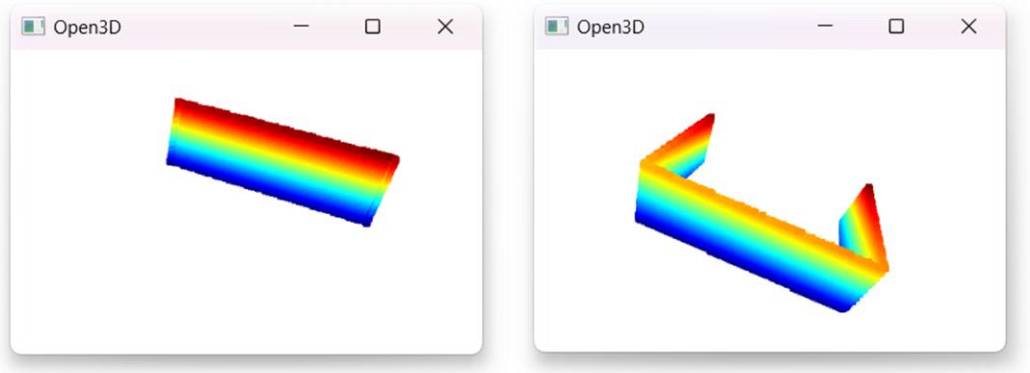


Fig 26: Detección del segundo plano

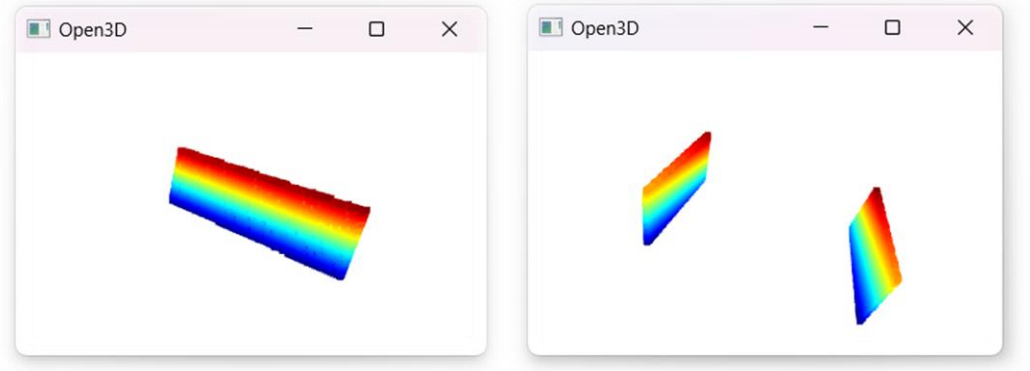


Fig 27: Detección del tercer plano

Las dos últimas figuras corresponden a la obtención del plano 2 y 3. El proceso se termina cuando se obtienen todos los planos, en el caso de una caja, 5.

Para la identificación del objeto plano, se realiza de forma análoga pero solo obteniendo 2 planos.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def get_planes(...) :'*** *en el Anexo 3, página 101*

5.3. Cálculo de trayectorias y ejecución

Una vez identificado el objeto y segmentado en diferentes planos, lo siguiente a realizar es el cálculo de los puntos objetivo y las trayectorias para realizar la soldadura.

Para obtener dicha información hay que seguir dos metodologías, una para el objeto de 2 planos y otra diferente para los objetos de 5 planos, los objetos tipo caja.

Primero se va a explicar cómo se va a posicionar la herramienta del robot en estos puntos que obtendremos.

5.3.1. Posicionamiento de la herramienta del robot

Para posicionar correctamente la herramienta y aumentar la probabilidad de que el robot pueda alcanzar el target se propone una forma para elegir la orientación.

Comúnmente la componente Z de la herramienta en las aplicaciones que se van a realizar apunta hacia al suelo, y para obtener una pose del robot menos problemática, se plantea orientar la componente Y hacia el hombro del robot. Esto se ha elegido así y no al punto de origen debido a la geometría del robot.

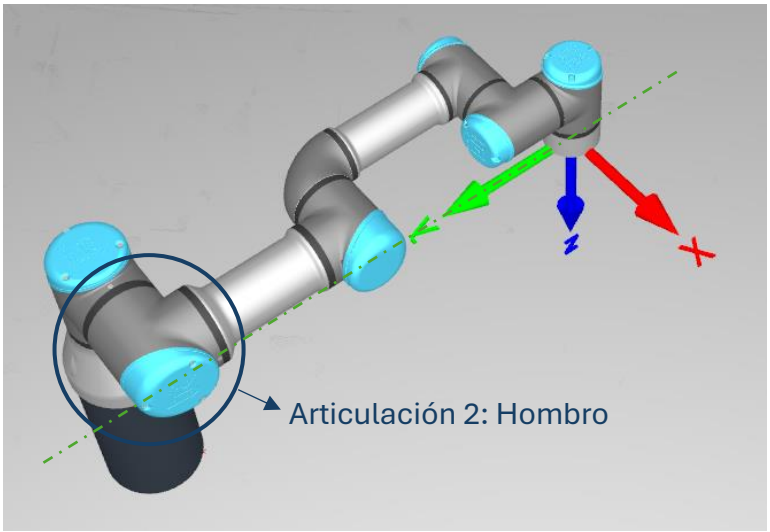


Fig 28: Alineación hombro-muñeca

Como se puede apreciar en la figura anterior, cuando el robot apunta hacia abajo, es fácil alinear la componente Y con la articulación 2.

Conociendo el punto de aplicación de la herramienta en el objeto a soldar y la dirección del eje Z que viene determinada por dicho objeto, se puede calcular la correcta orientación para que el robot este alineado.

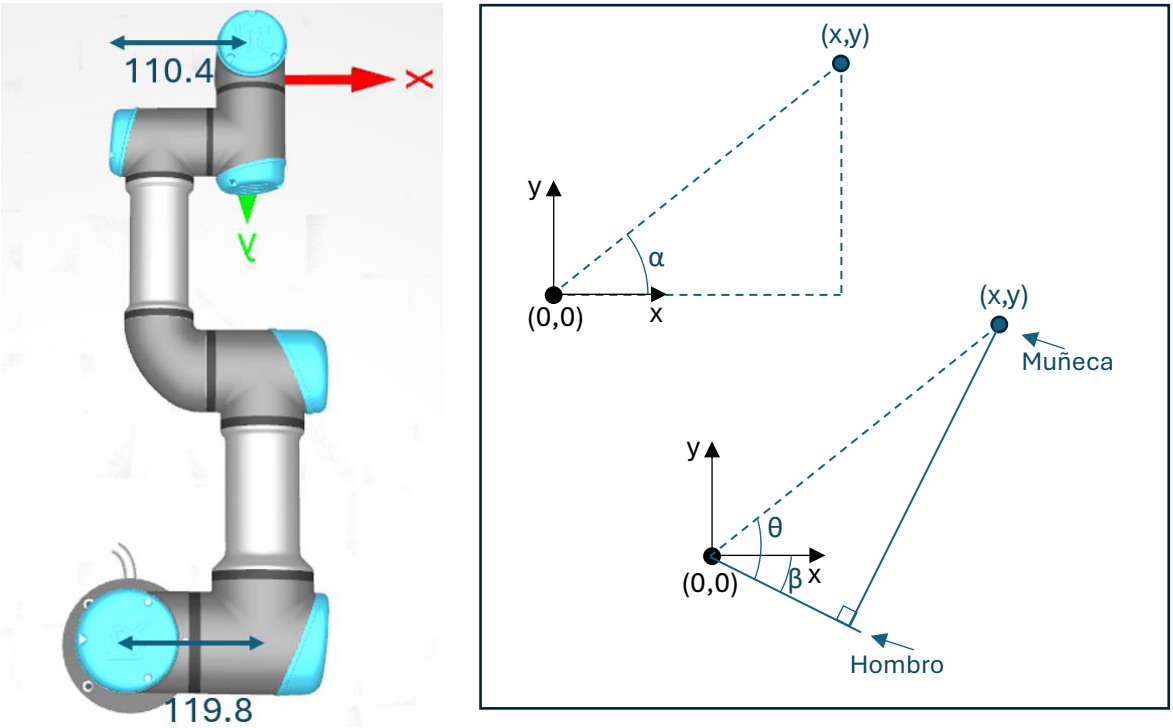


Fig 29: Geometría de robot UR3e

Para ello se plantea el esquema del robot en el plano XY visto desde arriba de este. A partir de calcular ciertos ángulos y con las medidas conocidas del robot se puede obtener el vector director para alinear la herramienta.

Se puede obtener el ángulo α a partir de la información x e y del punto objetivo:

$$\alpha = \cos^{-1}\left(\frac{x}{\sqrt{x^2+y^2}}\right)$$

Conociendo que las articulaciones 1 y 2 forman $90º$ el siguiente ángulo es θ. Se obtiene a partir de x e y, y la distancia donde la muñeca está en línea con el hombro $(110.4\ mm)$:

$$\theta = \cos^{-1}\left(\frac{110.4}{\sqrt{x^2+y^2}}\right)$$

El último paso para conocer el punto es calcular el ángulo β y con las medidas conocidas del robot obtener dicho punto:

$$\beta = \alpha - \theta$$

$$hx = \cos(\beta) \cdot 110.4$$

$$hy = \sin(\beta) \cdot 110.4$$

Una vez definido el vector del punto objetivo al hombro, luego se multiplica en cruz con la orientación en Z, que es conocida, para obtener la orientación en X y posteriormente se multiplica esta última para obtener la orientación en Y definitiva:

$Pobjetivo = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$ $Phombro = \begin{pmatrix} hx \\ hy \\ 0 \end{pmatrix}$ ambos puntos conocidos

$\overrightarrow{Vz} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$ vector de orientación de la componente Z conocido

$\overrightarrow{Vx} =?$ $\overrightarrow{Vy} =?$ Vectores de orientación de las componentes X e Y desconocidos

$$\overrightarrow{ObHom} = Pobjetivo - Phombro$$

$$\overrightarrow{Vx} = \overrightarrow{Vz} \times \overrightarrow{ObHom}$$

$$\overrightarrow{Vy} = \overrightarrow{Vz} \times \overrightarrow{Vx}$$

Finalmente se obtiene el vector unitario y ya es posible crear el target para el robot, donde alcanzará el punto objetivo y tendrá una correcta orientación, manteniendo la cámara lo más alejada de la base para evitar su destrucción.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def calculo_rotXY_hombro (...):'*** *en el Anexo 3, página 103*

5.3.2. Puntos en objeto de dos planos

En este apartado se pretende conseguir el punto inicial y final del objeto de dos planos, por donde deberá comenzar y finalizar la soldadura. Para ello se va a obtener una recta que coincida con la intersección de los planos y comparar esta con la nube de puntos.

Con la información de los dos planos se puede obtener puntos que estén incluidos en su intersección, para ello se debe asumir que la intersección de los dos planos corta el plano X, Y o Z, donde sus valores serán 0 para así poder formar sistemas de ecuaciones con solución única. Además, también se tiene información de un punto céntrico, que está en el interior del área de trabajo.

Se crean distintos sistemas de ecuaciones igualando los planos y asumiendo ciertos valores iguales a 0.

Datos para el primer sistema:

$plano1 = \{A1, B1, C1, D1\}$ $z = 0$

$plano2 = \{A2, B2, C2, D2\}$ $x =?$ $y =?$

Sistema con z = 0:

$$\left\{ \begin{matrix} A1 \cdot x + B1 \cdot y + D1 = 0 \\ A2 \cdot x + B2 \cdot y + D2 = 0 \end{matrix} \right\}$$

A partir del sistema anterior se obtiene un punto en el espacio XYZ con z = 0.

De forma análoga se realiza dos nuevos sistemas, el primero suponiendo x = 0 y el segundo y = 0.

Una vez resueltos los 3 sistemas se obtienen 3 puntos, con cada uno de ellos se calcula la distancia con el punto céntrico y se elige el punto más cercano como origen de la recta que se va a generar, de esta forma se reduce el error al obtener los parámetros de los planos.

Para encontrar el vector director de la recta de intersección, se utiliza el producto en cruz de los vectores normales de los planos. Este cálculo proporciona un vector que es perpendicular a ambos vectores normales, siendo este la intersección. Finalmente se obtiene el vector unitario del resultado.

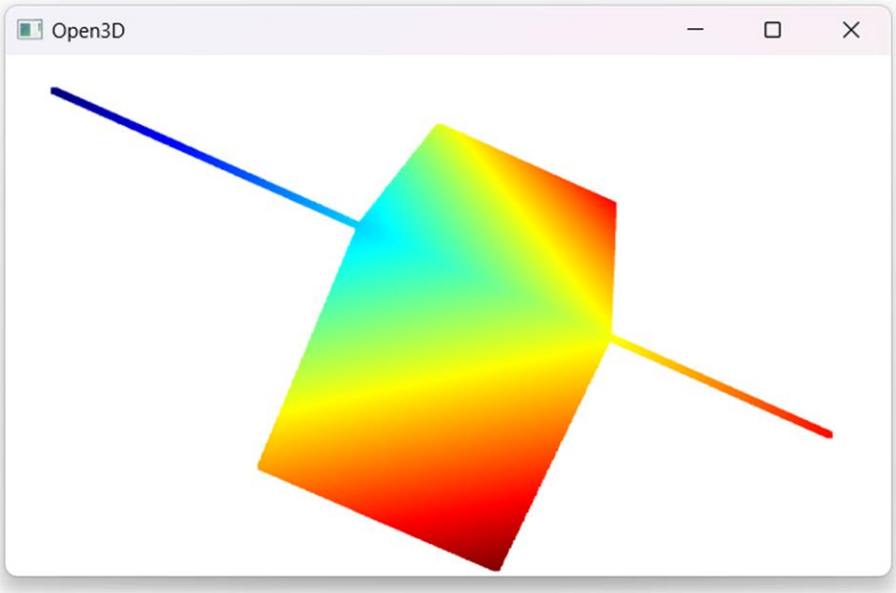


Fig 30: PCD del objeto y la línea de intersección

Con el origen con menor error, elegido anteriormente, y el vector director, se crea una recta comprendida en el espacio de trabajo, como se muestra en la figura anterior. En ella están representados el objeto inicial y la recta intersección.

Finalmente se compara la recta con el objeto de 2 planos, y los puntos de la recta que estén más cercanos a un valor umbral, se guardan en un vector. Luego de este vector se elegirá el primer y último valor, obteniéndose el inicio de la trayectoria y el final de esta.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def plane_object_point(...) :'*** *en el Anexo 3, página 105*

5.3.3. Targets en objeto de 2 planos

Para obtener los targets donde se va a colocar el robot, es necesario conocer el punto y la orientación de la herramienta.

En el objeto de 2 planos se plantea que la componente Z de la herramienta, la correspondiente a la punta del soldador, pase perpendicular a la línea de intersección y a su vez a la misma distancia de los dos planos.

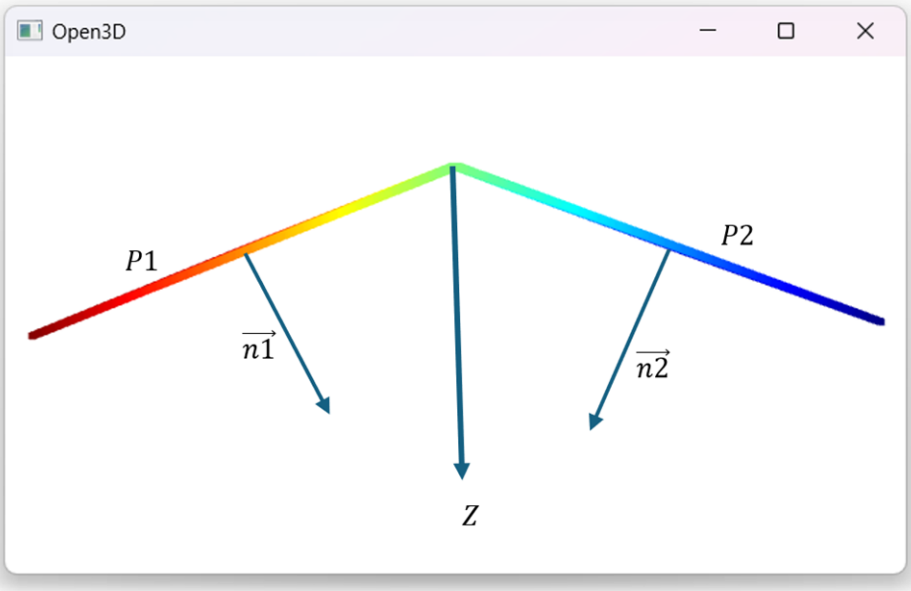
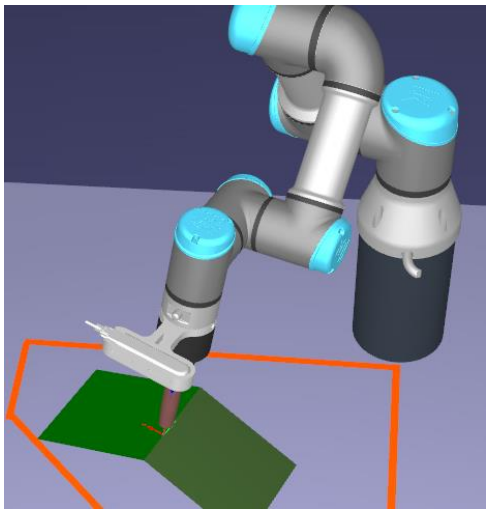


Fig 31: Orientación de la componente Z

En la figura anterior se muestra los dos planos y sus respectivas normales. La componente Z corresponde a la suma de las dos normales.

A partir de los dos puntos que indican el comienzo y el fin de la intersección y la componente Z de orientación, se obtienen las otras dos componentes X e Y de la forma previamente explicada en el punto 5.3.1.

Finalmente conocemos los targets que el robot debe cumplir para realizar la soldadura del objeto, en este caso dos planos.



En la figura 32 se puede observar tanto la correcta posición de la cámara, segura y alejada del robot y la soldadura de los planos, totalmente perpendicular a la intersección.

Fig 32: Soldando 2 planos

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def plane_object_targets(...) :'*** *en el Anexo 3, página 109*

5.3.4. Puntos en objeto de cinco planos

Para conseguir los puntos objetivos en los objetos tipo caja, ya sea la aplicación del interior o exterior de esta, se va a trabajar de forma parecida a la aplicación anterior. Primero se realiza una búsqueda de planos y con dicha información se consiguen los puntos necesarios.

En este caso, al tener 5 planos, podemos realizar sistemas de ecuaciones con grupos de 3 planos, de esta forma obtendremos el punto de intersección de los planos, que coincide con el que se busca.

En el sistema de ecuaciones siempre se encontrará el plano más horizontal con el suelo, es el plano en el cual se encuentran los 4 puntos.

$plano0 = \{A0, B0, C0, D0\}$ Correspondiente al plano principal

$plano1 = \{A1, B1, C1, D1\}$ Correspondiente a un lateral

$plano2 = \{A2, B2, C2, D2\}$ Correspondiente a otro lateral

$$\left\{ \begin{array}{l} A0 \cdot x + B0 \cdot y + C0 \cdot z + D0 = 0 \\ A1 \cdot x + B1 \cdot y + C1 \cdot z + D1 = 0 \\ A2 \cdot x + B2 \cdot y + C2 \cdot z + D2 = 0 \end{array} \right\}$$

Resolviendo este sistema obtenemos el punto en el sistema de coordenadas XYZ. De forma análoga teniendo siempre en el sistema de ecuaciones el plano 0 se obtienen los 3 puntos restantes.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def box_object_points(...) :'*** *en el Anexo 3, página 107*

5.3.5. Targets en objeto de 5 planos, interior de la caja

Para obtener los targets donde se va a colocar el robot, es necesario proporcionar una correcta orientación de la herramienta para no colisionar con las paredes del robot.

Se debe elegir un grado de inclinación de la herramienta teniendo en cuenta tanto las dimensiones de la caja como del propio soldador. El ángulo de inclinación a elegir es respecto a la base de la caja.

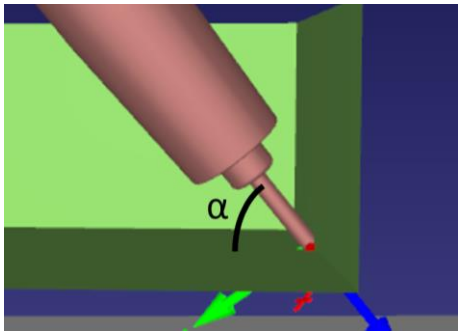


Fig 33: Inclinación herramienta plano inferior

En la figura anterior se muestra cual es el ángulo a elegir, el que forma la herramienta con el plano inferior. Una vez determinado el ángulo, se debe obtener la horientación en Z para cada uno de los 4 puntos que se corresponden con el objeto.

Conociendo los puntos de la caja, se puede calcular la longitud de las 4 aristas y el punto intermedio, es decir, el centroide de la caja en el plano inferior.

Utilizando el ángulo deseado, el centroide y los parametros del plano inferior, se calcula un punto en el espacio que servira para obtener las 4 orientaciones de la componente Z de los cuatro puntos:

$$Centroide = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad Arista = a\ (mm) \qquad Ángulo = b(rad)$$

$$plano0 = \{A0, B0, C0, D0\}$$

El punto en el espacio debe estar comprendido en la normal del plano inferior y pasar por el punto centroide, y ademas debe tener la distancia correcta para que se forme el angulo deseado.

$$puntoespacio = Centroide - \begin{pmatrix} A0 \cdot \tan(Ángulo) \cdot Arista/2 \\ B0 \cdot \tan(Ángulo) \cdot Arista/2 \\ C0 \cdot \tan(Ángulo) \cdot Arista/2 \end{pmatrix}$$

En los casos donde la caja sea cuadrada, el angulo que formara la herramienta en los vertices de la caja sera el elegido. En el caso de que la caja sea rectangular, se hara una media de las aristas, de esta forma en ciertos vertices el angulo será mayor y otros menor, siendo esta disparidad directamente relacionada con la diferencia de longitud de las aristas.

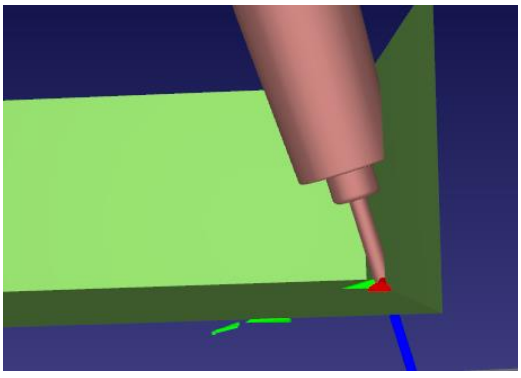


Fig 34: Colisión de la herramienta con el objeto

En la figura anterior se muestra las limitaciones que dependen principalmente de la herramienta, esta colisiona con la caja. La simulación se realizó con una inclinación de 75º del soldador respecto del plano inferior.

También dependiendo del tamaño de la caja si se eligen ángulos de incñinación menores de 40º, el soporte de la cámara podría colisionar con el objeto.

Una vez se ha obtenido el punto en el espacio que cumple los requisitos anteriores, ahora se va a calcular el target para el robot.

Primero se calcula el vector director desde el $puntoespacio$ a un punto de los vertices. De esta forma se obtiene la componente Z de orientación, para obtener la componente X e Y se realiza de manera análoga comentada anteriormente en el apartado 5.3.1.

Despues se obtiene la orientacion XYZ para los puntos restantes y se crean los diferentes target.

Finalmente se crea un target que tiene como posición, $puntoespacio$, y para la orientación se usa el vector director entre $puntoespacio$ y $Centroide$ para la componente Z, X e Y se obtienen de la misma forma que todos los puntos. Este target sirve para que el robot se aproxime y se aleje del objeto y esté correctamente orientado, de esta forma se evita que se aproxime al objeto desde un lateral colisionando con el. Desde dicha posición el robot puede comenzar la aplicación de soldadura de forma segura.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def inside_box_object_targets(...) :'*** *en el Anexo 3, página 111*

5.3.6. Targets en objeto de cinco planos, exterior de la caja

Para obtener los targets donde se va a colocar el robot, se debe elegir una orientación para la herramienta, por elección de diseño se ha optado por una orientación perpendicular al plano superior del objeto.

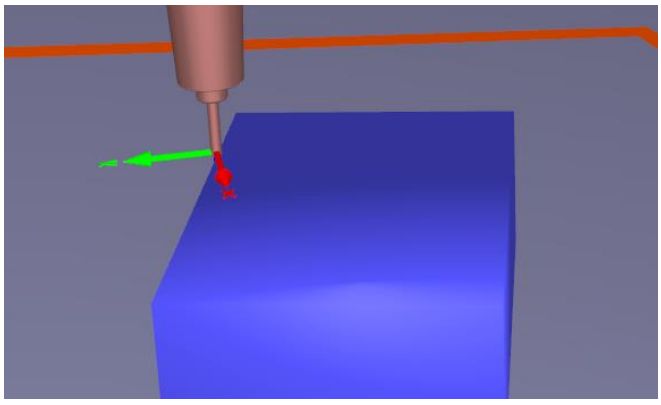


Fig 35: Inclinación de 90º de la herramienta con el plano superior

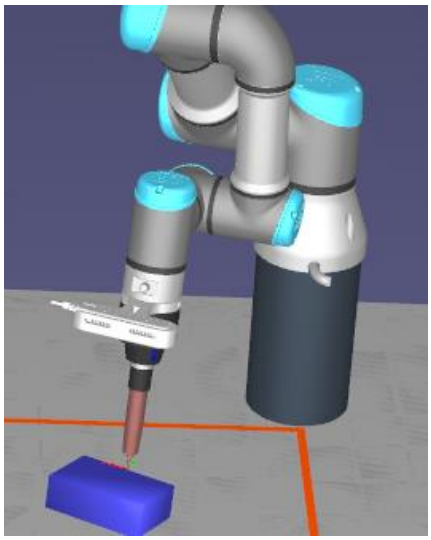
Como se puede apreciar en la figura anterior, la herramienta forma 90º con la caja.

Para obtener los targets necesarios para la soldadura, primero se obtiene la orientación de la componente Z a partir de la información del plano.

$$\overrightarrow{Vz} = \begin{pmatrix} A0 \\ B0 \\ C0 \end{pmatrix}$$

A partir de dicha componente de orientación y los 4 puntos en el espacio se obtienen las otras 2 componentes X e Y para formar dichos targets. Se realiza de manera análoga comentada anteriormente en el apartado 5.3.1.

Finalmente se crea un target para aproximarse y alejarse del robot que se obtiene a partir del primer punto de la caja desplazado 100 mm en Z y con la misma componente de orientación en Z. De esta forma como se ha realizado anteriormente, el robot se encuentra en una posición adecuada para comenzar la aplicación de soldadura de forma segura.



En la figura 36 se puede observar tanto la correcta posición de la cámara, segura y alejada del robot y la soldadura exterior de la caja, que es perpendicular al plano superior de la caja.

Fig 36: Soldando exterior de la caja

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def outside_box_object_targets(...) :'*** *en el Anexo 3, página 113*

5.3.7. Realización de trayectorias

Para realizar las trayectorias de las 3 aplicaciones anteriores, se realiza primero un acercamiento del robot al objeto con un MoveJ, Move Joint, donde el movimiento es más libre más rápido y con menos coste computacional. En la siguiente figura se muestra la posición inicial para distintas aplicaciones.

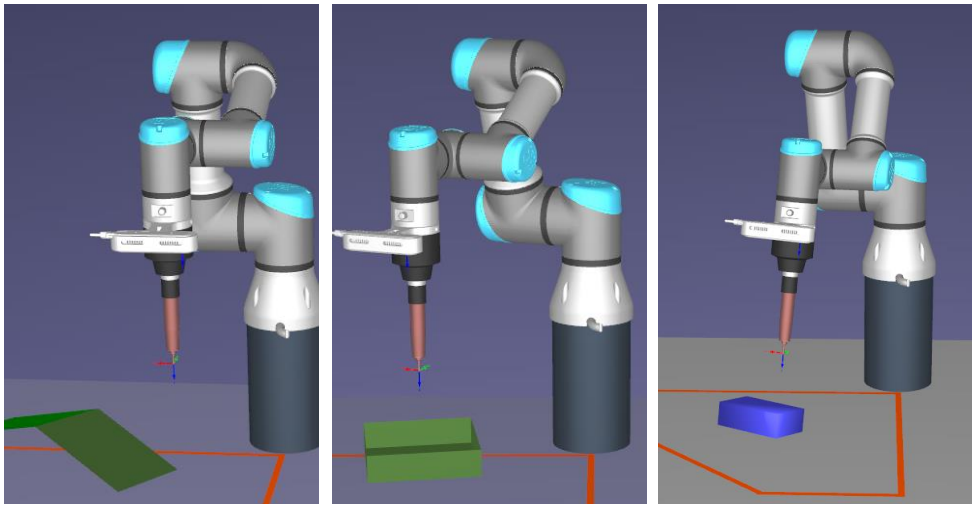


Fig 37: Posiciones iniciales del robot para distintas aplicaciones

Después durante la aplicación de soldadura se realiza MoveL, Move Lienar, alcanzando los 2 o 4 targets calculados anteriormente.

Finalmente, mediante MoveL, el robot se aleja a la posición inicial de la aplicación, se corresponde con la figura anterior.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def plane_object_aplication(...) :'*** *en el Anexo 3, página 115*

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def inside_box_object_aplication(...) :'*** *en el Anexo 3, página 116*

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def outside_box_object_aplication(...) :'*** *en el Anexo 3, página 117*

5.4. Método ICP

Para este último tipo de aplicación, se ha utilizado el algoritmo Iterative Closest Point (ICP). Dicho algoritmo es una técnica que se utiliza para alinear dos nubes de puntos tridimensionales.

Este método de soldadura, utilizando el algoritmo ICP sirve para una gran variedad de objetos, teniendo en cuenta el tamaño de este y que sea el algoritmo ICP capaz de realizar la correspondencia de las nubes.

Para el proyecto se ha elegido un objeto obtenido del software de RoboDK. En él se pueden encontrar distintos relieves y se plantea soldar las partes superiores del objeto.

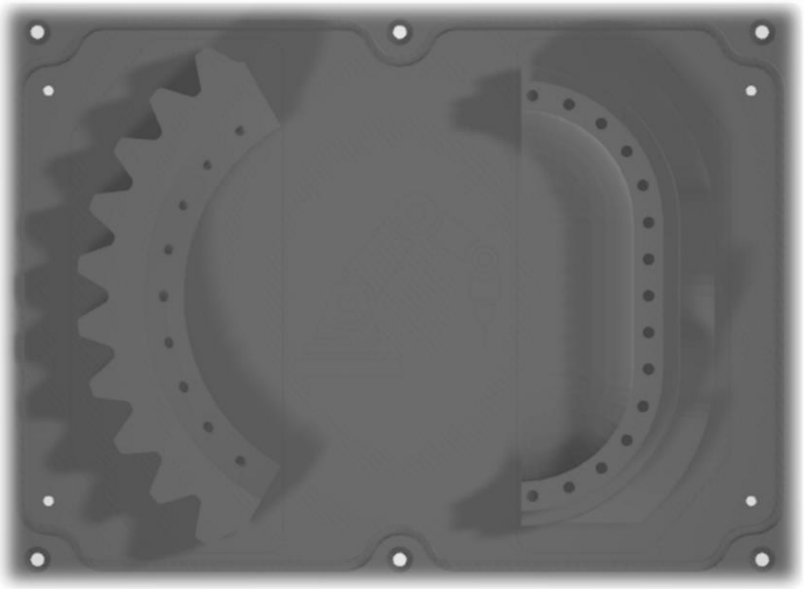


Fig 38: Objeto a identificar y soldar

Para realizar el algoritmo se debe tener un source previante guardada, que será una nube de puntos del objeto que se va a identificar. Ademas es imprescindible elegir y guardar los puntos necesarios para realizar la sodadura.

Despues se realiza la identificación del objeto de forma análoga a la aplicación de soldadura de dos planos, solo tomando una cáptura de imagen.

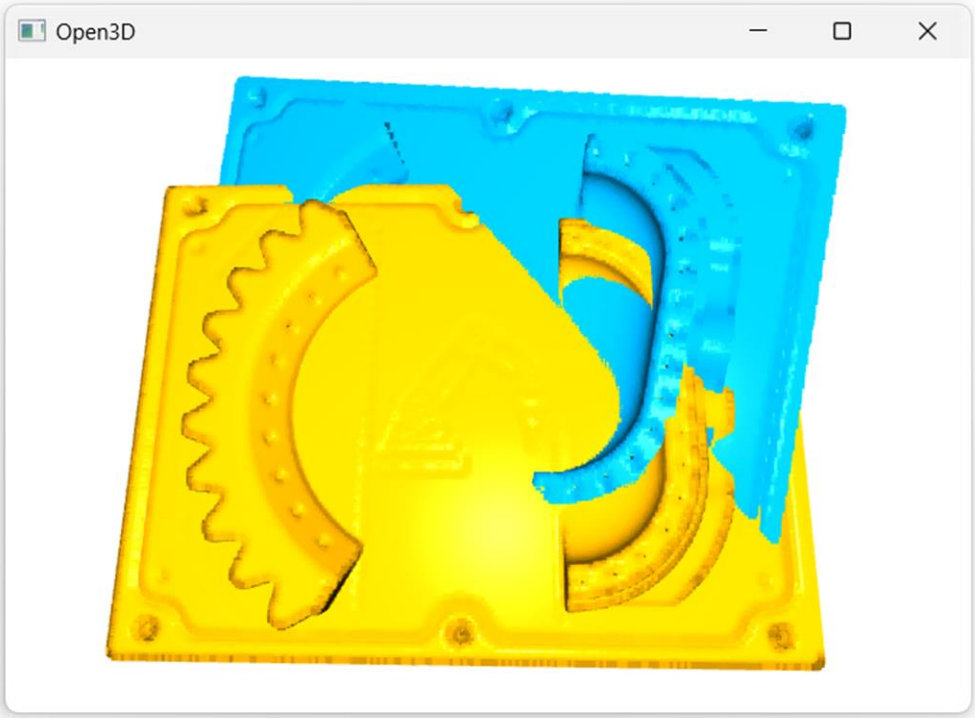


Fig 39: Diferencia entre la nube source y la nueva

En la figura anterior se muestra la nube de puntos source, la de color amarillo, y también se muestra la nube de puntos que se ha terminado de identificar, la de color azul.

Ahora se aplica al algoritmo ICP siguiendo unos pasos.

1. Inicialización: Se comienza con una transformación inicial de la nube source, se calculan los centroides de las nubes y se igualan, moviendo dicha nube source. Esta acción ayuda al algoritmo para obtener un resultado exitoso.
2. Correspondencia: Para cada punto en la nube de origen, source, se encuentra el punto más cercano en la nube destino, nueva nube. Esto crea un conjunto de pares puntos.
3. Estimación de Transformación: Con el conjunto de pares de puntos, se calcula la transformación óptima que minimiza la distancia entre los puntos.
4. Aplicación de la Transformación: Se aplica la transformación calculada a la nube source para alinearla mejor con la nueva nube.
5. Convergencia: Se repiten los pasos 2, 3 y 4 hasta que la convergencia sea alcanzada, es decir, hasta que la alineación de las nubess sea menor que un umbral o despues de un número fijo de iteraciones.

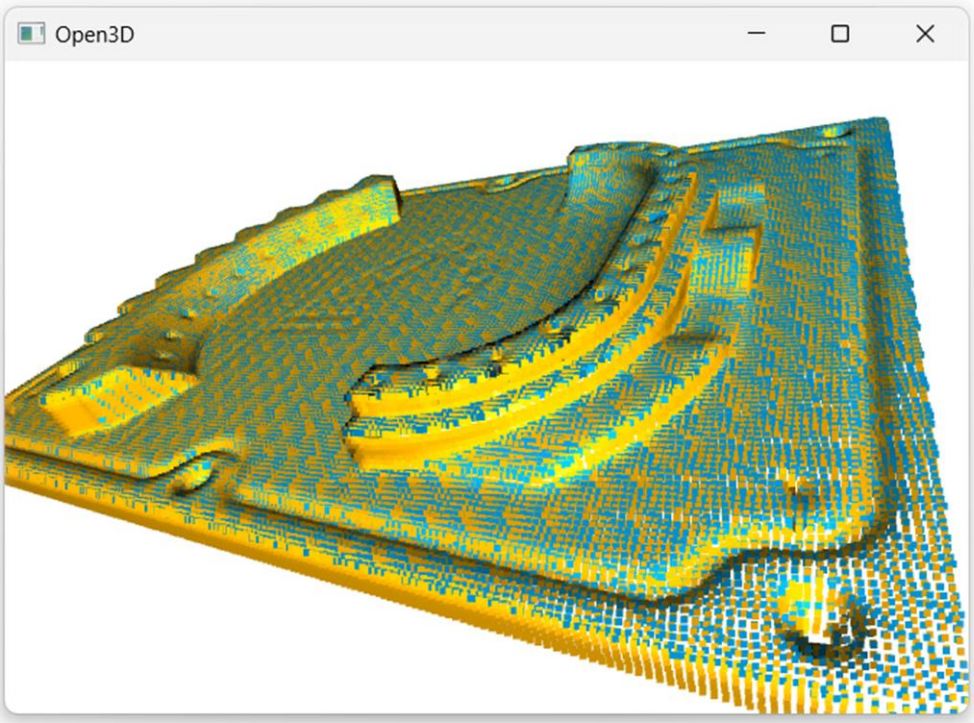


Fig 40: Acoplamiento final de las nubes

En la figura anterior se puede ver como las dos nubes quedan totalmente alineadas.

El algoritmo ICP proporciona una matriz de transformación, esta matriz servirá para transformar los puntos de interes para la soldadura de la nube source a la nueba nuve destino. A partir de los nuevos puntos se puede realizar la soldadura al objeto.

Los nuevos puntos de soldadura ya tenian la información de la componente Z de orientación, por lo tanto, se puede calcular las nuevas componentes X e Y de la misma forma comentada anteriormente en el apartado 5.3.1.

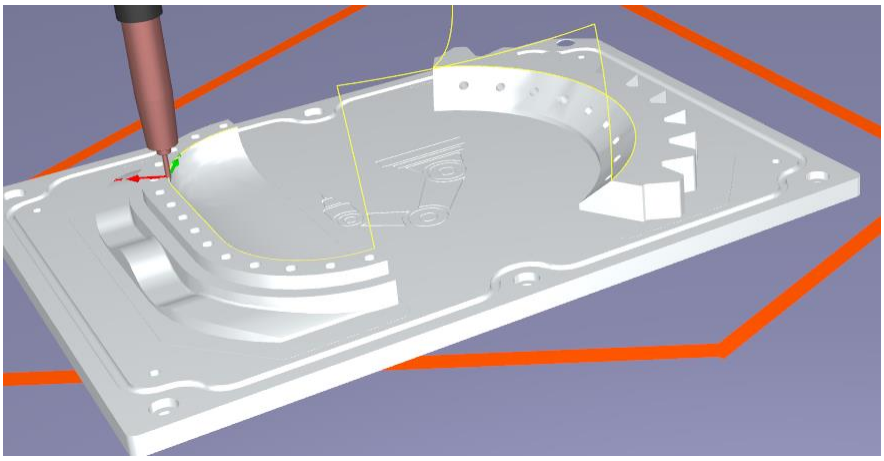


Fig 41: Soldando objeto mediante el método ICP

Finalmente aplicando distintos movimientos MoveL, MoveJ y MoveC se consigue la trayectoria deseada como se ve en la figura anterior. La línea amarilla representa la trayectoria realizada por el robot.

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def icp_identification(...) :'*** *en el Anexo 3, página 118*

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def icp_object_targets(...) :'*** *en el Anexo 3, página 119*

La implementación de lo explicado anteriormente en este apartado se encuentra en la función ***'def icp_object_aplication(...) :'*** *en el Anexo 3, página 120*

5.5. Interacción RoboDK-UR3e y distintas configuraciones y posibilidades

Para la comunicación entre el operador y el robot se han diseñado distintas interfaces con diferentes funcionalidades, tanto como para configurar la estación, el tipo de objeto esperado y hasta la posibilidad de visualizar el movimiento final antes de ejecutarlo en el robot real.

5.5.1. Interfaz de configuración

La primera interfaz solo aparece al inicio de la aplicación, sirve para configurar distintos parámetros que influirán en el comportamiento durante la ejecución de la identificación y realización del movimiento del robot.

Fig 42: Interfaz de Configuración Inicial

Se podrá seleccionar la configuración que se desee y continuar con la aplicación.

La funcionalidad de los distintos parámetros es el siguiente:

- STEP_BY_STEP: Antes de realizar la captura de una imagen aparecerá una pequeña interfaz de confirmación. Esto es útil para tener tiempo a comprobar el correcto funcionamiento de la cámara o ver si el objeto está dentro del alcance de la cámara. Una vez se este seguro de realizar la captura, se deberá pulsar el botón o eliminar la interfaz.

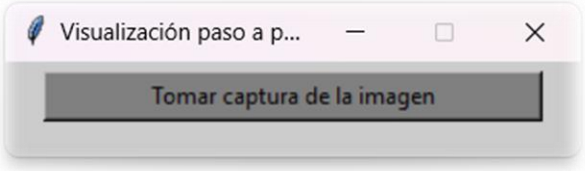


Fig 43: Interfaz de Visualización paso a paso

- VISUALIZE_PCD: Cada vez que se cree una nube de puntos de la imagen de profundidad obtenida, se mostrara por pantalla. Este parámetro resulta útil para comprobar si se está eliminando el entorno y obteniendo la información objetivo correctamente. Se podrá ver la "point cloud" de cada captura, así como la suma de todas ellas formando el objeto final.

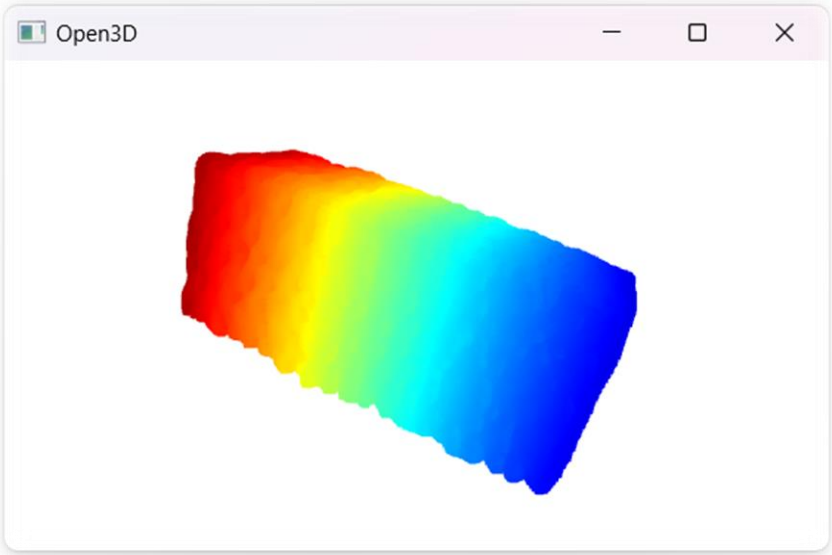


Fig 44: Visualización de la pcd del objeto

- MEHS_RECONSTRUCTION: Después de la identificación de cada objeto, se reconstruirá una malla de la nube de puntos y se colocará en la misma posición que se encuentra en la realidad, pero dentro de la estación en RoboDK. Está funcionalidad se recomienda utilizar para poder visualizar el movimiento del robot dentro de la estación y ver la aproximación y alejamiento al objeto. Una vez se haya realizado la soldadura el objeto desaparecerá. Si solo se está realizando pruebas en simulación no es de utilidad, ya que el objeto a soldar ya estará visible dentro del software.

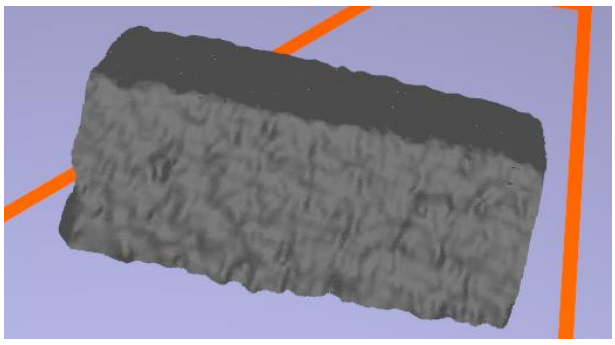


Fig 45: Reconstrucción de la pcd del objeto en RoboDK

- SAVE_PCD: La funcionalidad de este parámetro es guardar la nube de puntos en un archivo.pcd en el sistema. De esta forma se puede obtener información a posteriori de los objetos volviendo a cargar la nube de puntos.

5.5.2. Interfaz de aplicación

La segunda interfaz, la interfaz de aplicación, aparece siempre antes de comenzar a identificar el objeto y realizar la soldadura.

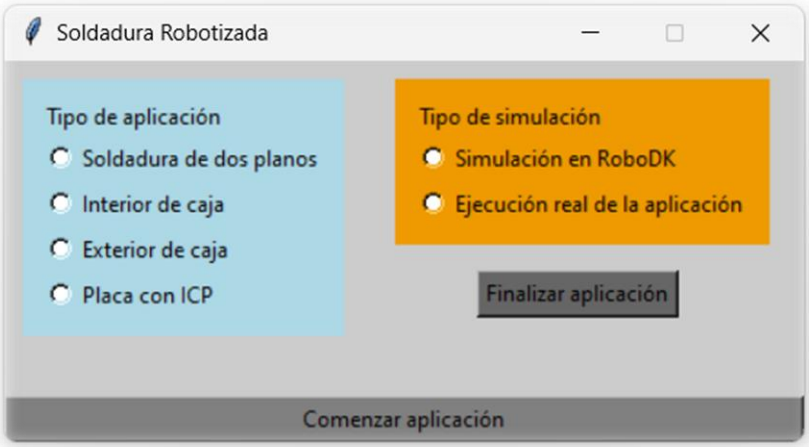


Fig 46: Interfaz de Soldadura Robotizada

En dicha interfaz se ha de determinar el tipo de aplicación, que dependerá del objeto en cuestión, y también se debe elegir el entorno, si es simulado o real.

Una vez seleccionadas las opciones deseadas se pulsa el botón de "Comenzar aplicación" y se continua con el funcionamiento del robot.

En caso de querer parar el programa se le debe pulsar "Final aplicación" o eliminar directamente la interfaz.

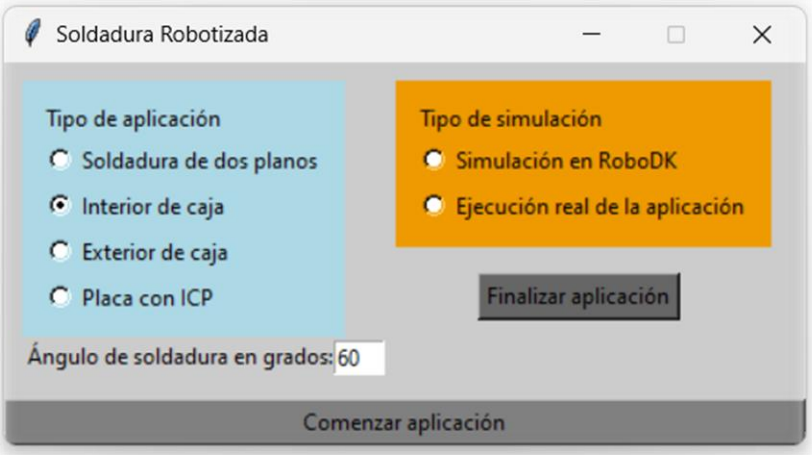


Fig 47: Interfaz de Soldadura Robotizada en opción Interior de Caja

Además, cuando se marca la opción en Tipo de aplicación, Interior de caja, aparece un recuadro de texto donde se podrá especificar el ángulo de inclinación de la herramienta a la hora de soldar.

5.5.3. Interfaz final del movimiento

La última interfaz aparece antes de realizar el movimiento de soldadura del robot.

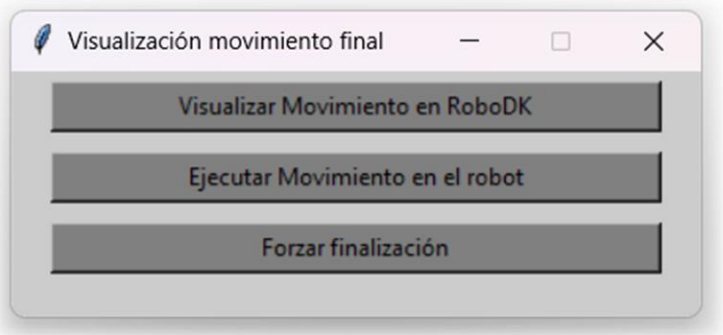


Fig 48: Interfaz de Visualización movimiento final

El primer botón "Visualizar Movimiento en RoboDK, hace que el software se desconecte momentáneamente del robot, para realizar el movimiento dentro de la estación, así el operario puede observar si el movimiento es correcto. Esta funcionalidad se puede repetir las veces que se desee con el fin de estar seguros del movimiento.

El segundo botón "Ejecutar Movimiento en el robot", sirve para continuar con el movimiento y terminar la soldadura.

El tercer botón "Forzar finalización", finaliza la aplicación sin realizar la soldadura, debido a un error observado al visualizar el movimiento en RoboDK o cualquier otro motivo.

Además, para ayudar a visualizar los puntos y las intersecciones, se han creado unos pequeños objetos que sirven para mostrar al operario información relevante.

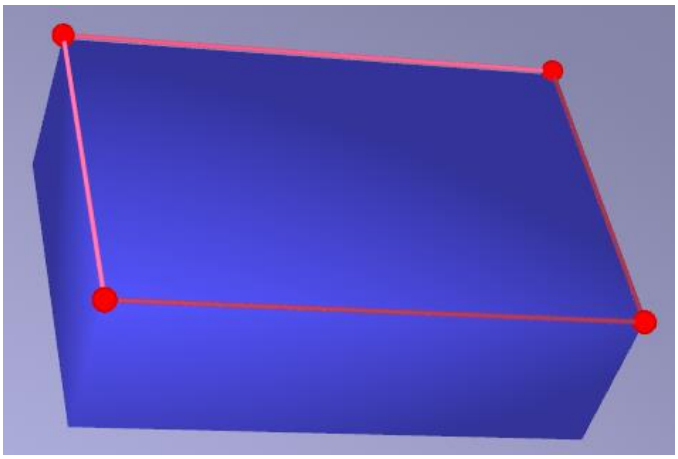


Fig 49: Indicadores de las trayectorias en el objeto

En la figura anterior se está indicando mediante esferas rojas, los puntos donde va a posicionar la herramienta el robot. La línea rosa indica la trayectoria que va a seguir el robot. Estos indicadores aparecerán también en la aplicación de soldadura de dos planos y en la aplicación interior de la caja.

Una vez realizada la soldadura o se haya forzado la finalización, los indicadores desaparecerán.

5.6. Seguridad

Se han implementado 3 capas de seguridad para garantizar un entorno seguro para los trabajadores en cuestión y para el correcto funcionamiento del robot y reducir los daños materiales que pudiera sufrir este.

- Control de colisiones en el software RoboDK:

Dentro del software RoboDK, se ha habilitado el control de colisiones. Esta funcionalidad permite simular y predecir posibles colisiones durante el movimiento del robot. El software analiza las trayectorias y genera alertas en caso de una posible colisión, de esta forma se puede parar la ejecución del movimiento antes de que suceda en el entorno real. Esta medida preventiva es efectiva para asegurar que las rutas del robot sean adecuadas y eficientes, evitando daños materiales durante la ejecución.

En el caso del presente proyecto el entorno de trabajo era bastante amplio y despejado, en otra situación se podría diseñas el entorno real dentro de RoboDK con los objetos necesarios y el robot estaría totalmente seguro gracias al control de colisiones.

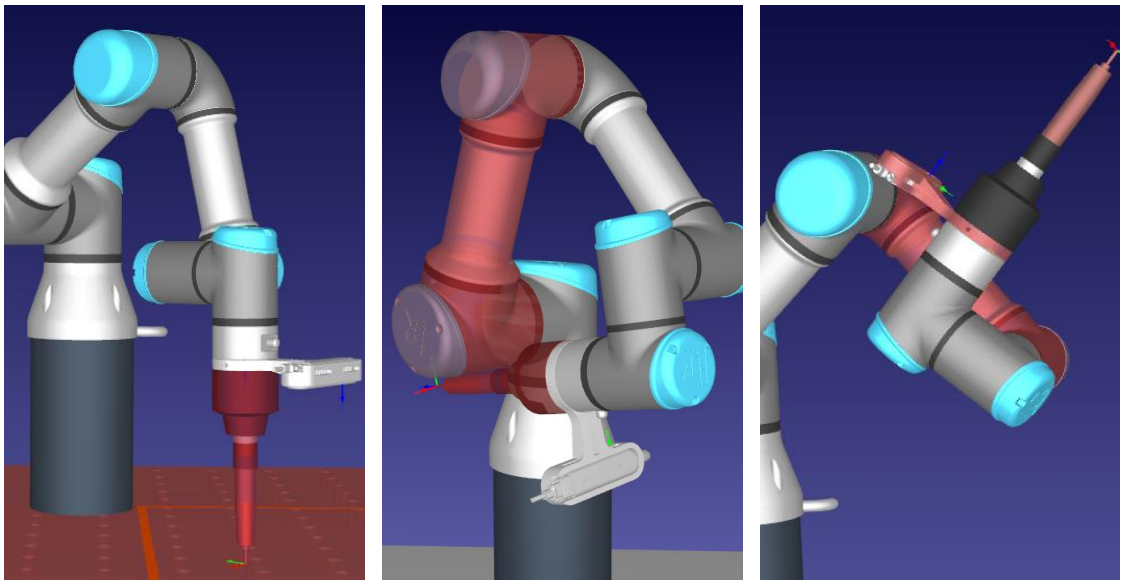


Fig 50: Distintos casos de colisiones dentro de RoboDK

En la figura anterior se muestran distintos casos. El primero nos indica que la herramienta va a colisionar con la mesa. En el segundo caso nos alerta que la herramienta va a colisionar con un eslabón del robot. En el último caso se indica que la cámara va a chocar con un eslabón.

Cabe mencionar que en toda la aplicación la configuración del robot, es decir, como se posicionan sus articulaciones, se ha planteado siempre de la misma forma, gracias a ello se han podido elegir las mejores formas de generar las trayectorias y con la mayor seguridad.

Además, como los objetos están en la mesa, la herramienta apunta siempre hacía el suelo y lo más alejada del robot posible, por lo que la última imagen sería difícil que ocurriese.

- Detector de manos mediante visión por computador:

Se ha integrado un sistema de detección de manos utilizado el sensor RGB que tiene la cámara Intel Realsense. Se usa un algoritmo de detección de manos basado en MediaPipe. Cuando se está moviendo el robot, el detector monitorea continuadamente el área de trabajo, identificando la presencia de manos humanas en tiempo real.

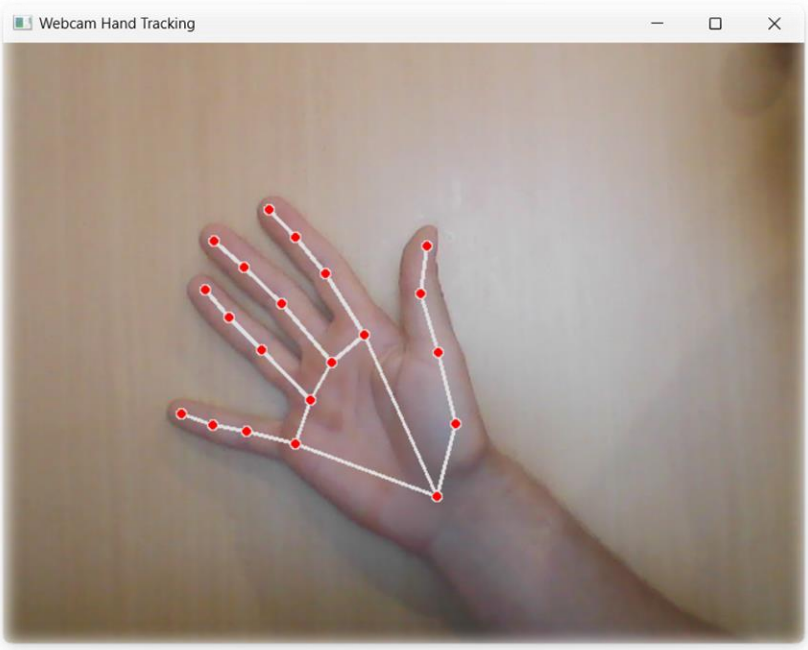


Fig 51: Detección de mano

Si se detecta una mano desde la cámara, significa que hay un humano cerca y podría suponer un riesgo, por lo tanto, el software detiene el robot y aparece un mensaje alertando de la aparición de una mano y preguntando si desea continuar una vez se está seguro.

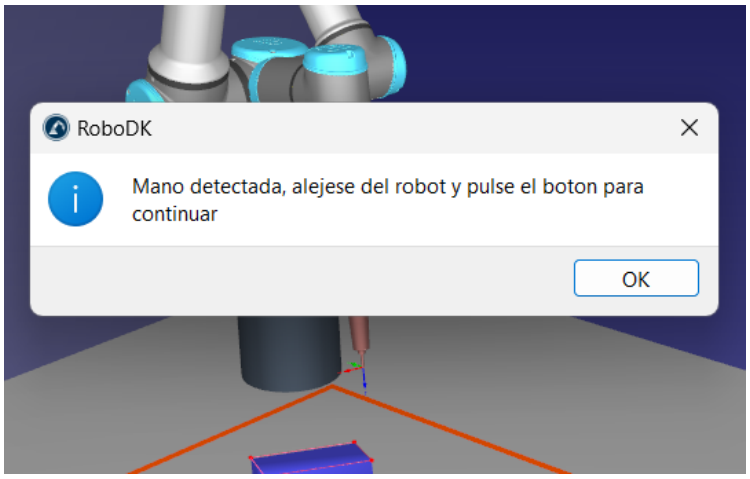


Fig 52: Mensaje de alerta

Este monitoreo para las manos solo se ejecuta cuando el robot está identificando el objeto o realizando la soldadura, cuando el operario está eligiendo el tipo de objeto mediante la interfaz principal no se ejecuta la detección de manos. De esta forma se puede acceder al área de trabajo y cambiar el objeto o su posición sin la aparición del mensaje de seguridad.

- Sensores de Fuerza del Robot UR3e:

El Robot Ur3e está equipado con sensores que monitorean la fuerza y el torque en cada una de sus 6 articulaciones. Cuando dichos sensores detectan una fuerza externa que supera cierto umbral, el robot detiene su operación de inmediato. Con esta medida se trata de minimizar el riesgo de colisiones o la destrucción total del robot al intentar atravesar una pared u objeto rígido.

Esta medida de seguridad está integrada en el robot y para desbloquear el robot se deberá realizar desde el PolyScope del robot, la interfaz mostrada en la pantalla táctil creada por el fabricante Universal Robots.

6. JUSTIFICACIÓN DETALLADA DE LA SELECCIÓN O DIMENSIONAMIENTO DE ELEMENTOS

6.1. Elementos alternativos

Para el proyecto se valoró el material disponible en el laboratorio y se descarto el siguiente elemento.

6.1.1. Robot ABB IRB 140

El robot ABB IRB 140 es un robot industrial antropomórfico fabricado por ABB, empresa líder en robótica y automatización. Diseñado para aplicaciones repetitivas de manipulación y ensamblaje en entornos industriales, el IRB 140 es compacto, versátil y altamente preciso.

Este modelo de ABB tiene una carga útil máxima de 6 kg y un alcance máximo de 810 mm, lo que le permite manejar una gran variedad de tareas de montaje y manipulación en espacios reducidos. Su estructura de brazo articulado consta de seis ejes de movilidad que le brindan una gran flexibilidad de movimiento y permiten un posicionamiento y orientación preciso en el espacio de trabajo.

Fig 53: Robot ABB IRB 140

El IRB 140 cuenta con una serie de características y tecnologías avanzadas que mejoran su rendimiento y eficiencia. Equipado con tecnologías avanzadas y opciones personalizables, el IRB 140 es una solución eficiente para automatizar procesos en diferentes industrias.

El robot IRB 140 es una opción válida para casi cualquier ámbito industrial, pero en el caso del proyecto debido a él gran tamaño del robot y la posición de él en el laboratorio, dificulta su uso, no siendo el mejor en este caso donde se va a trabajar en conjunto con más personas relativamente cerca al robot.

6.2. Elementos seleccionados

Los elementos que finalmente se han seleccionado tiene distintas funcionalidades y características que cumplen con los requisitos del proyecto. A los siguientes elementos también se deben incorporar los objetos diseñados con la impresora 3D y objetos de prueba, principalmente eran cajas de cartón.

6.2.1. Robot colaborativo UR3e

El robot UR3e es un robot colaborativo de la serie e-Series fabricado por Universal Robots. Está diseñado para trabajar de forma segura y eficiente junto a personas, el UR3e es compacto y versátil. Es útil y eficiente para gran número de aplicaciones en entornos industriales y de laboratorio.

El UR3e tiene una capacidad de carga útil de 3 kg y un alcance máximo de 500 mm, lo que le permite manipular objetos con precisión en áreas de trabajo más pequeñas. Con seis grados de libertad, el robot puede moverse de manera flexible y alcanzar diferentes posiciones y ángulos.

Está equipado con sensores y tecnología de detección de fuerza que le permiten interactuar de forma segura con los operadores humanos sin necesidad de barreras físicas de seguridad.

Fig 54: Robot UR3e

A diferencia con el robot ABB IRB 140, el UR3e es de menor tamaño y mas seguro a la hora de realizar tareas colaborando con personas, ya que estas van introduciendo y extrayendo los objetos. Además, gracias a las características del efector final es muy sencillo diseñar nuevas herramientas desde cero, como en el caso del proyecto.

6.2.2. Intel Realsense D415

La cámara Intel RealSense D415 es una cámara de profundidad que ofrece alta precisión en aplicaciones como el escaneo 3D. Utiliza un obturador giratorio (Rolling shutter) en el sensor de profundidad, lo que proporciona una calidad de profundidad superior por grado. Además, cuenta con un sensor RGB integrado, lo que la hace adecuada para autenticación facial, escaneo 3D y captura volumétrica.

Esta cámara es ligera, potente y de bajo coste, lo que la convierte en una opción ideal para el desarrollo y la producción de soluciones de detección. Puede integrarse fácilmente gracias a su software personalizable, lo que permite la creación de dispositivos que comprenden e interactúan mejor con su entorno.

La cámara D415 ofrece un sensor de profundidad con una resolución de hasta 1280x720 y una frecuencia de 90 fps. El sensor RGB integrado cuenta con una resolución de 1920x1080 a 30 fps.

La D415 es adecuada para el proyecto, donde se va a utilizar su tecnología para capturar imganees de profundidad con cierta. Además, gracias a un pequeño tamaño y forma alargada es fácilmente integrable en la herramienta del robot

Fig 55: Intel Realsense D415

6.2.3. Quick changer tool OnRobot

Se ha utilizado un intercambiador de herramientas, el modelo QC-R v2 de OnRobot. El intercambiador es una solución diseñada para mejorar la flexibilidad y eficacia de los robot colaborativos, en el proyecto no a sido necesario el uso de dos herramientas, pero permitía una rápida colocación del soporte con la cámara, esto resulta útil a la hora de hacer pruebas y se necesita montar y desmontar varias veces.

Fig 56: Intercambiador de herramientas

7. PRUEBAS Y RESULTADOS

7.1. PRUEBAS EN SIMULACIÓN

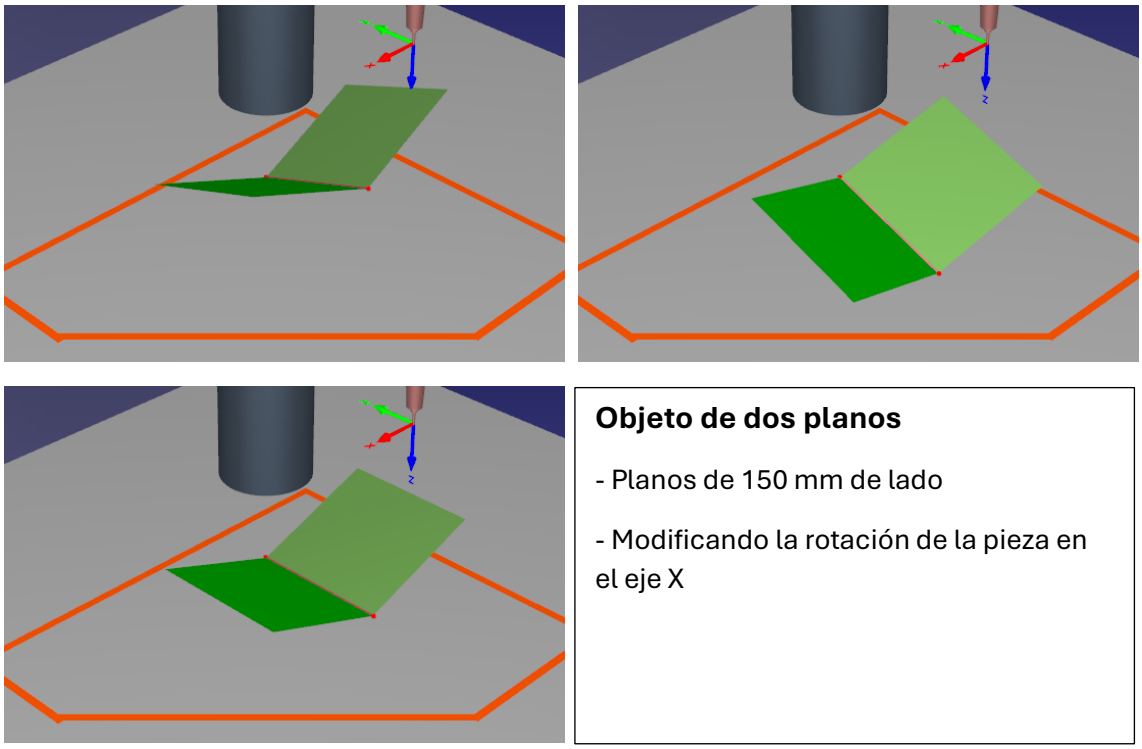
Se realizaron pruebas en simulación con objetos de distintos tamaños y en múltiples posiciones. En el software RoboDK las condiciones son perfectas y el error producido por la cámara o el robot es mínimo.

Antes de comenzar es preciso resaltar las limitaciones mecánicas del robot y la cámara, que son la principal limitación, en ciertas ocasiones es imposible que el robot realice la trayectoria debido a su propio alcance o estructura y otras veces debido a la cercanía del objeto a la cámara, esta no es capaz de identificarlo.

Para la simulación se usaron distintos objetos con distintas posiciones y orientaciones. Las siguientes imágenes son capturas de pantalla durante la simulación donde se puede apreciar como la trayectoria a seguir está correctamente definida por líneas rosas y puntos rojos.

7.1.1. Intersección de dos planos

- Pruebas exitosas de soldadura en la intersección de dos planos:

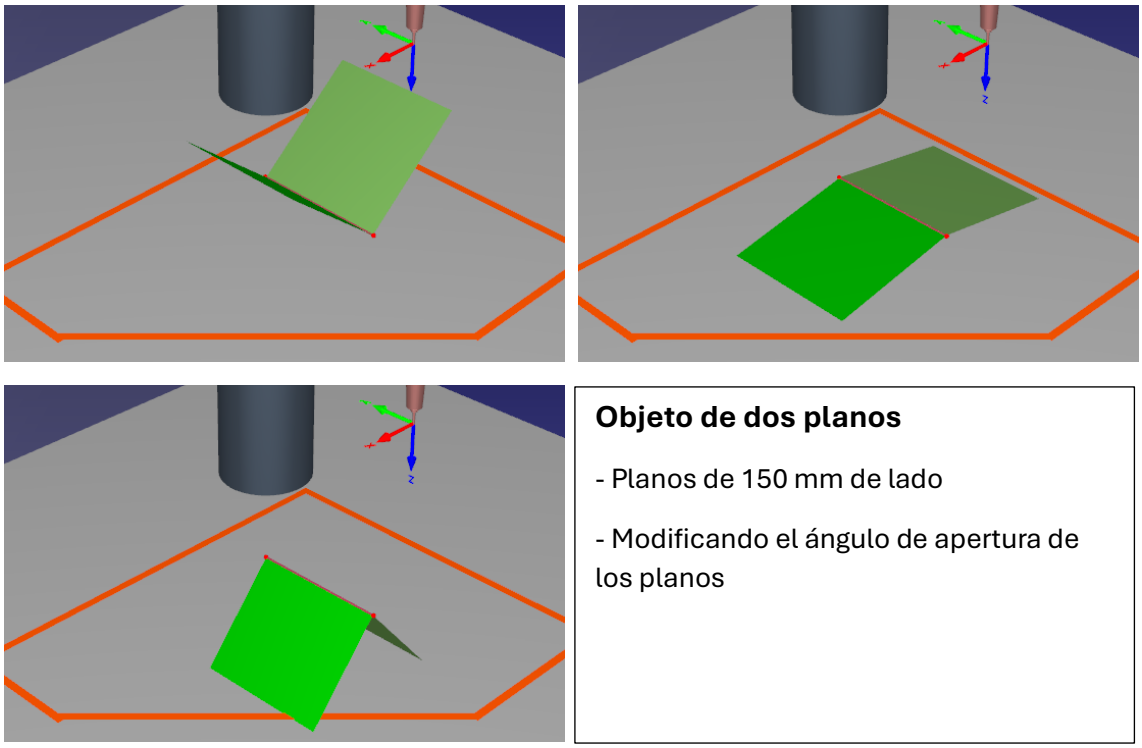


Objeto de dos planos

- Planos de 150 mm de lado

- Modificando la rotación de la pieza en el eje X

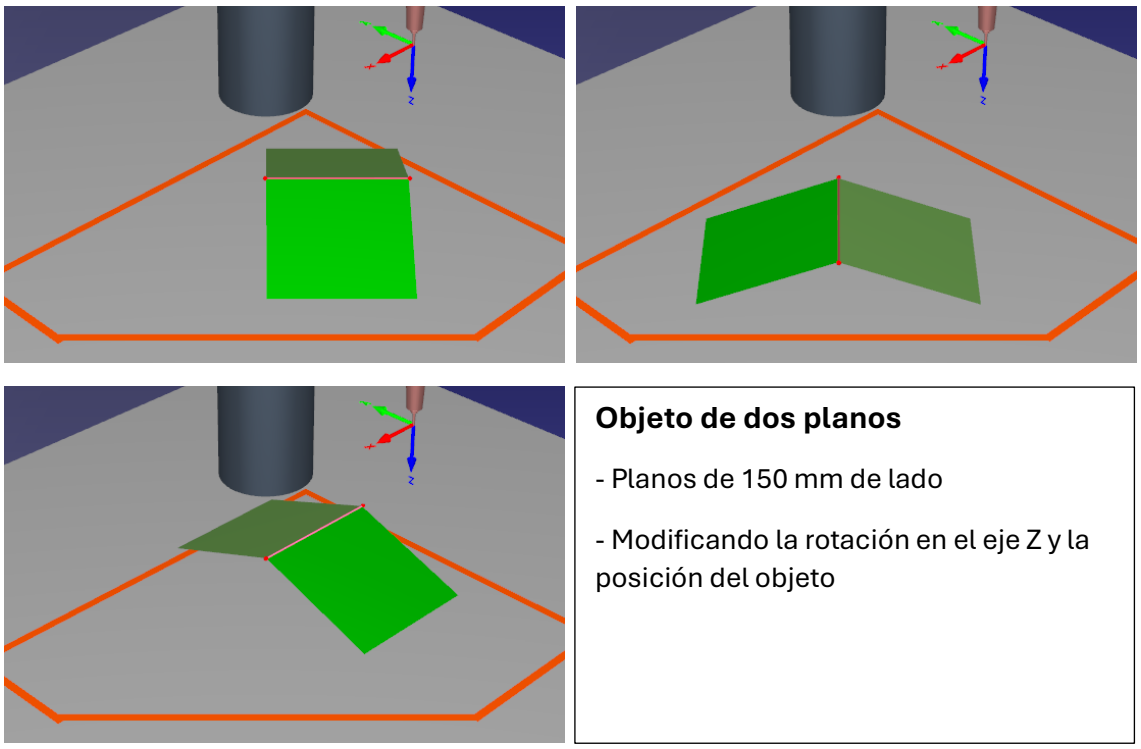
Fig 57: Pruebas exitosas en objeto de dos planos 1



Objeto de dos planos

- Planos de 150 mm de lado
- Modificando el ángulo de apertura de los planos

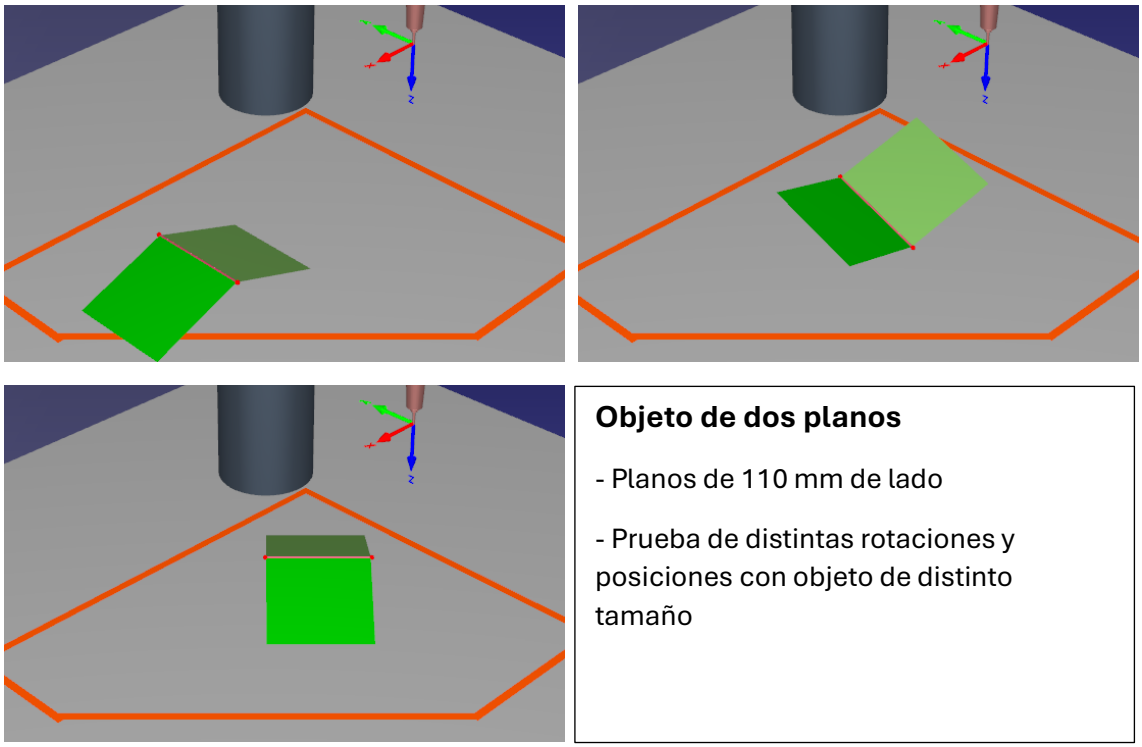
Fig 58: Pruebas exitosas en objeto de dos planos 2



Objeto de dos planos

- Planos de 150 mm de lado
- Modificando la rotación en el eje Z y la posición del objeto

Fig 59: Pruebas exitosas en objeto de dos planos 3



Objeto de dos planos

- Planos de 110 mm de lado
- Prueba de distintas rotaciones y posiciones con objeto de distinto tamaño

Fig 60: Pruebas exitosas en objeto de dos planos 4

- Limitaciones en la soldadura de la intersección de dos planos:

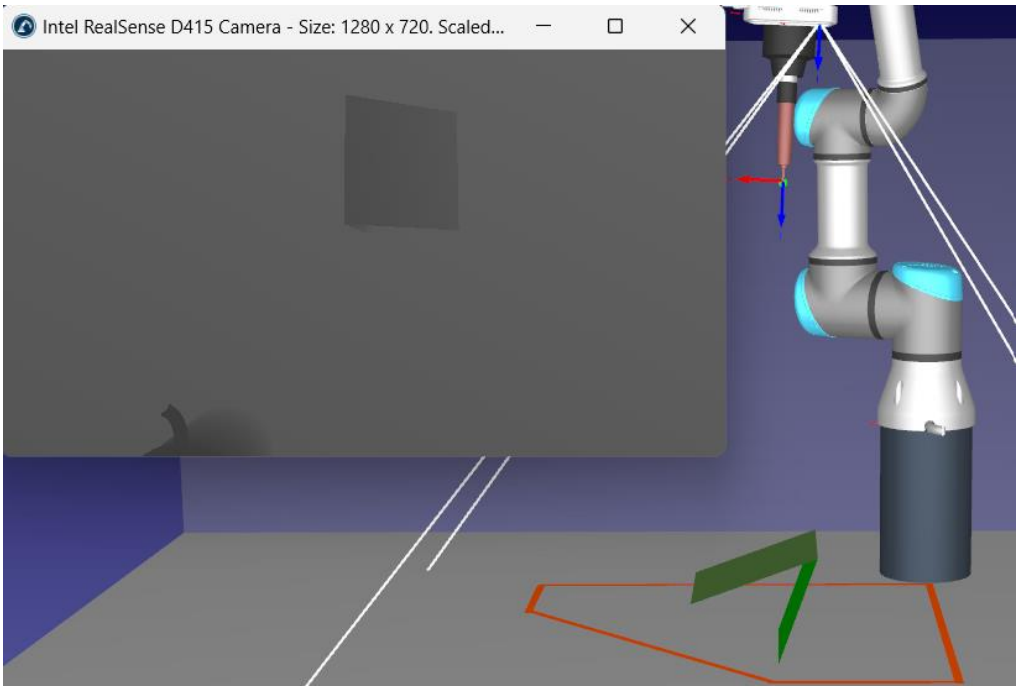


Fig 61: Pruebas con vista de la cámara

Esta es una prueba fallida, donde el objeto está de una forma muy poco intuitiva para realizar la identificación y la soldadura. En la vista de la cámara se puede apreciar una pequeña parte del plano inferior y es posible reconocer el plano y obtener la trayectoria.

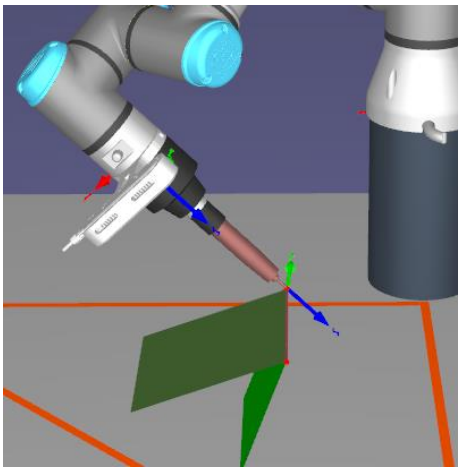


Fig 62: Orientación errónea

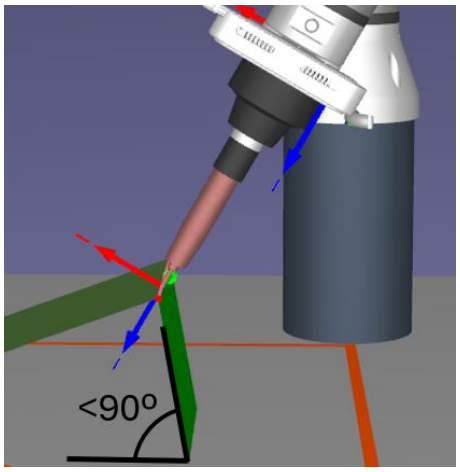


Fig 63: Orientación correcta

Pero se puede observar en la figura 62 que la orientación de la herramienta no es la correcta. Esto es debido a la forma de calcular dicha orientación. Una de las limitaciones en este tipo de objeto es que los planos no deben sobrepasar los 90º de inclinación con respecto a la horizontal.

En la figura 63 se ha tenido en cuenta la limitación y no se ha sobrepasado los 90º de inclinación entre el plano y la horizontal, se ha tomado de referencia la mesa porque está horizontal al suelo. En este caso la soldadura se puede realizar sin ningún problema.

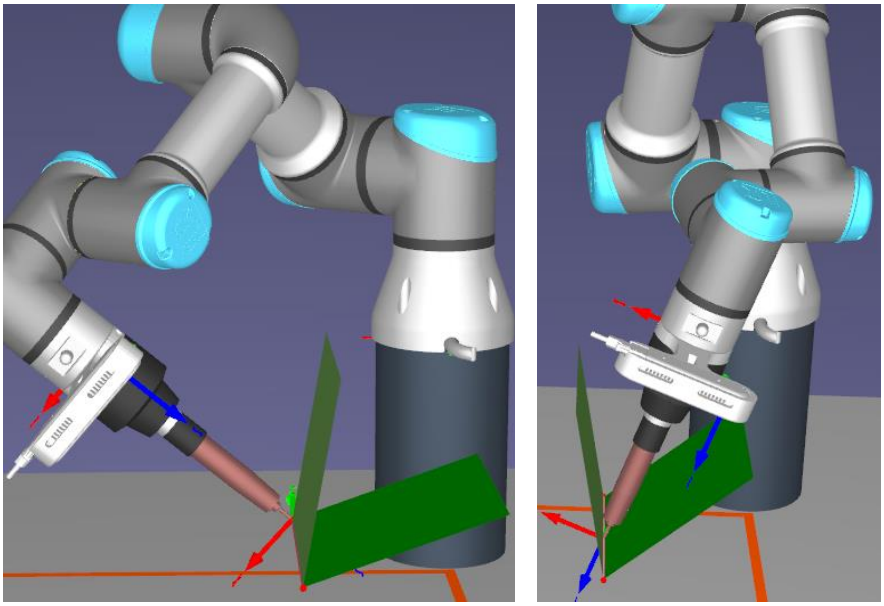
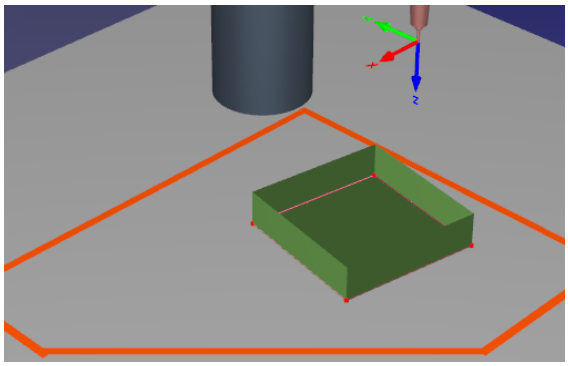
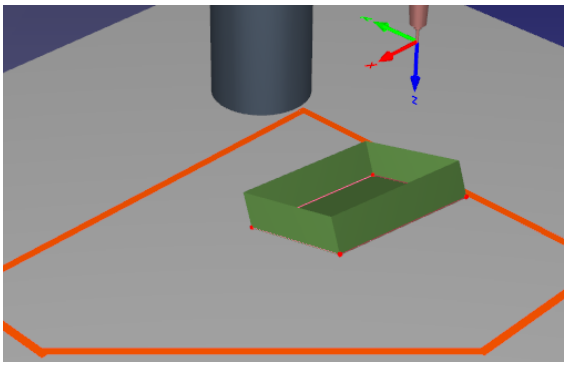
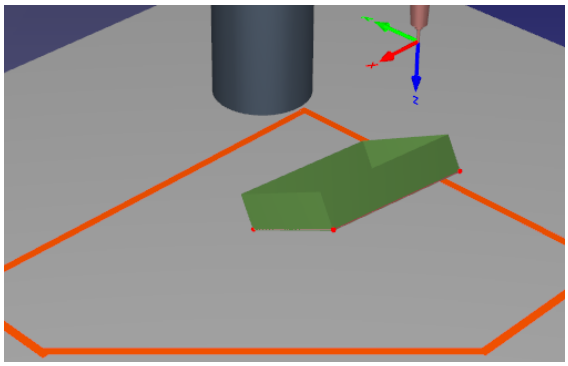


Fig 64: Ejemplo de resolver el problema

En esta última figura se puede apreciar la limitación con otro ejemplo y la forma de resolver el problema. La pieza es la misma, solo se ha modificado su orientación, de esta forma si es realizable la soldadura.

7.1.2. Interior de una caja

- Pruebas exitosas en la soldadura del interior de una caja:

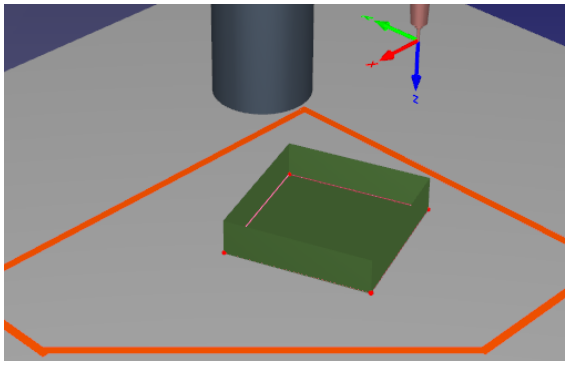
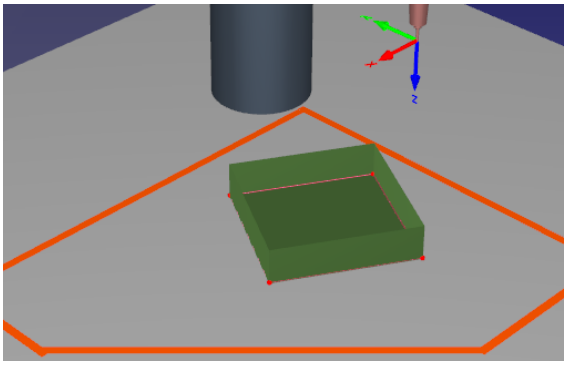
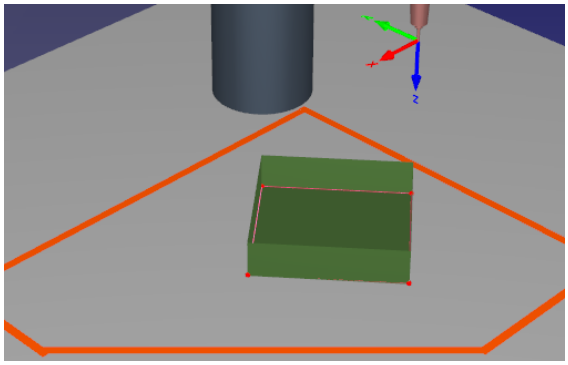




Interior de caja

- Caja de 150 mm de lado y 37.5 mm de altura

- Modificando la rotación de la caja en el eje X

Fig 65: Pruebas exitosas en interior de caja 1

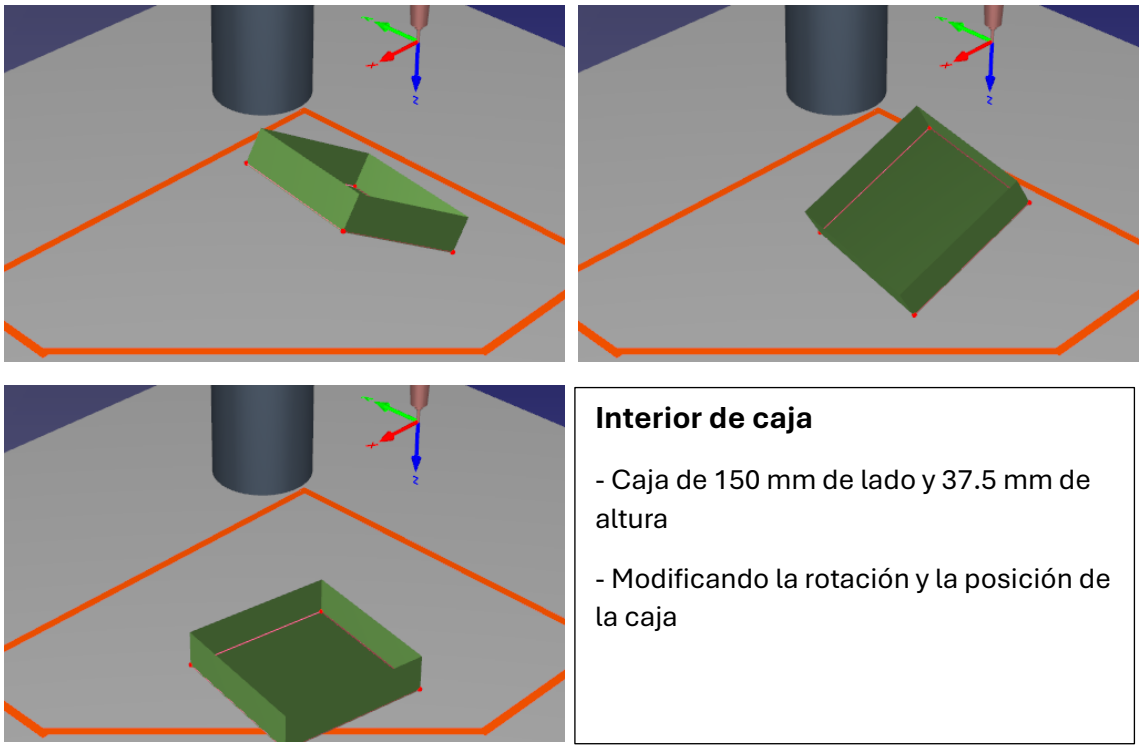




Interior de caja

- Caja de 150 mm de lado y 37.5 mm de altura

- Modificando la rotación de la caja en el eje Z

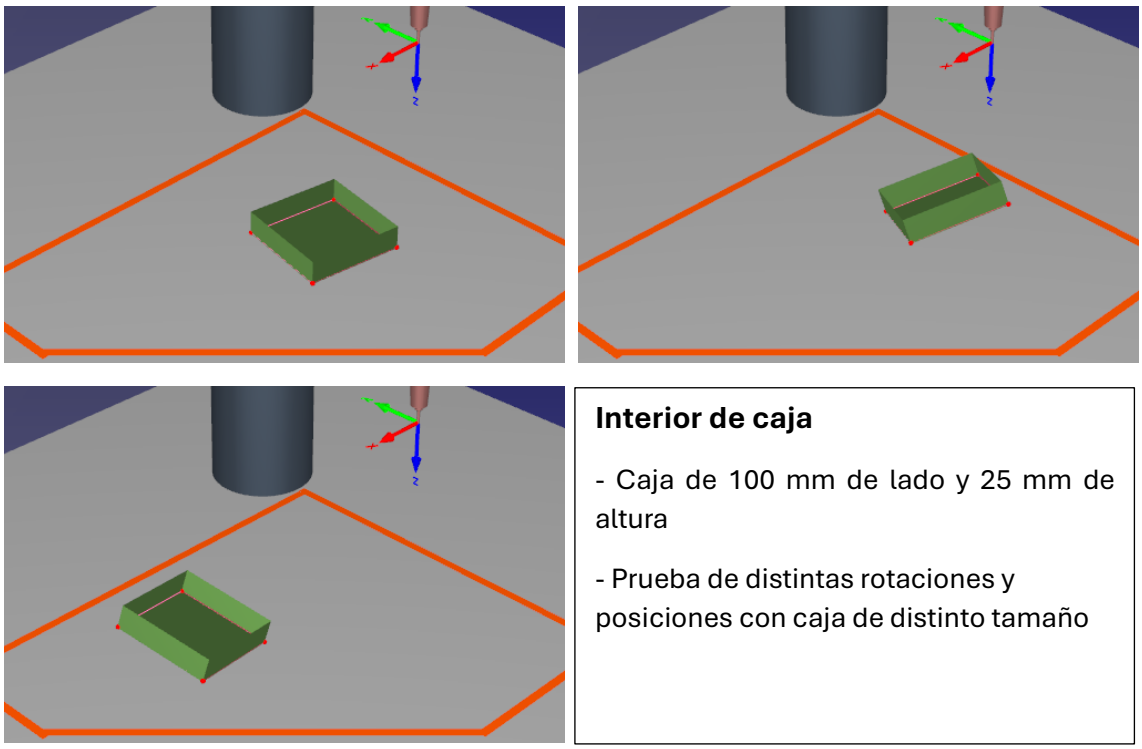
Fig 66: Pruebas exitosas en interior de caja 2



Interior de caja

- Caja de 150 mm de lado y 37.5 mm de altura
- Modificando la rotación y la posición de la caja

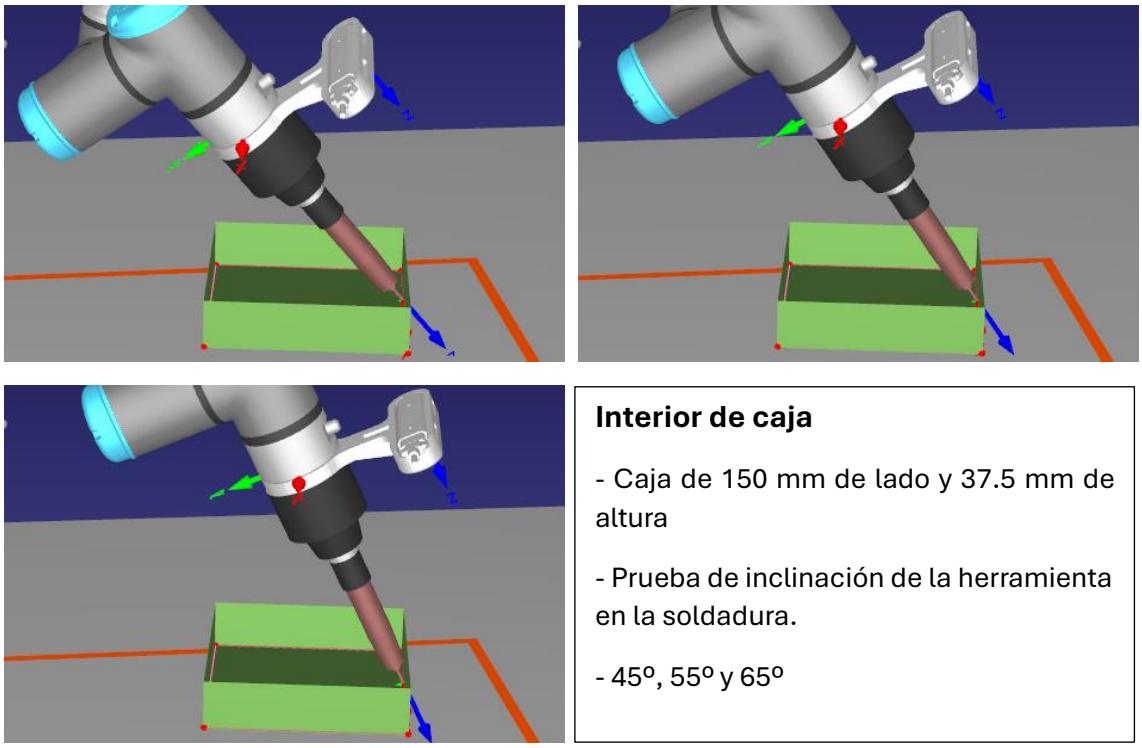
Fig 67: Pruebas exitosas en interior de caja 3



Interior de caja

- Caja de 100 mm de lado y 25 mm de altura
- Prueba de distintas rotaciones y posiciones con caja de distinto tamaño

Fig 68: Pruebas exitosas en interior de caja 4



Interior de caja

- Caja de 150 mm de lado y 37.5 mm de altura

- Prueba de inclinación de la herramienta en la soldadura.

- 45º, 55º y 65º

Fig 69: Pruebas exitosas en interior de caja 5

- Limitaciones en la soldadura del interior de una caja:

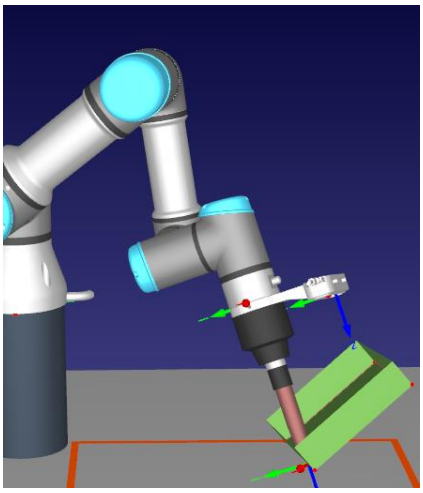


Fig 70: Inclinación de 40º del objeto

La primera limitación es similar a lo que sucede con la intersección de dos planos. En este caso el plano inferior no debe superar los 45º respecto a la horizontal. Si supera esa inclinación el plano que se tomará como inferior será otro y los cálculos serán erróneos.

En la figura de la derecha se muestra la caja con una inclinación de 40º sobre el eje x, en ese caso es realizable el cálculo de las trayectorias y su posterior soldadura.

La segunda limitación, se debe a que la caja no debe estar perfectamente alineada, es decir con una rotación en Z de 0º o 180º, esto produce errores a la hora de ordenar los puntos para que tenga sentido la trayectoria.

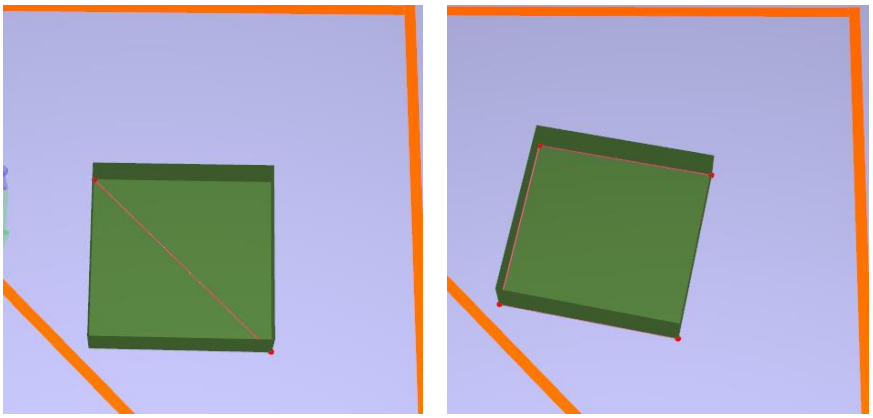


Fig 71: Ejemplos de rotación de la caja (interior)

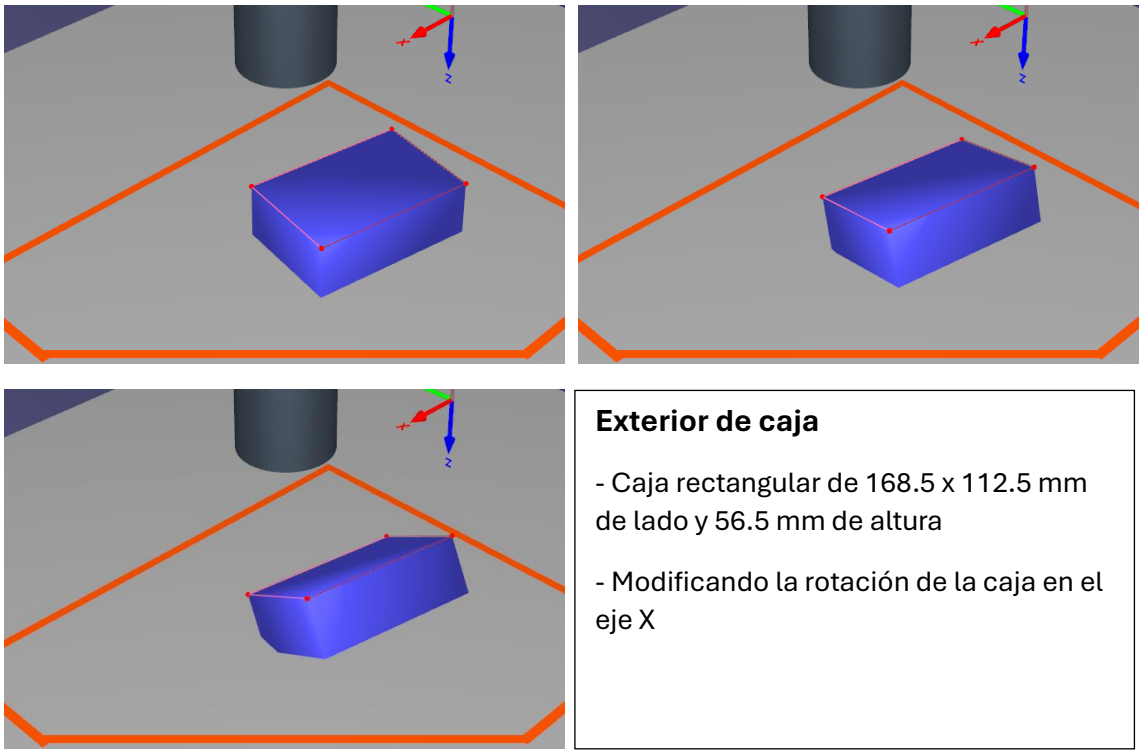
En la figura anterior se puede apreciar primero que la caja está totalmente alineada, las marcas naranjas de la parte superior y derecha están alineada con los ejes X e Y respectivamente. En este caso no se han podido identificar el orden de los puntos y por ello la trayectoria no es correcta.

Después se ha rotado la caja 10º en el eje Z y se puede apreciar que la trayectoria esta perfectamente calculada.

La última limitación es debida al grado de inclinación con el que se desea realizar la soldadura en la intersección, como se comentó en el punto 5.3.5. este grado de inclinación dependerá tanto del tamaño y forma de la herramienta de soldadura como del tamaño de la caja. Se ha obtenido un rango seguro de entre 45 y 65 º de inclinación, pero al final deberá ser el ingeniero al cargo el que decida la inclinación óptima.

7.1.3. Exterior de una caja

- Pruebas exitosas en la soldadura del interior de una caja:

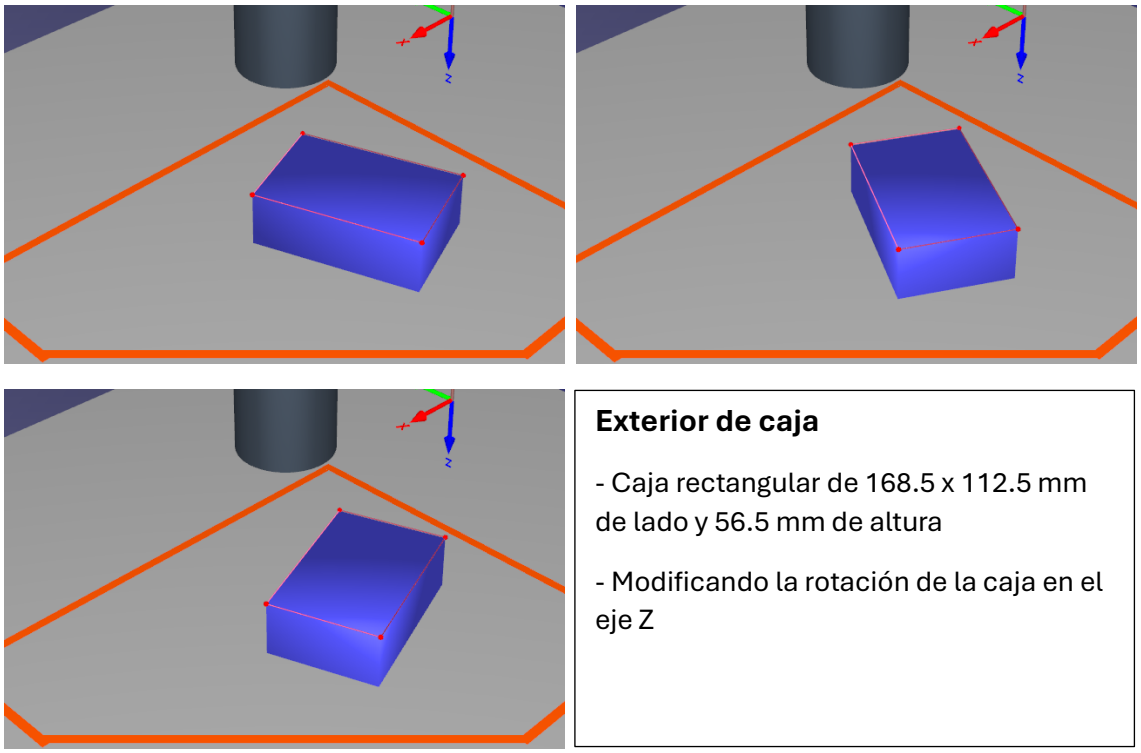


Exterior de caja

- Caja rectangular de 168.5 x 112.5 mm de lado y 56.5 mm de altura

- Modificando la rotación de la caja en el eje X

Fig 72: Pruebas exitosas en exterior de caja 1

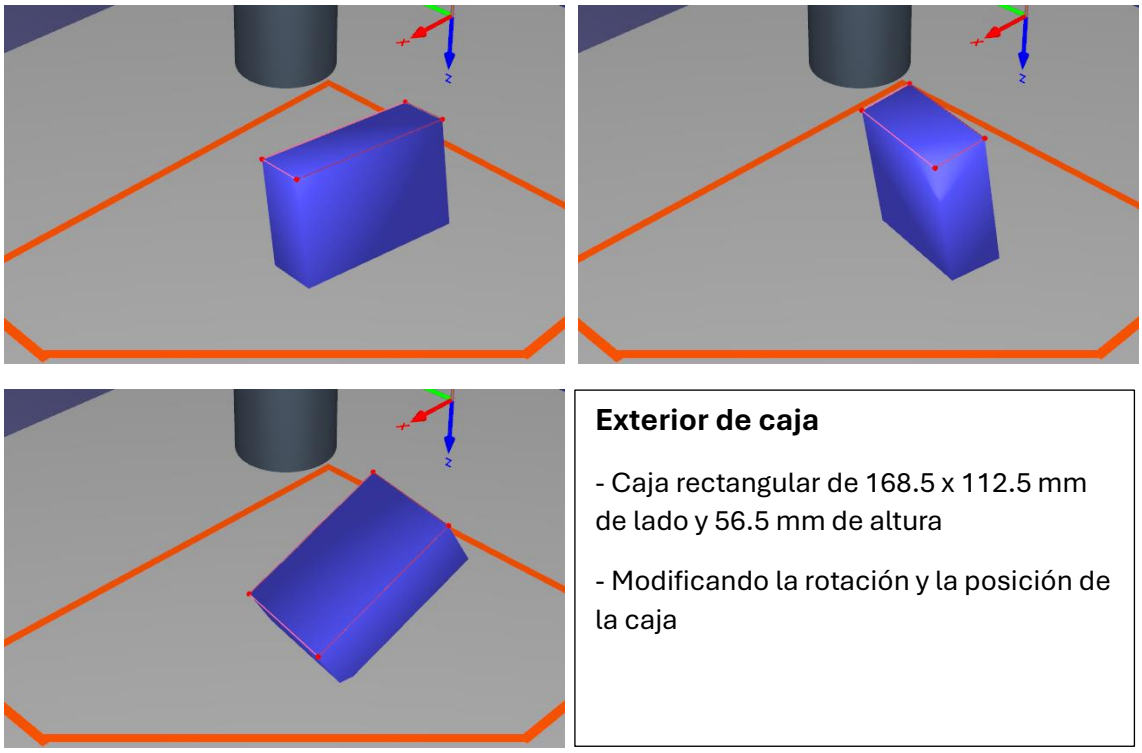


Exterior de caja

- Caja rectangular de 168.5 x 112.5 mm de lado y 56.5 mm de altura

- Modificando la rotación de la caja en el eje Z

Fig 73: Pruebas exitosas en exterior de caja 2

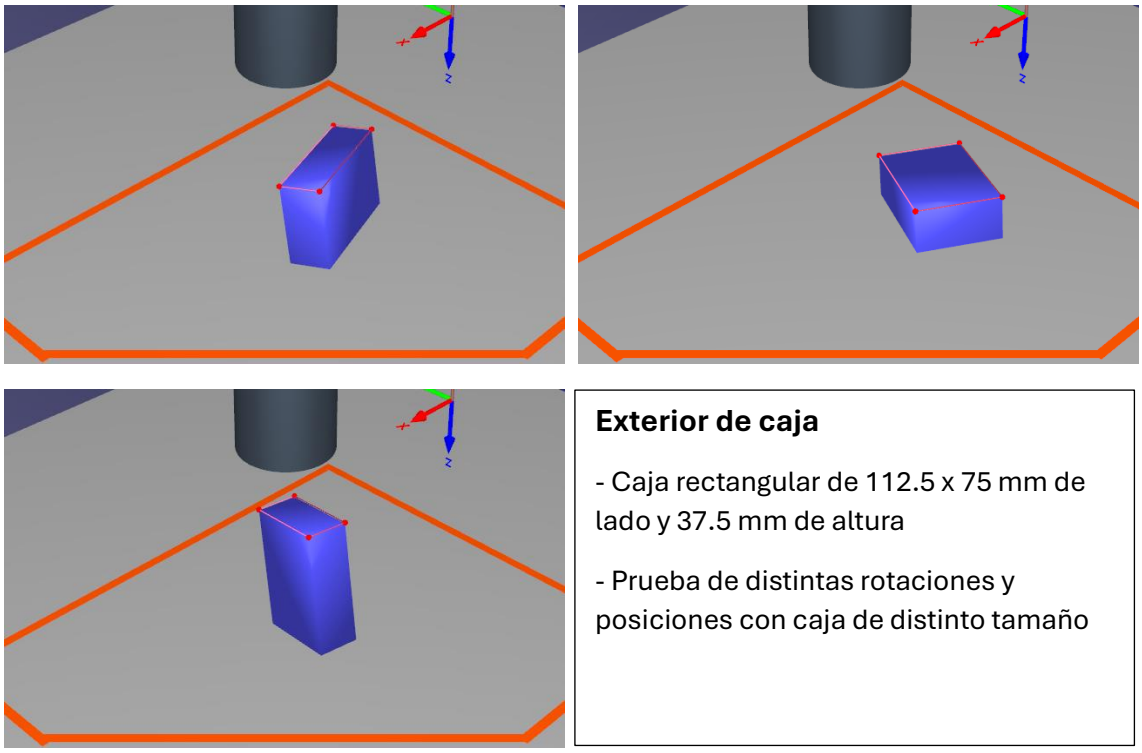


Exterior de caja

- Caja rectangular de 168.5 x 112.5 mm de lado y 56.5 mm de altura

- Modificando la rotación y la posición de la caja

Fig 74: Pruebas exitosas en exterior de caja 3



Exterior de caja

- Caja rectangular de 112.5 x 75 mm de lado y 37.5 mm de altura

- Prueba de distintas rotaciones y posiciones con caja de distinto tamaño

Fig 75: Pruebas exitosas en exterior de caja 4

- Limitaciones en la soldadura del interior de una caja:

La primera limitación no es en sí una limitación sino un detalle a tener en cuenta. La cara que se desee soldar deberá estar con una inclinación menor de 45

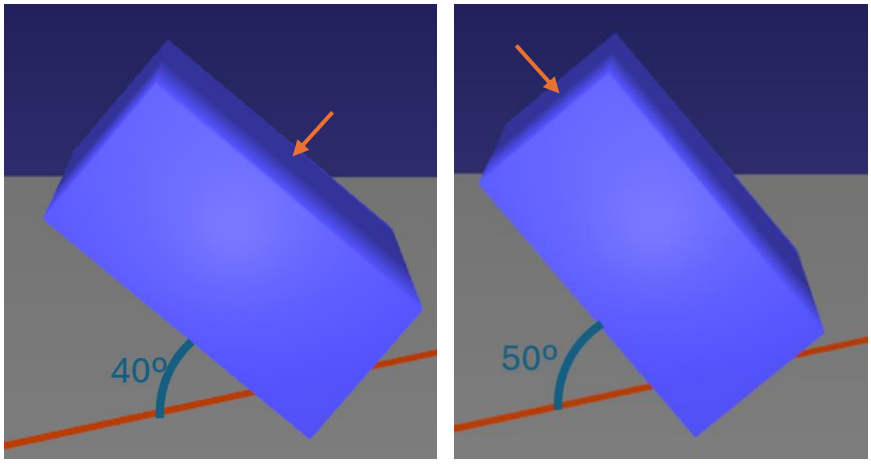


Fig 76: Ejemplos de inclinación de la caja (exterior)

Como se puede apreciar en la figura anterior, la cara tomada como superior en la captura de la izquierda ha sido la de mayor tamaño, ya que esta formaba un ángulo de 40º respecto a la horizontal y por lo tanto se han calculado las trayectorias sobre esa cara.

En la segunda captura se puede apreciar que las trayectorias calculadas están en otra cara, esto es debido a que la cara que anteriormente estaba a 40º ahora está a 50º, y la nueva cara que se ha utilizado para la soldadura ahora tiene un ángulo de 40º respecto a la horizontal.

Por lo tanto, se debe conocer esta característica a la hora de colocar la caja para soldar la cara deseada.

La segunda limitación, se debe a que la caja no debe estar perfectamente alineada, es decir con una rotación en Z de 0º o 180º, sucede exactamente lo mismo que al realizar la soldadura del interior de una caja, como se ha comentado previamente

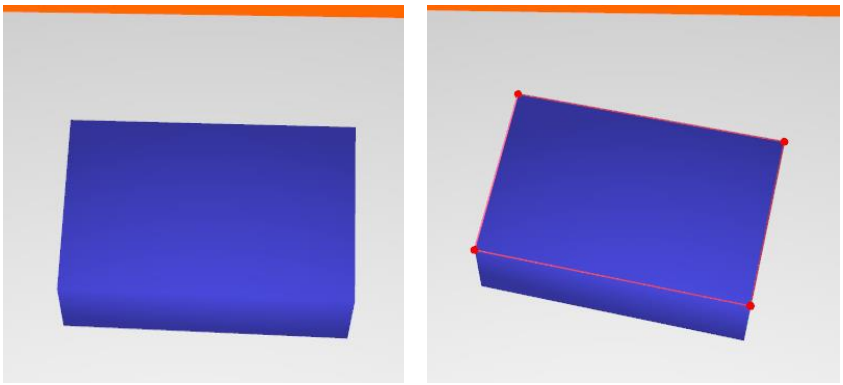


Fig 77: Ejemplos de rotación de la caja (exterior)

En la figura anterior se puede apreciar como la primera caja está totalmente alineada y la segunda no, está rotada 10º. El problema es el mismo que con el interior de la caja, si están alineadas no es posible ordenar los puntos y se produce un error en el cálculo de las trayectorias.

La última limitación es debida principalmente al alcance del robot y al posicionamiento de la caja. En este tipo de aplicación en ciertas ocasiones no es posible obtener información de algunas de las caras.

Esta pérdida de información sucede porque con el alcance de 0,5m del robot y la necesidad de que el objeto debe estar a más de 0,5m de la cámara para ser detectado, resulta imposible para ciertas situaciones recrear el objeto.

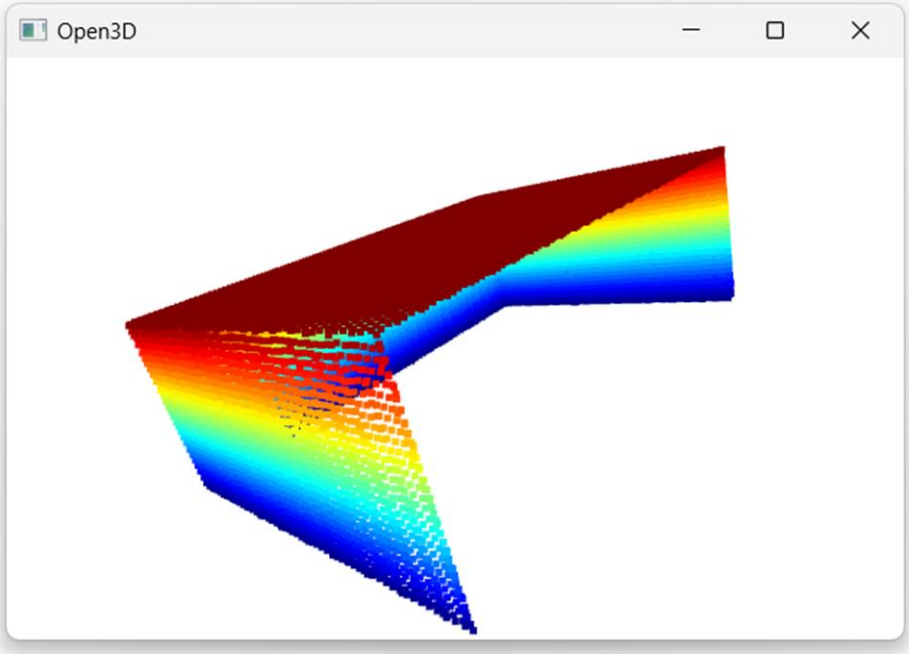


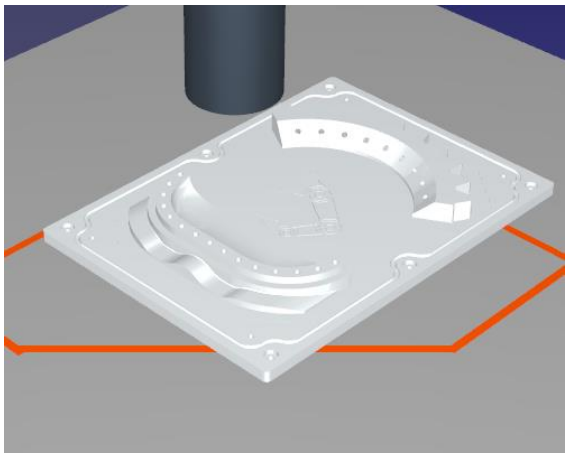
Fig 78: Nube de puntos incompleta de una caja

Como se puede apreciar en la figura anterior, debido a la posición y rotación de la caja, no ha sido posible obtener información de todas las caras. Esto produce un error en el cálculo de las trayectorias al no tener información de los 5 planos necesarios.

7.1.4. Método ICP

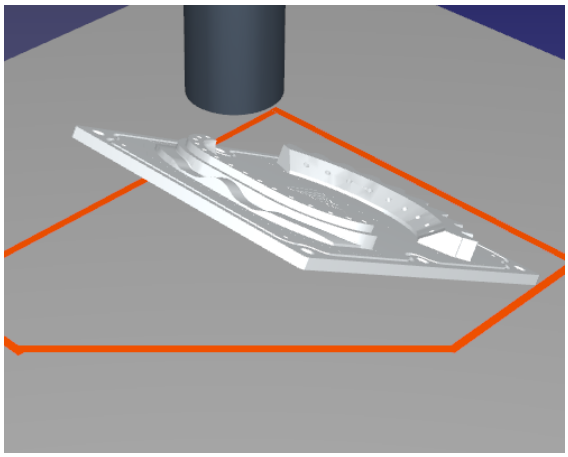
- Pruebas exitosas en la soldadura usando el Método ICP:

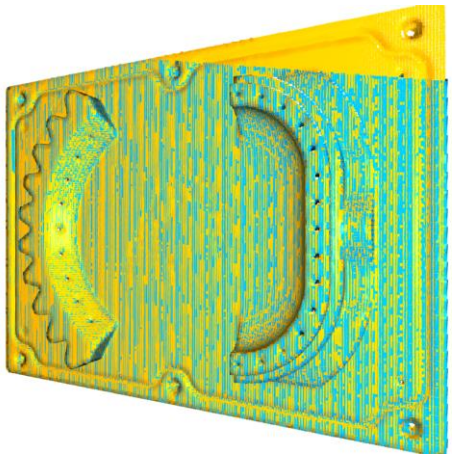
En las siguientes figuras se mostrará la posición de la pieza y las dos nubes solapadas, es decir, la nube de color amarillo es la nube conocida y la nube azul es la obtenida por la cámara. Si están solapadas significa que la transformación de los puntos es correcta, por lo tanto la trayectoria a realizar es correcta también.



Método ICP
- Rot [X,Y',Z''] deg X=0, Y=0, Z=0

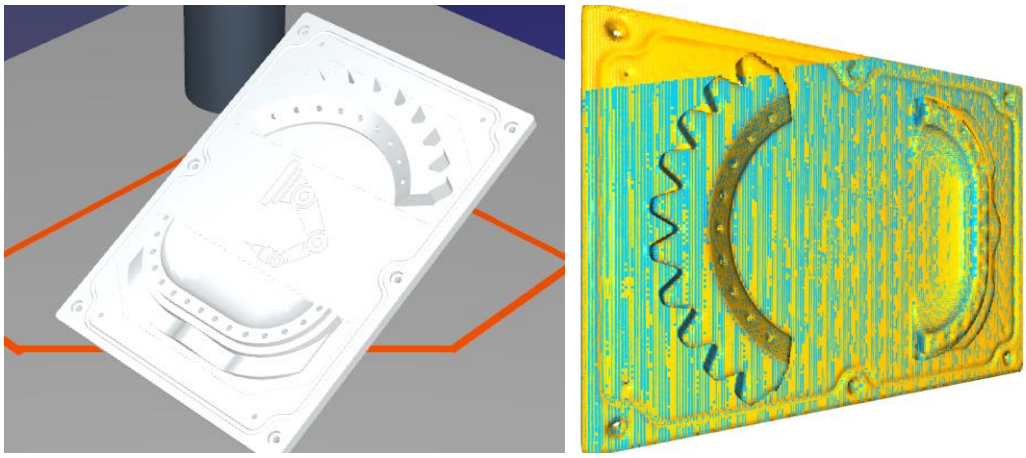
Fig 79: Pruebas exitosas con Método ICP 1





Método ICP
- Rot [X,Y',Z''] deg X=0, Y=-30, Z=0

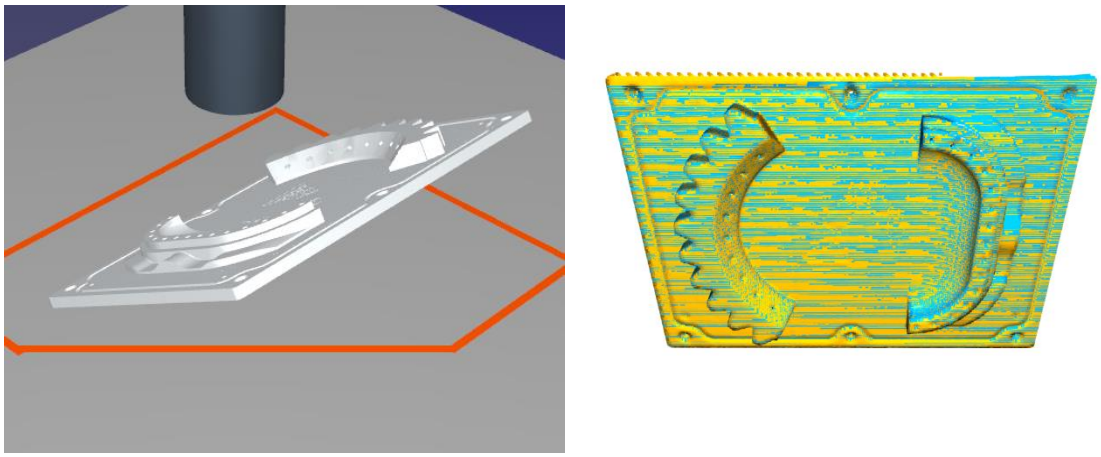
Fig 80: Pruebas exitosas con Método ICP 2



Método ICP

- Rot [X,Y',Z''] deg X=0, Y=30, Z=0

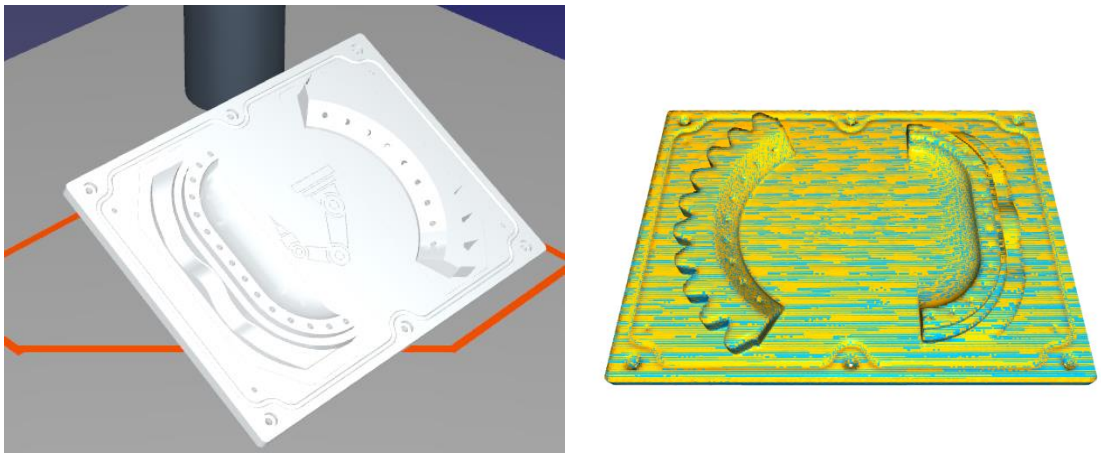
Fig 81: Pruebas exitosas con Método ICP 3



Método ICP

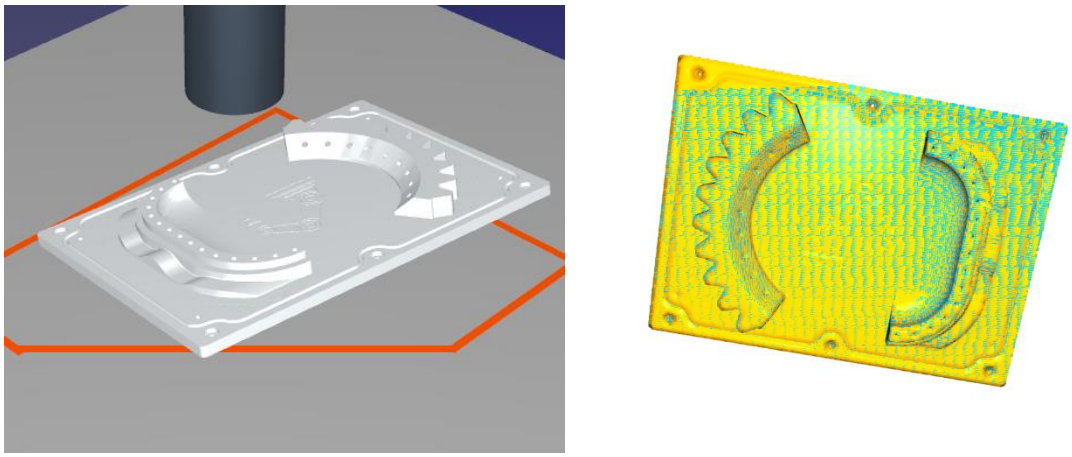
- Rot [X,Y',Z''] deg X=30, Y=0, Z=0

Fig 82: Pruebas exitosas con Método ICP 4



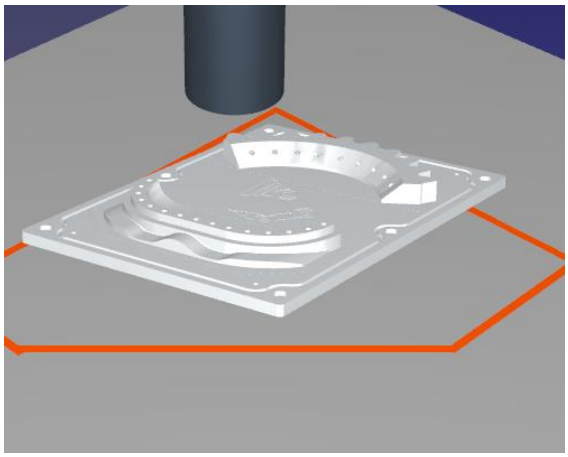
Método ICP
- Rot [X,Y',Z''] deg X=-30, Y=0, Z=0

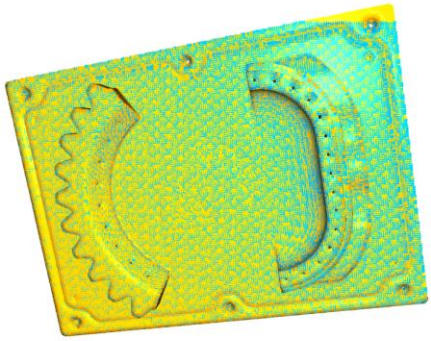
Fig 83: Pruebas exitosas con Método ICP 5



Método ICP
- Rot [X,Y',Z''] deg X=10, Y=0, Z=-10

Fig 84: Pruebas exitosas con Método ICP 6

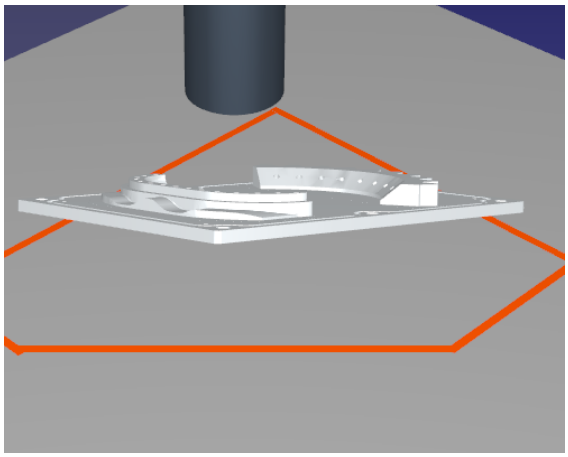


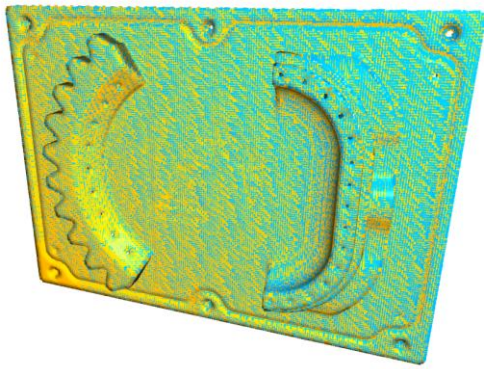


Método ICP

- Rot [X,Y',Z''] deg X=10, Y=-10, Z=10

Fig 85: Pruebas exitosas con Método ICP 7

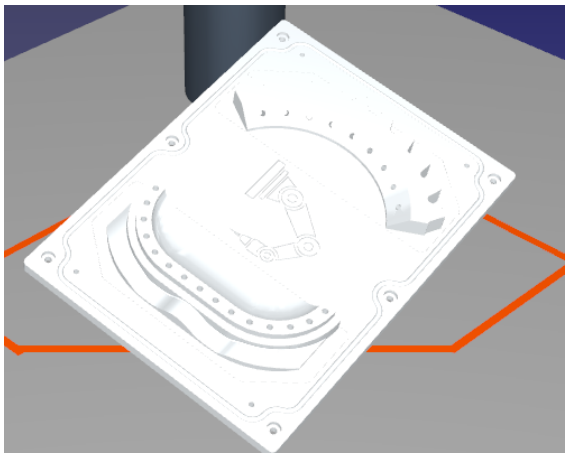


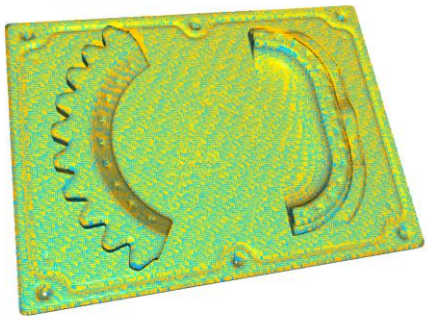


Método ICP

- Rot [X,Y',Z''] deg X=20, Y=-20, Z=0

Fig 86: Pruebas exitosas con Método ICP 8





Método ICP			
- Rot [X,Y',Z''] deg	X=-20,	Y=10,	Z=10

Fig 87: Pruebas exitosas con Método ICP 9

- Limitaciones en la soldadura usando el método ICP

La principal limitación se debe a la capacidad del método ICP para determinar la transformación que debe aplicar a la nube source, la nube inicial y conocida del objeto, para obtener la nueva nube.

Realizando las pruebas se han obtenido ciertos límites de inclinación de la pieza, dichos limites se pueden observar en las figuras, Fig 80, Fig 81, Fig 82 y Fig 83. Con las rotaciones en los ángulos indicados se logra resolver la correspondencia de los puntos.

En las figuras restantes se muestra la capacidad del algoritmo ICP para resolver el problema.

En el caso de la pieza de pruebas si se rota más de 10º o menos de -10º el eje Z, el algoritmo no es capaz de resolver el problema. Dicho error es producido por la forma plana de la pieza a la cual este tipo de rotaciones son más difíciles de resolver la correspondencia de los puntos.

Las pruebas usando el método ICP se han podido llevar a cabo con una gran variedad de posiciones y rotaciones, pero este tipo de aplicación es menos robusta que las 3 anteriores y además para cada pieza habría que realizar un guardado de una nube original y elegir cada punto de forma individual. Las otras 3 aplicaciones anteriores siendo más exitosas, sirven para una amplia gama de tamaños, posiciones y rotaciones, lo cual las dota de una mayor flexibilidad que usando el algoritmo ICP.

7.2. PRUEBAS EN EL LABORATORIO

Una vez conocidas las capacidades y limitaciones del proyecto se realizaron distintas pruebas en el laboratorio con un robot real y usando la cámara elegida.

Primero se realizaron numerosas calibraciones de la cámara para conocer sus parámetros intrínsecos. Los resultados de 4 calibraciones son los siguientes:

$$Calib\_mtx = \begin{pmatrix} 919.814 & 0 & 661.079 \\ 0 & 919.099 & 363.060 \\ 0 & 0 & 0 \end{pmatrix}$$ Overall RMS re-projection error: 0.268 Nº de Fotos: 13
$$Calib\_mtx = \begin{pmatrix} 919.908 & 0 & 662.738 \\ 0 & 919.099 & 363.060 \\ 0 & 0 & 0 \end{pmatrix}$$ Overall RMS re-projection error: 0.226 Nº de Fotos: 13
$$Calib\_mtx = \begin{pmatrix} 924.549 & 0 & 658.52 \\ 0 & 926.632 & 360.349 \\ 0 & 0 & 0 \end{pmatrix}$$ Overall RMS re-projection error: 0.210 Nº de Fotos: 16
$$Calib\_mtx = \begin{pmatrix} 920.410 & 0 & 661.239 \\ 0 & 919.928 & 363.169 \\ 0 & 0 & 0 \end{pmatrix}$$ Overall RMS re-projection error: 0.260 Nº de Fotos: 16

Teniendo en cuenta el error de reproyección, que es la diferencia entre la posición proyectada de un punto en la imagen usando el modelo de cámara, es decir, la matriz de parámetros intrínsecos, y su posición real en la foto. En este caso, se ha elegido como mejor modelo la tercera matriz de parámetros intrínsecos, ya que se ha obtenido el menor error de reproyección, 0,21.

El segundo paso una vez conocidos los parámetros anteriores se realizaron varias camera-hand-calibration para obtener los parámetros extrínsecos de la cámara. Los resultados de 4 camera-hand-calibration son los siguientes:

$$camera\_pose = \begin{pmatrix} 0.9998 & -0.0189 & -0.0027 & -33.1729 \\ 0.0188 & 0.9993 & -0.03228 & -86.4695 \\ 0.0034 & 0.0322 & 0.9995 & 69.7168 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Nº de Fotos: 13
$$camera\_pose = \begin{pmatrix} 0.9998 & -0.0193 & -0.0009 & -34.6051 \\ 0.0193 & 0.9992 & -0.0343 & -87.0364 \\ -0.0003 & 0.0343 & 0.9994 & 56.1689 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Nº de Fotos: 13
$$camera\_pose = \begin{pmatrix} 0.9998 & -0.0172 & 0.0023 & -16.9112 \\ 0.0173 & 0.9992 & -0.0365 & -87.4286 \\ -0.0017 & 0.0366 & 0.9993 & 63.7507 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Nº de Fotos: 16
$$camera\_pose = \begin{pmatrix} 0.9998 & -0.0213 & -0.0008 & -32.9936 \\ 0.0213 & 0.9991 & -0.0359 & -83.6659 \\ 0.0016 & 0.0358 & 0.9994 & 69.0214 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Nº de Fotos: 16

Estas matrices relacionan el efector final del robot con el objetivo de la cámara, de esta forma la posición del objetivo es siempre conocida.

A la vez se midió la posición del objetivo de la cámara mediante una regla y un pie de rey en el mismo laboratorio. También se usaron los planos de la pieza 3D creada para conocer mejor esta posición.

Se cálculo la media de las matrices eliminando la tercera ya que variaba demasiado respecto a las otras, el resultado es este:

$$camera\_pose = \begin{pmatrix} 0.9998 & -0.0198 & -0.0009 & -33.5904 \\ 0.0198 & 0.9992 & -0.0359 & -85.7239 \\ 0.0016 & 0.0341 & 0.9994 & 64.969 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Comparando con las mediciones en el laboratorio se tomó como correcta la matriz de parámetros extrínsecos.

Las primeras pruebas realizadas fueron con el objeto de dos planos. Para ello se usaron 2 cartones de distinto tamaño que podían ser doblados y colocados en distintas orientaciones.

Las pruebas de identificación del objeto fueron exitosas. Para comprobar que la trayectoria se generaba correctamente, se analizaba la nube de puntos creada y también se reconstruía la malla resultante dentro del software.

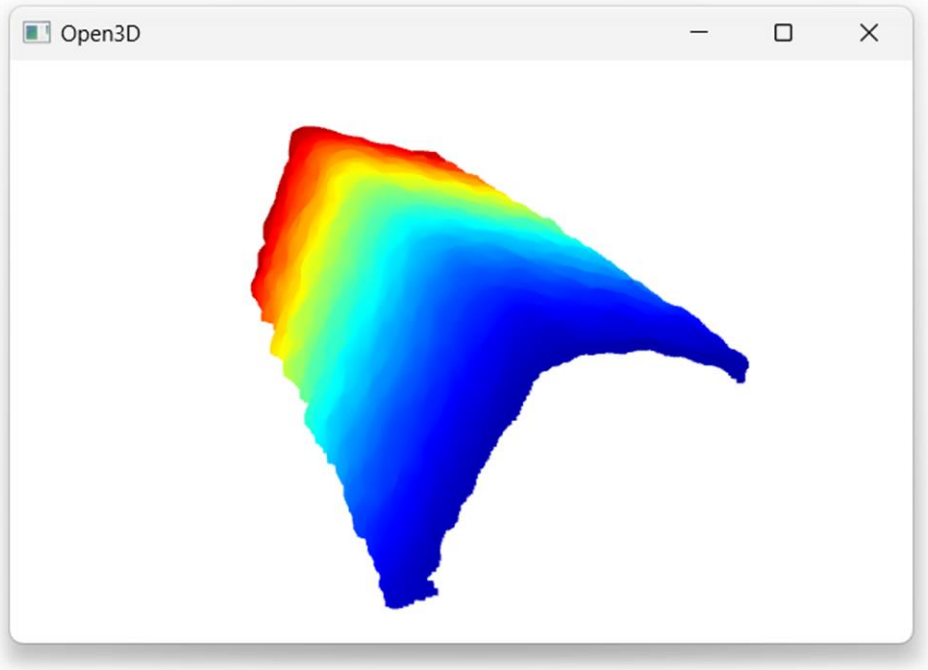


Fig 88: Nube de puntos de objeto real

A pesar de que la identificación no es perfecta como sucedía en el simulador, el cálculo de las trayectorias si se puede realizar sin problemas.

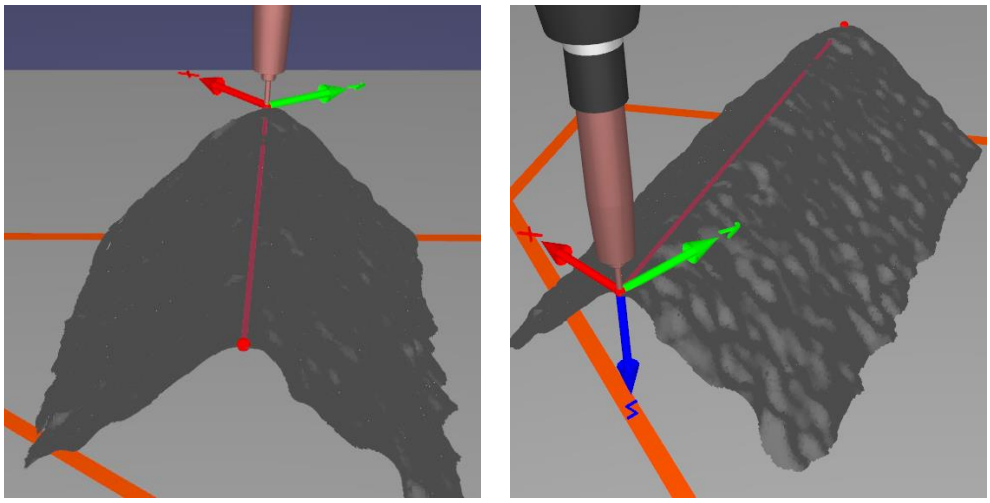


Fig 89: Trayectoria calculada en la malla creada

En la figura anterior se ha creado la malla en la estación de trabajo y se ha calculado la trayectoria para la soldadura. Como se puede apreciar el resultado de las trayectorias es correcto.

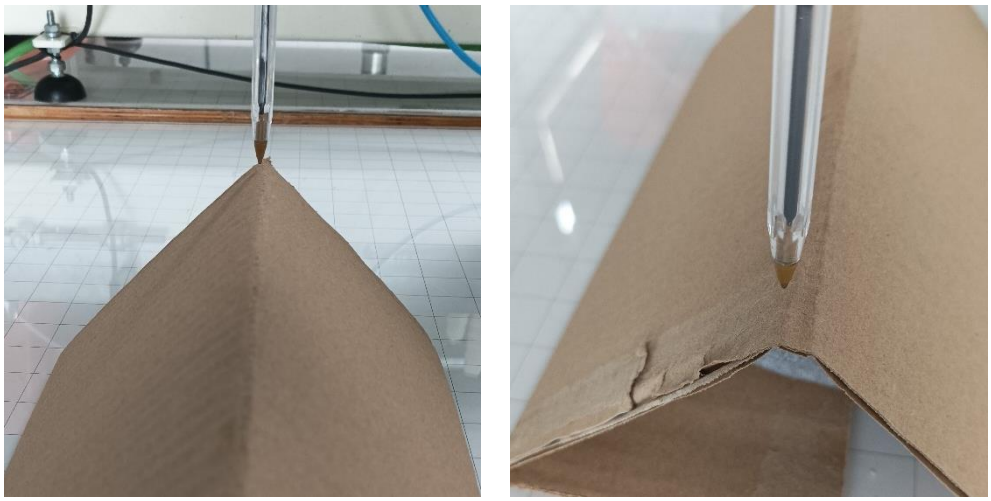


Fig 90: Error en la trayectoria del objeto real

La figura anterior se corresponde con el mismo objeto identificado anteriormente, como se puede apreciar el robot no se posiciona correctaente en el objeto, esta prueba tenía un error entre 5 y 10 mm, sucede igual con las demás pruebas.

Este problema que sucede con el objeto real principalmente es debido al error introducido con los parámetros extrínsecos de la cámara. Cuando se crea la nube de puntos esta se referencia al sistema de coordenadas del robot, si dichos parámetros no están perfectamente obtenidos los puntos referenciados estarán ligeramente desplazados.

Que a la hora de crear la malla en RoboDK y el robot se posicione bien es un indicador más de que no ha fallado el cálculo de la trayectoria sino la obtención de los datos del mundo real.

El segundo tipo de aplicación a probar fue el interior de la caja. Con el objeto anterior, el plano, solo se realizaba una sola captura del objeto para obtener toda su información. En este caso se deben realizar 5 capturas del objeto en distintas posiciones del robot.

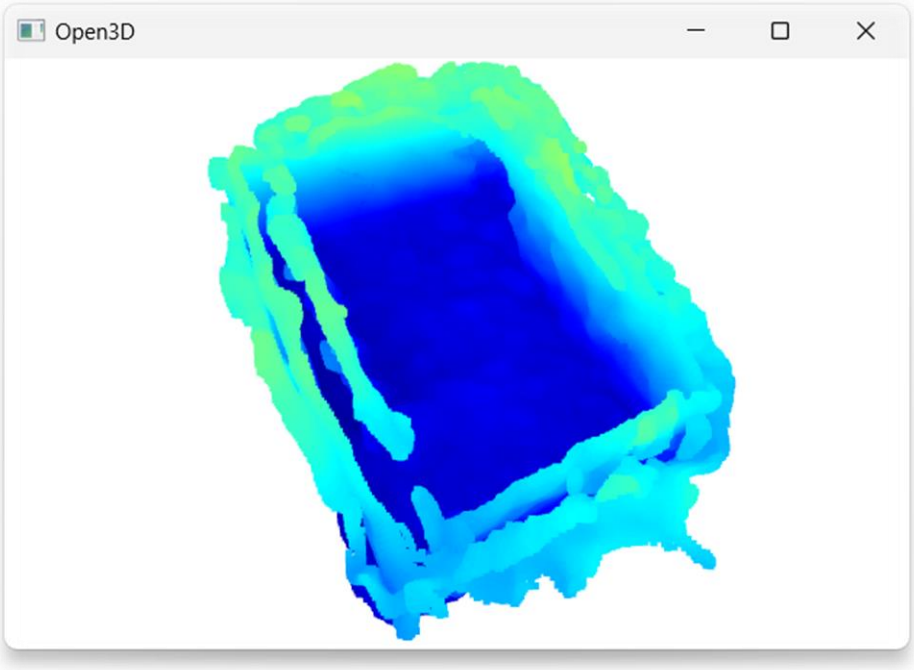


Fig 92: Nube de puntos resultante de una caja real 1

Como se puede apreciar en la figura anterior, la creación de la nube de puntos no es muy buena, por lo que, en algunos casos se puede obtener las trayectorias a realizar, pero como es obvio no van a estar bien.

También se hicieron pruebas con la aplicación de exterior de la caja.

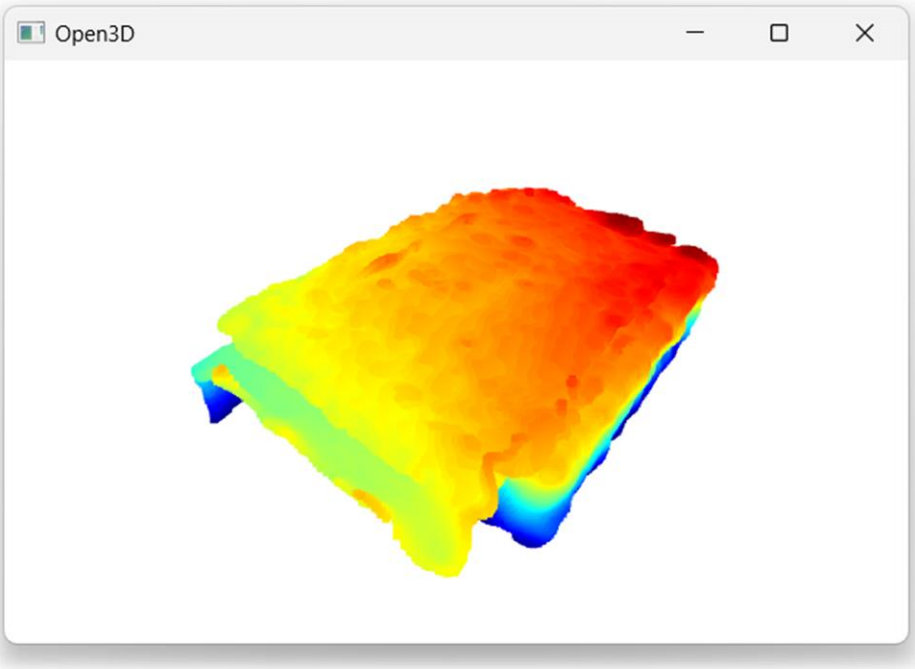


Fig 93: Nube de puntos resultante de una caja real 2

En la figura anterior se puede observar que aparece el mismo problema, el objeto no se reconstruye de una forma correcta. Además, también se aprecia la capacidad de la cámara Intel Realsense 415D, donde debería se una cara plana, aparecen ciertas formas que no deberían estar. Esto último nos muestra la potencia de captar imágenes de profundidad de la cámara.

Tras realizar más calibraciones de cámara y distintas pruebas con más objetos, se finalizaron las pruebas de laboratorio sin haber podido resolver los problemas de calibración con los medios obtenidos.

El problema principal que ha imposibilitado la viabilidad del proyecto ha sido una acumulación de errores a la hora de la captura de imágenes, estos errores son 4 :

- Parámetros extrínsecos:

El cálculo de la posición de la cámara respecto al robot debe ser totalmente precisa a la hora de referenciar el objeto y conocer su posición con exactitud. Dicho resultado de la matriz resultante es el que más error genera en el cómputo global de errores.

- Parámetros intrínsecos:

El error de estos parámetros está directamente relacionado con el procesamiento de los datos obtenidos por la cámara. Dichos parámetros son los que dan a partir de la captura de imagen obtenida la nube de puntos resultante, es decir la forma del objeto en sí. Los parámetros intrínsecos también tienen importancia en el cálculo de los parámetros intrínsecos, el error en el cálculo de estos modifica el resultado final de los parámetros extrínsecos.

- Repetibilidad:

Este tipo de error está relacionado directamente con el robot, se refiere a la capacidad del robot de estar en la posición exacta que se ha seleccionado. En el caso del robot UR3e es de 0,1 mm, es una distancia pequeña, pero puede influir ya que esta distancia es en la muñeca del robot, pero dependiendo como sea este error puede resultar en un error mayor a la hora de referenciar el objeto debido a estar a cierta distancia, este error puede aumentar.

- Soporte de la cámara:

El soporte diseñado y creado con una impresora 3D tiene cierta soltura, y aunque la cámara dentro de él está totalmente fija y no ha posibilidad de que se caiga o sea dañada, se puede mover y modificar su orientación que por poco que sea influye directamente en los parámetros extrínsecos aumentando el error en las capturas de imágenes.

8. CONCLUSIONES

En el proyecto, se ha desarrollado un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo.

El objetivo principal era crear una forma innovadora de integrar una cámara en un robot para realizar una soldadura, con el fin dotar de gran flexibilidad a una célula de trabajo. Esto permitiría que, con la ayuda de un operador, se lograra soldar una amplia variedad de piezas de diferentes tamaños y formas, sin necesidad de modificar o reprogramar el robot.

A pesar de las limitaciones existentes, los resultados obtenidos en la simulación dentro del software RoboDK han sido un éxito. Se ha conseguido que el sistema sea capaz de identificar cuatro tipos de objetos, en una amplia gama de tamaños, siempre que se encuentren dentro del alcance del robot y la cámara.

Sin embargo, al trasladar las pruebas al entorno real, aparecieron problemas principalmente relacionados con la obtención precisa de la información del objeto. Estos errores resultaron irreparables, lo que impidió la continuación de las pruebas.

Si se deseara avanzar con el proyecto se deberían resolver ciertos problemas. Las soluciones a estos podrían ser las siguientes:

- Una cámara de profundidad con mayores prestaciones que la IntelRealsense D415 para obtener mejores calibraciones y representaciones de los objetos mediante nube de puntos.
- Un soporte de mejores características que fije la cámara mejor y genere menos errores.

Debido al tiempo, materiales disponibles y los problemas que aparecieron, no se ha implementado ningún tipo de soldador al robot real en el laboratorio. Sin embargo, se diseñó el soporte correspondiente que permite la instalación de un soldador al robot y la cámara.

9. BIBLIOGRAFÍA

Todo sobre los Robots Colaborativos. Neobotik. Recuperado de https://www.neobotik.com/robots-colaborativos/#:~:text=%C2%BFQu%C3%A9%20son%20los%20robots%20colaborativos,laboral%2C%20de%20ah%C3%AD%20su%20nombre

Ventajas y aplicaciones de visión artificial en los robots colaborativos. Universal Robots. Recuperado de https://www.universal-robots.com/es/blog/vision-artificial-en-robots/

El Robot UR3e. Universal Robots. Recuperado de https://www.universal-robots.com/products/ur3-robot/

Universal Robots e-Series Manual de usuario. (2018). Universal Robots. Recuperado de https://cfzcobots.com/wp-content/uploads/2018/06/UR3e_User_Manual_es_Global.pdf

UR3e Technical Specifications. Universal Robots. Recuperado de https://www.universal-robots.com/media/1807464/ur3e-rgb-fact-sheet-landscape-a4.pdf

ABB IRB140. Eurobots. Recuperado de https://www.eurobots.es/irb-140-es.html

Intel RealSense Depth Camera D415. Intel. Recuperado de https://www.intelrealsense.com/depth-camera-d415/

Open3D Documentation. Open3D. Recuperado de https://www.open3d.org/docs/release/

RoboDK Documentation. RoboDK. Recuperado de https://robodk.com/doc/en/Basic-Guide.html

RoboDK API for Python. RoboDK. Recuperado de https://robodk.com/doc/en/PythonAPI/index.html

DETECTOR DE MANOS EN PYTHON, CON «opencv» Y «mediapipe». Recuperado de https://programacionpython80889555.wordpress.com/2021/06/22/detector-de-manos-en-python-con-opencv-y-mediapipe/

Quick-Changer. OnRobot. Recuperado de https://onrobot.com/es/productos/quick-changer

Configuración del streaming de la cámara Intel Realsense D415. Github. Recuperado de https://github.com/IntelRealSense/librealsense/tree/master/wrappers/python

ANEXO 1. RELACIÓN DEL TRABAJO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE DE LA AGENDA 2030

Anexo al Trabajo de Fin de Máster: Relación del trabajo con los Objetivos de Desarrollo Sostenible de la agenda 2030

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**			X	
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

El sistema de soldadura diseñado tiene gran relación con la industria, la innovación y las infraestructuras.

La implementación de este sistema implica el desarrollo y aplicación de tecnologías avanzadas, como la robótica y la visión artificial. Al aplicar dichas técnicas avanzadas con distintos sistemas para realizar una aplicación concreta, se demuestra la capacidad de la industria para adaptar y utilizar tecnologías innovadoras en diferentes contextos.

Además, este sistema tiene un impacto directo en la innovación, ya que combina diferentes disciplinas y conocimientos para lograr un objetivo específico. La integración de la robótica y visión artificial para la soldadura de una pieza representa una solución innovadora en el campo de la automatización y la robótica.

Finalmente, en cuanto a las infraestructuras, este sistema puede tener implicaciones en la optimización de procesos y la mejora de la productividad en diferentes sectores gracias a la gran capacidad de adaptación a diferentes y nuevas piezas. Dicha flexibilidad podría traducirse en una mayor eficiencia y productividad en la cadena de producción.

ANEXO 2. FICHAS TÉCNICAS DE LOS ELEMENTOS

Robot UR3e:

UNIVERSAL ROBOTS

UR3e
Technical Specifications

With a 3 kg payload capacity and 500 mm reach, the compact form factor of the UR3e makes it a fit for tight workspaces.

Today, more than 50,000 UR collaborative industrial robots have been delivered to customers across industries and around the world. UR3e is one of four e-Series cobots, each with a different payload and reach combination. e-Series brings incredible flexibility and unparalleled ease of use to your application.

Contact

Universal Robots A/S
Energivej 25
5260 Odense
Denmark
+45 89 93 89 89
sales@universal-robots.com
universal-robots.com

UR3e

Specification

Payload	3 kg (6.6 lbs)
Reach	500 mm (19.7 in)
Degrees of freedom	6 rotating joints
Programming	12 inch touchscreen with polyscope graphical user interface

Performance

Power, Consumption, Maximum Average	300 W	
Power, Consumption, Typical with moderate settings (approximate)	100 W	
Safety	17 configurable safety functions	
Certifications	EN ISO 13849-1, PLd Category 3, and EN ISO 10218-1	
Force Sensing, Tool Flange	Force, x-y-z	Torque, x-y-z
Range	30.0 N	10.0 Nm
Precision	2.0 N	0.1 Nm
Accuracy	3.5 N	0.1 Nm

Movement

Pose Repeatability per ISO 9283	± 0.03 mm	
Axis movement	Working range	Maximum speed
Base	± 360°	± 180°/s
Shoulder	± 360°	± 180°/s
Elbow	± 360°	± 180°/s
Wrist 1	± 360°	± 360°/s
Wrist 2	± 360°	± 360°/s
Wrist 3	Infinite	± 360°/s
Typical TCP speed	1 m/s (39.4 in/s)	

Features

IP classification	IP54
ISO 14644-1 Class Cleanroom	5
Noise	Less than 60 dB(A)
Robot mounting	Any orientation
I/O ports	
Digital in	2
Digital out	2
Analog in	2
Tool I/O Power Supply Voltage	12/24 V
Tool I/O Power Supply	600 mA

Physical

Footprint	Ø 128 mm
Materials	Aluminium, Plastic, Steel
Tool (end-effector) connector type	M8 | M8 8-pin
Cable length robot arm	6 m (236 in) cable included. 12 m (472 in) and high-flex options available.
Weight including cable	11.2 kg (17.7 lbs)
Operating temperature range	0-50°C
Humidity	90%RH (non-condensing)

Control Box

Features

IP classification	IP44
ISO 14644-1 Class Cleanroom	6
Operating temperature range	0-50°C
Humidity	90%RH (non-condensing)
I/O ports	
Digital in	16
Digital out	16
Analog in	2
Analog out	2
Quadrature Digital Inputs	4
I/O Power Supply	24V 2A
Communication	500 Hz Control frequency Modbus TCP PROFINET Ethernet/IP USB 2.0, USB 3.0
Power source	100-240VAC, 47-440Hz

Physical

Control box size (W x H x D)	460 mm x 449 mm x 254 mm (18.2 in x 17.6 in x 10 in)
Weight	12 kg (26.5 lbs)
Materials	Powder Coated Steel

The control box is also available in an OEM version.

Teach Pendant

Features

IP classification	IP54
Humidity	90%RH (non-condensing)
Display resolution	1280 x 800 pixels

Physical

Materials	Plastic, PP
Weight	1.6 kg (3.5 lbs) including 1m of TP cable
Cable length	4.5 m (177.17 in)

The teach pendant is also available in a 3PE option.

Cámara Intel Realsense 415D:

Features	**Use environment:** Indoor/Outdoor **Image sensor technology:** Rolling Shutter	**Ideal range:** .5 m to 3 m
Depth	**Depth technology:** Stereoscopic **Minimum depth distance (Min-Z) at max resolution:** ~45 cm **Depth Accuracy:** <2% at 2 m[1]	**Depth Field of View (FOV):** 65° × 40° **Depth output resolution:** Up to 1280 × 720 **Depth frame rate:** Up to 90 fps
RGB	**RGB frame resolution:** 1920 × 1080 **RGB frame rate:** 30 fps **RGB sensor technology:** Rolling Shutter	**RGB sensor FOV (H × V):** 69° × 42° **RGB sensor resolution:** 2 MP
Major Components	**Camera module:** Intel RealSense Module D415	**Vision processor board:** Intel RealSense Vision Processor D4
Physical	**Form factor:** Camera Peripheral **Length × Depth × Height:** 99 mm × 20 mm × 23 mm	**Connectors:** USB-C* 3.1 Gen 1* **Mounting mechanism:** – One 1/4-20 UNC thread mounting point. – Two M3 thread mounting points.

Intercambiador de herramientas:

DATASHEET

QUICK CHANGERS

1. Datasheet

1.1. Quick Changers

Quick Changer

Name	Quick Changer I/O support – Robot Side	Quick Changer - Robot Side	Quick Changer - Robot Side 4.5A	Quick Changer - Robot Side
Item #	102326	102037	104277	109498
Version	QC-R – I/O	QC-R v2	QC-R v2-4.5	QC-R v3
Illustration				

Dual Quick Changer

Name	Dual Quick Changer	Dual Quick Changer 4.5A	Dual Quick Changer
Item #	101788	104293	109878
Version	Dual QC v2	Dual QC v2-4.5	Dual QC v3
Illustration			

If not specified, the data represent the combination of the different Quick Changer types/ sides.

Technical data	Quick Changer	Dual Quick Changer	Units
Rated payload *	25 55.11	30 66.13	[kg] [lbs]
Permissible force *	400	600	[N]
Permissible torque **	40		[Nm]
Repeatability	±0.02		[mm]
IP Classification	67		
Operating life (Tool change)	5.000		[cycles]
Operating temperature	From 5 to 50 From 41 to 122		[°C] [°F]

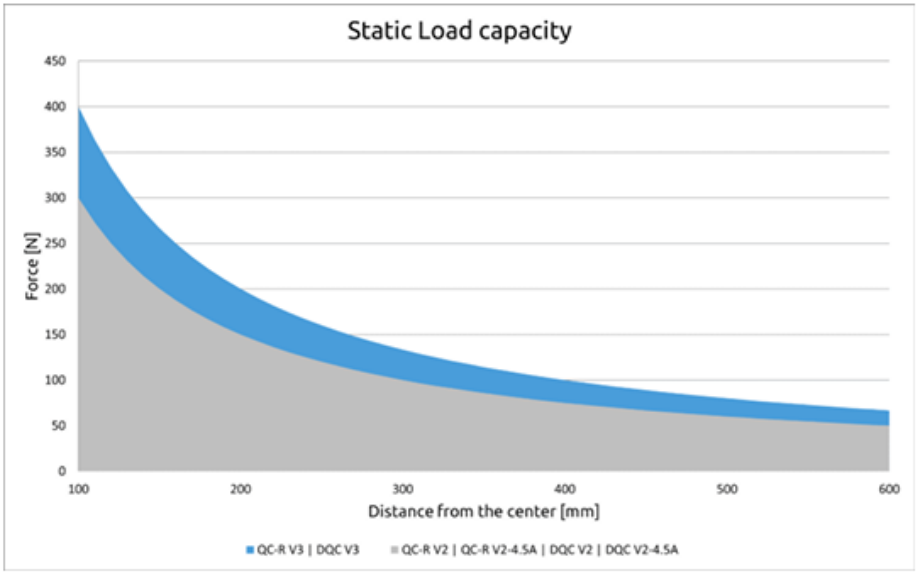
* See static load capacity graph below.

** See QC Maximum Allowed Torque for more details.

	Quick Changer	Quick Changer for I/O	Dual Quick Changer	Quick Changer - Tool Side	Units
Weight	0.06 0.13	0.093 0.21	0.41 0.9	0.14 0.31	[kg] [lb]
Dimensions	See Mechanical dimension section				

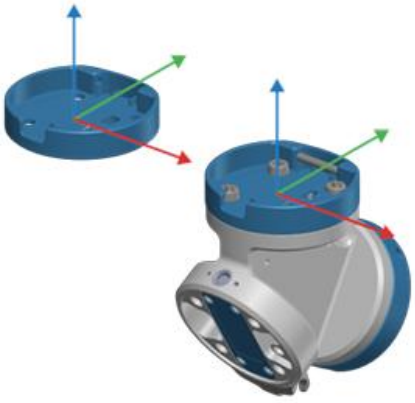
QC-R V3 | DQC V3 and the QC-R V2 | QC-R V2-4.5A | DQC V2 | DQC V2-4.5A

The following graph shows the load capacity that the QC-R V3 | DQC V3 and the QC-R V2 | QC-R V2-4.5A | DQC V2 | DQC V2-4.5A can handle in a static situation. The values for a situation with an acceleration of 2g are half of the static values.



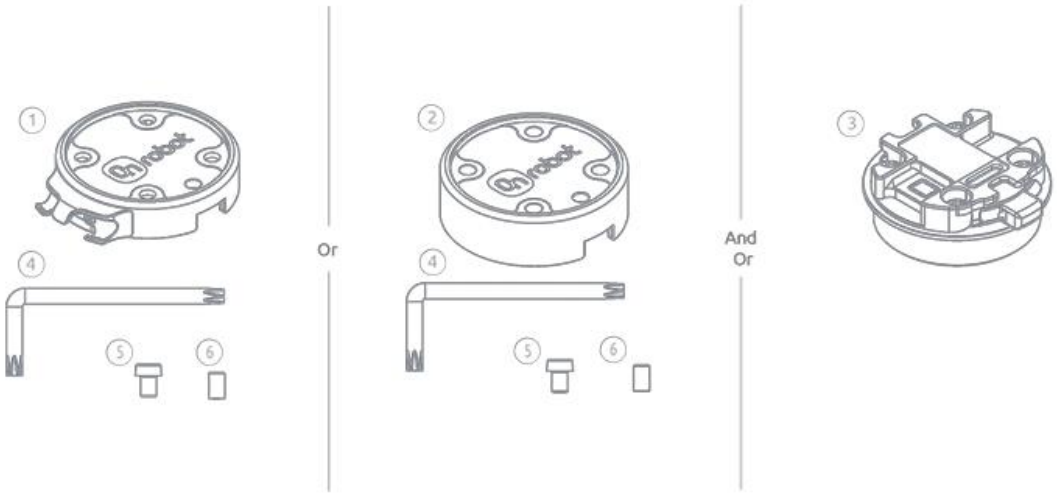
QC Maximum Allowed Torque

The maximum allowed torque applied to the Quick Changer and Dual Quick changer is 40 Nm. The picture below shows the coordinate system from where the maximum allowed torque is calculated.



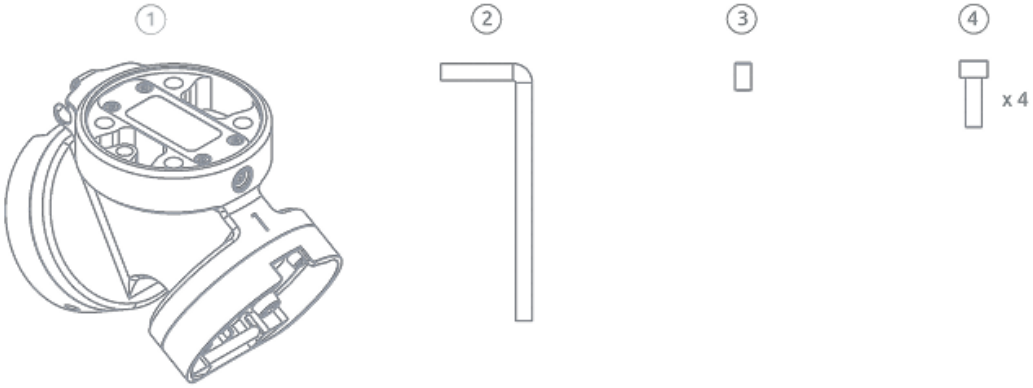
1.2. Quick Changers box content

Quick Changer Robot Side / Quick Changer - I/O Robot Side / Quick Changer Tool Side package content



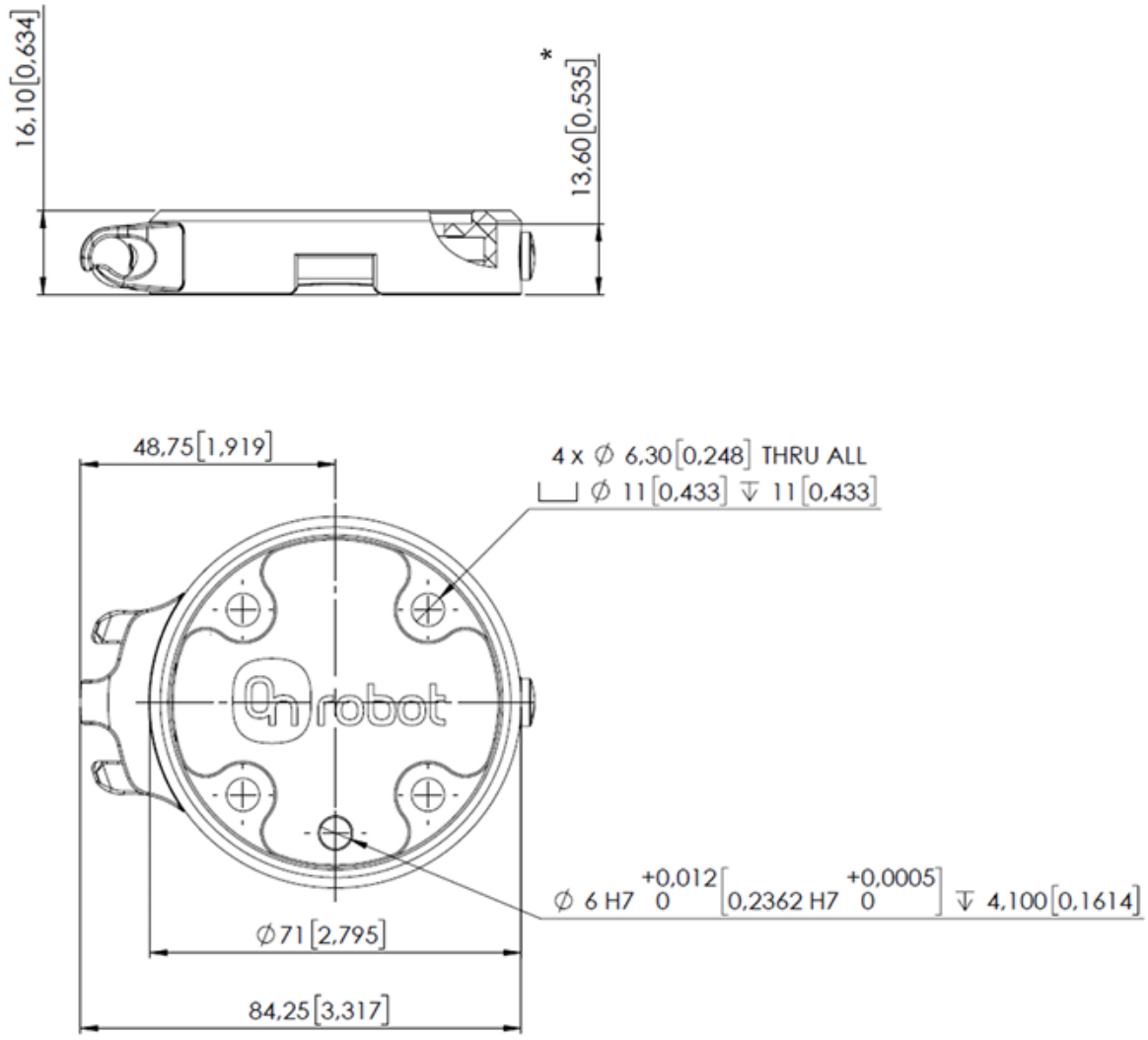
① Quick Changer Robot Side ② Quick Changer - I/O Robot Side ③ Quick Changer Tool Side ④ Torx key T30 ⑤ Screw M6x8mm

⑥ Pin Ø6x10mm



① Dual Quick Changer ② Allen 5 mm key ③ Pin ④ M6x18 mm Screws

1.3. Quick Changer - Robot Side



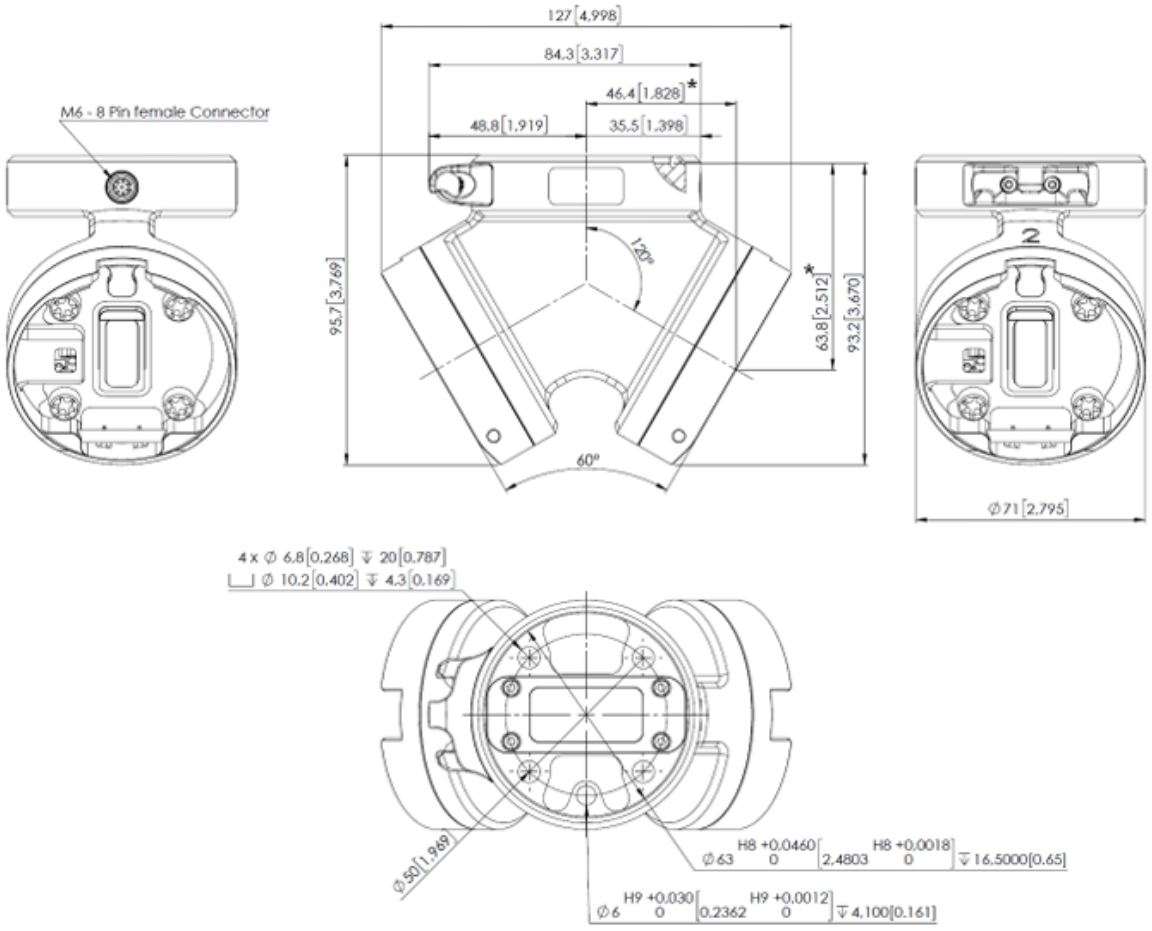
* Distance from Robot flange interface to OnRobot tool.

All dimensions are in mm and [inches].

NOTE:

The cable holder (on the left side) is only required with the long (5 meter) cable.

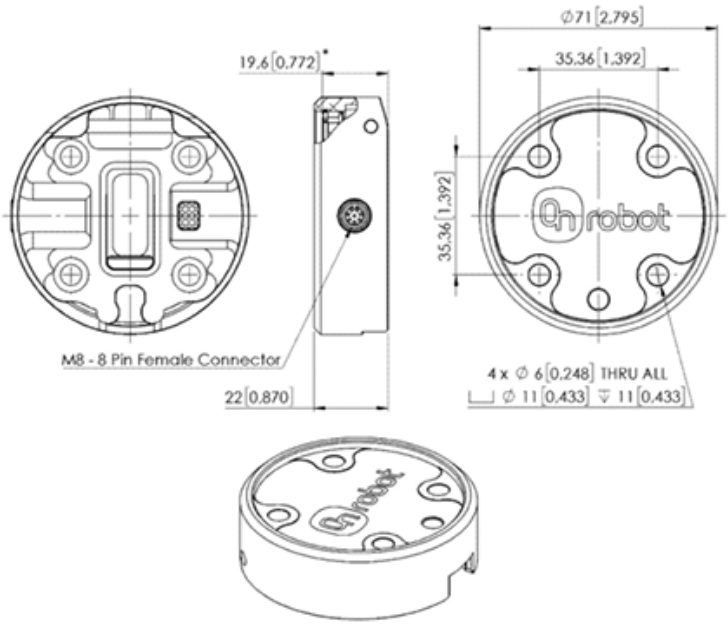
1.4. Dual Quick Changer



* Distance from Robot flange interface to OnRobot tool

All dimensions are in mm and [inches].

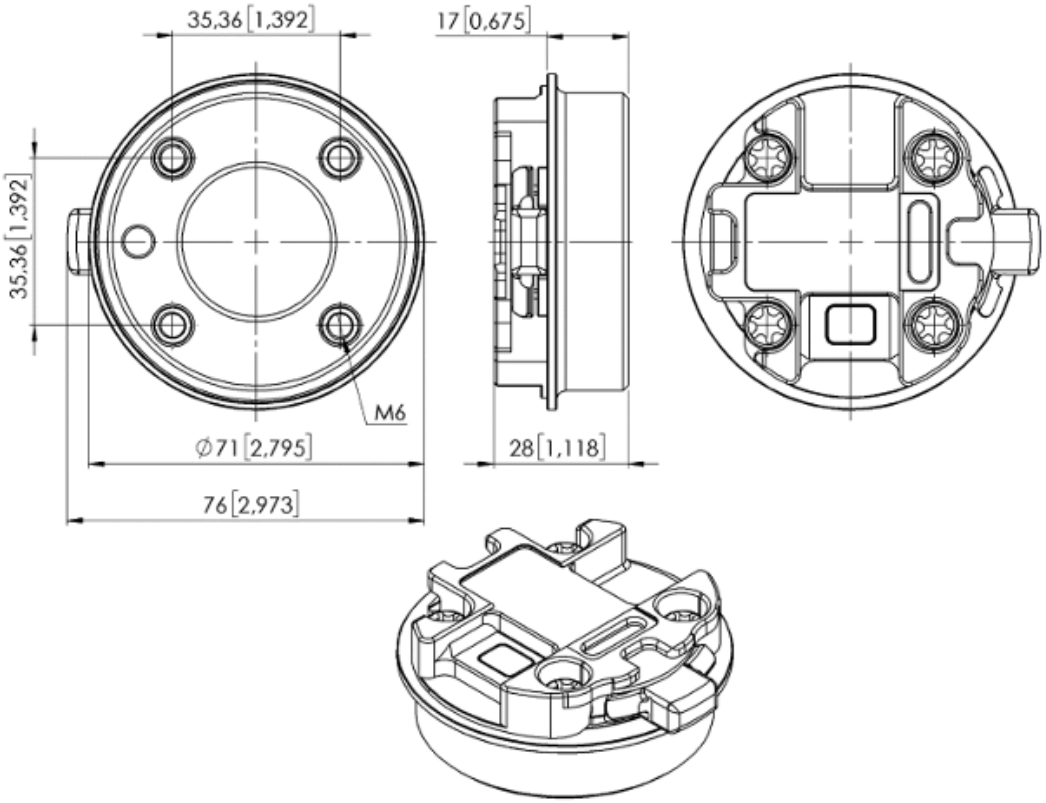
1.5. Quick Changer for I/O - Robot Side



* Distance from Robot flange interface to OnRobot tool

All dimensions are in mm and [inches].

1.6. Quick Changer - Tool Side



ANEXO 3. CÓDIGO IMPLEMENTADO

Aplicación:

```
from robodk.robolink import *
from robodk import *
from tempfile import TemporaryDirectory
import numpy as np
import open3d as o3d
import pyrealsense2 as rs
import cv2
import threading
from queue import Queue
import time
import tkinter as tk
import mediapipe as mp
import copy

#----------------------------------
CAMERA_NAME = 'Intel RealSense D415 Camera'
CAMERA_SIMULATION = True
PLANE_OBJECT = False
INSIDE_BOX_OBJECT = False
OUTSIDE_BOX_OBJECT = False
ICP_OBJECT = False
STEP_BY_STEP = False # Espera antes de cada captura de imagen
VISUALICE_PCD = False # Printea cada pcd despues de la foto
MESH_RECONSTRUCTION = False # Reconstrucción de la malla
SAVE_PCD = False # Guardar la nube de puntos
ANGLE_INSIDE = 65 # Angulo Predeterminado
#---------------- Configuración de la estación ----------------
RDK = Robolink()

# Obtener el robot
robot = RDK.Item('', ITEM_TYPE_ROBOT)
robot_base = RDK.Item('UR3 Base', ITEM_TYPE_FRAME)

# Obtener la cámara simulada de la estación
cam_item = RDK.Item(CAMERA_NAME, ITEM_TYPE_CAMERA)
if not cam_item.Valid():
    cam_item = RDK.Cam2D_Add(RDK.ActiveStation())
    cam_item.setName(CAMERA_NAME)
#if CAMERA_SIMULATION: cam_item.setParam('Open', 1)

# Obtener la herramienta de la cámara, el objetivo de esta
cam_tool = RDK.Item(CAMERA_NAME, ITEM_TYPE_TOOL)

# Obtener la herramienta de soldadura, la punta de esta
welding_tool = RDK.Item('Welding Gun', ITEM_TYPE_TOOL)

# SetSpeed y ZoneData
robot.setSpeed(150, 1000) # Velocidad en mm/s y aceleración en mm/s^2
robot.setZoneData(0)

# Activar la herramienta de la cámara
robot.setPoseTool(cam_tool) # Asi puedo guardar los valores de cam_pose con
robot.Pose()

#-------------------------------------------------------------------------------
```

```
class MainInterface:
    # Interfaz principal

    def __init__(self, master, end):
        self.master = master
        master.title("Soldadura Robotizada")
        master.geometry("440x210")
        master.configure(bg="gray80")
        self.variable_finalizacion = end
        master.resizable(False, False)
        # Variables para almacenar la opción seleccionada
        self.opcion_aplicacion = tk.IntVar()
        self.opcion_simulacion = tk.IntVar()

        # Primer conjunto de radiobuttons
        frame_izquierda = tk.Frame(master, bg="lightblue", padx=10, pady=10)
        frame_izquierda.place(x=10, y=10)
            label1 = tk.Label(frame_izquierda, text="Tipo de aplicación",
bg="lightblue")

        self.radio_button1 = tk.Radiobutton(frame_izquierda, text="Soldadura de
dos planos", variable=self.opcion_aplicacion, value=1, bg="lightblue",
command=self.angulo_caja)
        self.radio_button2 = tk.Radiobutton(frame_izquierda, text="Interior de
caja", variable=self.opcion_aplicacion, value=2, bg="lightblue",
command=self.angulo_caja)
        self.radio_button3 = tk.Radiobutton(frame_izquierda, text="Exterior de
caja", variable=self.opcion_aplicacion, value=3, bg="lightblue",
command=self.angulo_caja)
        self.radio_buttonICP1 = tk.Radiobutton(frame_izquierda, text="Placa con
ICP", variable=self.opcion_aplicacion, value=4, bg="lightblue",
command=self.angulo_caja)

        # Segundo conjunto de Radiobuttons
        frame_derecha = tk.Frame(master, bg="orange2", padx=10, pady=10)
        frame_derecha.place(x=215, y=10)
            label2 = tk.Label(frame_derecha, text="Tipo de simulación",
bg="orange2")

        self.radio_button4 = tk.Radiobutton(frame_derecha, text="Simulación en
RoboDK", variable=self.opcion_simulacion, value=1, bg="orange2")
        self.radio_button5 = tk.Radiobutton(frame_derecha, text="Ejecución real
de la aplicación", variable=self.opcion_simulacion, value=2, bg="orange2")

        # Crear Botón
            self.boton = tk.Button(master, text="Comenzar aplicación",
command=self.eleccion, bg="gray")
            self.botonend = tk.Button(master, text="Finalizar aplicación",
command=self.finalizar, bg="gray40")

        # Colocar elementos en la ventana
        label1.pack(anchor='w')
        label2.pack(anchor='w')
        self.radio_button1.pack(anchor='w')
        self.radio_button2.pack(anchor='w')
        self.radio_button3.pack(anchor='w')
        self.radio_buttonICP1.pack(anchor='w')
        self.radio_button4.pack(anchor='w')
        self.radio_button5.pack(anchor='w')
        self.boton.pack(side=tk.BOTTOM, fill=tk.X)
        self.botonend.place(x=260, y=115)

        # Label y Entry para elegir angulo en "Interior de caja"
          self.label_angulo = tk.Label(master, text="Ángulo de soldadura en
grados:", bg="gray80")
        self.entry_angulo = tk.Entry(master,width=4)

        master.protocol("WM_DELETE_WINDOW", self.finalizar)

    def angulo_caja(self):
        # comprobar si "Interior de caja" esta seleccionado
        if self.opcion_aplicacion.get() == 2:
```

```
            self.label_angulo.place(x=10, y=150)
            self.entry_angulo.place(x=181, y=152)
        else:
            self.label_angulo.place_forget()
            self.entry_angulo.place_forget()

    def eleccion(self):
        # Obtener la opción seleccionada
        opcion_aplicacion = self.opcion_aplicacion.get()
        opcion_simulacion = self.opcion_simulacion.get()

            global PLANE_OBJECT, INSIDE_BOX_OBJECT, OUTSIDE_BOX_OBJECT,
CAMERA_SIMULATION, ICP_OBJECT, ANGLE_INSIDE

        # Realizar acciones dependiendo del valor
        if opcion_aplicacion == 1:
            PLANE_OBJECT = True
        elif opcion_aplicacion == 2:
            INSIDE_BOX_OBJECT = True
            ANGLE_INSIDE = float(self.entry_angulo.get())
        elif opcion_aplicacion == 3:
            OUTSIDE_BOX_OBJECT = True
        elif opcion_aplicacion == 4:
            ICP_OBJECT = True

        if opcion_simulacion == 1:
            CAMERA_SIMULATION = True
        elif opcion_simulacion == 2:
            CAMERA_SIMULATION = False

        # Cerrar la ventana después de mostrar el mensaje
        self.master.destroy()

    def finalizar(self):
        # Cerrar la ventana y poner a true la variable de cierre
        self.variable_finalizacion.set(True)
        self.master.destroy()
```

```
class SimulationInterface:
    # Interfaz antes de ejecutar la soldadura

    def __init__(self, master, option):
        self.master = master
        master.title("Visualización movimiento final")
        master.geometry("350x125")
        master.configure(bg="gray80")
        self.variable_option = option
        master.resizable(False, False)

        # Crear botones
           self.botonsim = tk.Button(master, text="Visualizar Movimiento en
RoboDK", command=self.simulacion, bg = "gray")
          self.botonreal = tk.Button(master, text="Ejecutar Movimiento en el
robot", command=self.ejecucion, bg = "gray")
              self.botonend = tk.Button(master, text="Forzar finalización",
command=self.finalizar, bg = "gray")

        self.botonsim.pack(side=tk.TOP, fill=tk.X, padx=20, pady=5)
        self.botonreal.pack(side=tk.TOP, fill=tk.X, padx=20, pady=5)
        self.botonend.pack(side=tk.TOP, fill=tk.X, padx=20, pady=5)

        master.protocol("WM_DELETE_WINDOW", self.finalizar)

    def simulacion(self):
        self.variable_option.set(1)
        self.master.destroy()

    def ejecucion(self):
        self.variable_option.set(2)
        self.master.destroy()

    def finalizar(self):
        # Cerrar la ventana y poner a true la variable de cierre
        self.variable_option.set(3)
        self.master.destroy()
```

```
class SbSInterface:
    # Interfaz para realizar step by step

    def __init__(self, master, option):
        self.master = master
        master.title("Visualización paso a paso")
        master.geometry("300x50")
        master.configure(bg="gray80")
        self.variable_option = option
        master.resizable(False, False)

        # Crear botones
         self.botonsim = tk.Button(master, text="Tomar captura de la imagen",
command=self.continuar, bg = "gray")

        self.botonsim.pack(side=tk.TOP, fill=tk.X, padx=20, pady=5)

        master.protocol("WM_DELETE_WINDOW", self.continuar)

    def continuar(self):
        self.variable_option.set(1)
        self.master.destroy()
```

```
class ConfigurationInterface:
    # Interfaz para la configuración inicial de la estación

    def __init__(self, master):
        self.master = master

        master.title("Configuración Inicial")
        master.geometry("320x110")
        master.resizable(False, False)
        master.protocol("WM_DELETE_WINDOW", self.save_config)
        master.configure(bg="gray80")

        self.paso_a_paso_var = tk.BooleanVar(value=STEP_BY_STEP)
        self.visualice_pcd_var = tk.BooleanVar(value=VISUALICE_PCD)
        self.mesh_reconstruction_var = tk.BooleanVar(value=MESH_RECONSTRUCTION)
        self.save_pcd_var = tk.BooleanVar(value=SAVE_PCD)

             self.checkbox1 = tk.Checkbutton(master, text="STEP_BY_STEP",
variable=self.paso_a_paso_var, anchor="w", justify="left", bg="gray80")
        self.checkbox1.grid(row=1, column=0, pady=5, padx=10, sticky="w")
             self.checkbox2 = tk.Checkbutton(master, text="VISUALICE_PCD",
variable=self.visualice_pcd_var, anchor="w", justify="left", bg="gray80")
        self.checkbox2.grid(row=2, column=0, pady=5, padx=10, sticky="w")
          self.checkbox3 = tk.Checkbutton(master, text="MESH_RECONSTRUCTION",
variable=self.mesh_reconstruction_var, anchor="w", justify="left", bg="gray80")
        self.checkbox3.grid(row=1, column=1, pady=5, padx=10, sticky="w")
                self.checkbox4 = tk.Checkbutton(master, text="SAVE_PCD",
variable=self.save_pcd_var, anchor="w", justify="left", bg="gray80")
        self.checkbox4.grid(row=2, column=1, pady=5, padx=10, sticky="w")

            self.botonsim = tk.Button(master, text="Guardar Configuración",
command=self.save_config, bg="gray")
        self.botonsim.grid(row=3, column=0, columnspan=2, pady=10)

    def save_config(self):
        global STEP_BY_STEP, VISUALICE_PCD, MESH_RECONSTRUCTION, SAVE_PCD
        STEP_BY_STEP = self.paso_a_paso_var.get()
        VISUALICE_PCD = self.visualice_pcd_var.get()
        MESH_RECONSTRUCTION = self.mesh_reconstruction_var.get()
        SAVE_PCD = self.save_pcd_var.get()
        print("Configuración guardada:")
        print("STEP_BY_STEP:", STEP_BY_STEP)
        print("VISUALICE_PCD:", VISUALICE_PCD)
        print("MESH_RECONSTRUCTION:", MESH_RECONSTRUCTION)
        print("SAVE_PCD:", SAVE_PCD)
        self.master.destroy()


def realsense_streaming(output_queue, depth_request_event):
    # Función para utilizar la Intel Realsense, se crea un hilo y cuando llega
el evento
    #  depth_request_event se realiza una captura de la imagen de profundidad
     # también se esta realizando la busqueda de manos, si aparece una, el
programa se detendrá

    # Configure depth and color streams
    pipeline = rs.pipeline()
    config = rs.config()

    # Get device product line for setting a supporting resolution
    pipeline_wrapper = rs.pipeline_wrapper(pipeline)
    pipeline_profile = config.resolve(pipeline_wrapper)
    device = pipeline_profile.get_device()
    device_product_line = str(device.get_info(rs.camera_info.product_line))

    found_rgb = False
    for s in device.sensors:
        if s.get_info(rs.camera_info.name) == 'RGB Camera':
            found_rgb = True
            break
    if not found_rgb:
        print("The demo requires Depth camera with Color sensor")
```

```
        exit(0)

    config.enable_stream(rs.stream.depth, 1280, 720, rs.format.z16, 30)
    config.enable_stream(rs.stream.color, 1280, 720, rs.format.bgr8, 30)

    # Inicializar MediaPipe Hands
    mp_hands = mp.solutions.hands
    hands = mp_hands.Hands()
    mp_drawing = mp.solutions.drawing_utils

    # Start streaming
    pipeline.start(config)

    # Create an align object
    # rs.align allows us to perform alignment of depth frames to others frames
    # The "align_to" is the stream type to which we plan to align depth frames.
    align_to = rs.stream.color
    align = rs.align(align_to)

    try:
        while True:

            # Wait for a coherent pair of frames: depth and color
            frames = pipeline.wait_for_frames()

           # Align the depth frame to color frame Si quiero que esten alineados
deja esta linea
            frames = align.process(frames)

            depth_frame = frames.get_depth_frame()
            color_frame = frames.get_color_frame()
            if not depth_frame or not color_frame:
                continue

            # Convert images to numpy arrays
            depth_image = np.asanyarray(depth_frame.get_data())
            color_image = np.asanyarray(color_frame.get_data())
             # Apply colormap on depth image (image must be converted to 8-bit
per pixel first)
           depth_colormap = cv2.applyColorMap(cv2.convertScaleAbs(depth_image,
alpha=0.03), cv2.COLORMAP_JET)

            # Hand tracking
            mediapipe_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB)
            results = hands.process(mediapipe_image)

            # Draw anotations
            img = cv2.cvtColor(mediapipe_image, cv2.COLOR_RGB2BGR)
            if results.multi_hand_landmarks:
                for hand_landmarks in results.multi_hand_landmarks:
                                mp_drawing.draw_landmarks(img,  hand_landmarks,
connections=mp_hands.HAND_CONNECTIONS )
                #print("Mano detectada")
                 RDK.ShowMessage("Mano detectada, alejese del robot y pulse el
boton para continuar")
                # sys.exit()

            #images = np.hstack((color_image, depth_colormap))
            images = np.hstack((img, depth_colormap))
            small_images = cv2.resize(images, (1280, 360))
            # Show images
            cv2.namedWindow('RealSense', cv2.WINDOW_AUTOSIZE)
            #cv2.imshow('RealSense', images)
            cv2.imshow('RealSense', small_images)

            # Evento para proporcionar la imagen de profundidad
            if depth_request_event.is_set():
                # Capturar la imagen de profundidad y ponerla en la cola
                depth_image_to_send = np.copy(depth_image)
                output_queue.put(depth_image_to_send)
                # Restablecer el evento después de enviar la imagen
                depth_request_event.clear()
```

```
            key = cv2.waitKey(1)
            # Press esc or 'q' to close the image window
            if key & 0xFF == ord('q') or key == 27 or not hilofuncionamiento:
                cv2.destroyAllWindows()
                break

    finally:
        # Stop streaming
        pipeline.stop()
```

```
def webcam_hand_tracking():
    # Función para realizar la busqueda de manos desde la cámara del ordenador
    # sirve para realizar pruebas, similar a la función para la intel pero en
este caso solo
    # realiza el hand tracking

    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        print("Error al abrir la cámara")
        return

    # Inicializar MediaPipe Hands
    mp_hands = mp.solutions.hands
    hands = mp_hands.Hands()
    mp_drawing = mp.solutions.drawing_utils

    try:
        while True:
            # Capturar frame-by-frame
            ret, color_image = cap.read()
            if not ret:
                break

            # Convertir la imagen a RGB para MediaPipe
            mediapipe_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB)
            results = hands.process(mediapipe_image)

            # Dibujar anotaciones
            img = cv2.cvtColor(mediapipe_image, cv2.COLOR_RGB2BGR)
            if results.multi_hand_landmarks:
                for hand_landmarks in results.multi_hand_landmarks:
                                mp_drawing.draw_landmarks(img,  hand_landmarks,
mp_hands.HAND_CONNECTIONS)
                 RDK.ShowMessage("Mano detectada, alejese del robot y pulse el
boton para continuar")
            # Mostrar imagen
            cv2.imshow('Webcam Hand Tracking', img)

            key = cv2.waitKey(1)
            # Presionar 'q' para cerrar la ventana de imagen
            if key & 0xFF == ord('q') or key == 27 or not hilofuncionamiento:
                break

    finally:
        # Liberar la captura de video y cerrar ventanas
        cap.release()
        cv2.destroyAllWindows()
```

```
def settings_to_dict(settings):
    # Función para obtener los parámetros de la cámara simulada en la estación

    if not settings:
        return {}

    settings_dict = {}
    settings_list = [setting.split('=') for setting in settings.strip().split('
')]
    for setting in settings_list:
        key = setting[0].upper()
        val = setting[-1]

        if key in ['FOV', 'PIXELSIZE', 'FOCAL_LENGTH', 'FAR_LENGTH']:
            val = float(val)
        elif key in ['SIZE', 'ACTUALSIZE', 'SNAPSHOT']:
            w, h = val.split('x')
            val = (int(w), int(h))
        elif key == val.upper():
            val = True  # Flag

        settings_dict[key] = val

    return settings_dict


def get_depth_from_camera():
    # Función para obtener la imagen de profundidad, sirve tanto para el entorno
simulado como real
    # output: -depth_image: Imagen de profundidad

     # Si STEP_BY_STEP es True antes de realizar la captura se espera a una
confirmación con la interfaz
    while STEP_BY_STEP:
        root = tk.Tk()
        option = tk.IntVar(value=0)
        interfaz = SbSInterface(root,option)
        root.mainloop()

        if option.get() == 1:
            break

    # Dependiendo de si es la cámara simulada o la real ,obtiene su imagen de
profundidad
    if CAMERA_SIMULATION:
        # Obtener la imagen de profundidad mediante socket
        depth32_socket = None
        bytes_img = RDK.Cam2D_Snapshot("", cam_item, 'DEPTH')
        if isinstance(bytes_img, bytes) and bytes_img != b'':
            # By socket
            depth32_socket = np.frombuffer(bytes_img, dtype='>u4')
            w, h = depth32_socket[:2]
                depth32_socket = np.flipud(np.reshape(depth32_socket[2:], (h,
w))).astype(np.uint32)

        # Scale it
             depth_image = (depth32_socket / np.iinfo(np.uint32).max) *
cam_settings['FAR_LENGTH']
        depth_image = depth_image.astype(np.float32)

    else:
        # Solicitar la imagen de profundidad
        depth_event.set()
        # Esperar a que la cola tenga la imagen de profundidad
        depth_image = queue.get()

    return depth_image
```

```
def get_image_and_camera_pose(targets):
    # Función para mover el robot a los targets proporcionados y tomar las fotos
    # del objeto
    # input: - targets: Posiciones donde se ha de posicionar el robot
    # output: - depth_image: Lista con las matrices que contienen la información
de las imagenes de profundidad
    #         - camera_pose: Lista con las matrices de transformación de la
posición de la cámara en cada foto

    depth_image = []
    camera_pose = []
    i = 0 # Indice para saber el número de target en el que esto

    while i < len(targets):
        target = targets[i]

        try:
            robot.MoveJ(target)
            # Depth Image
            depth_image.append(get_depth_from_camera())

            # Camera pose
            camera_pose.append(robot.Pose())
            i+=1
        except:
            print("No funciona la captura de imagen "+str(i))
            i +=1

    return depth_image, camera_pose
```

```
def get_point_cloud(depth, cam_pose):
    # Función que a partir de nubes de puntos y posiciones de cámara, crea el
    # objeto referenciado a la base del robot y filtra el entorno
    # input: - depth_image: Lista con las matrices que contienen la información
de las imagenes de profundidad
     #         - camera_pose: Lista con las matrices de transformación de la
posición de la cámara en cada foto
    # output: - pcd_combined: Nube de puntos completa del objeto

    pcd_combined = o3d.geometry.PointCloud() # Creación de nube de puntos general
    pcd_combined.transform(robot_base.Pose()) # La coloco en la base del robot

    for id in range(len(depth)):

        # Creación de nube de puntos a partir de la cámara
                                                              pcd               =
o3d.geometry.PointCloud.create_from_depth_image(o3d.geometry.Image(depth[id]),
cam_mtx, cam_pose[id]) # Creación de nube de puntos a partir de la cámara

        if CAMERA_SIMULATION:
              # Extracción de la posición de la cámara y su correspondiente
traslación de la nube de puntos al lugar
            tras_vector = cam_pose[id].Pos()
            trans_matrix = np.eye(4)
            trans_matrix[:3, 3] = tras_vector
            pcd.transform(trans_matrix)

            # Point filter
            points = np.asarray(pcd.points)
             filtered_points = points[(points[:, 0] >= 10) & (points[:, 0] <=
650) &(points[:, 1] >= 70) & (points[:, 1] <= 650) &(points[:, 2] > -195)]
            pcd.points = o3d.utility.Vector3dVector(filtered_points)

        else:
              # Extracción de la posición de la cámara y su correspondiente
traslación de la nube de puntos al lugar
            tras_vector = cam_pose[id].Pos()
            tras_vector = np.array(tras_vector)/1000
            tras_vector[abs(tras_vector) < 0.00001] = 0

            points = np.asarray(pcd.points)
            points = (points + tras_vector)*1000
             filtered_points = points[(points[:, 0] >= 10) & (points[:, 0] <=
450) &(points[:, 1] >= 90) & (points[:, 1] <= 450) &(points[:, 2] > -180)]

            pcd.points = o3d.utility.Vector3dVector(filtered_points)

        # Combinación de nubes de puntos en la nube de puntos general
        #if VISUALICE_PCD: o3d.visualization.draw_geometries([pcd])
        pcd_combined += pcd

    return pcd_combined
```

```
def get_planes(pcd, num_planes):
    # Función que a partir de la nube de puntos y el numero de planos esperados
obtiene
    # los planos en el objeto y sus ecuaciones
    # input: - pcd_combined: Nube de puntos completa del objeto
    #        - num_planes: Numero de planos esperados en el objeto
    # output: - planes: Lista con la informacion de las nubes de puntos de los
planos obtenidos
    #         - coefficients: Lista con los coeficientes correspondientes a cada
plano

    # Inicializar listas para almacenar los planos encontrados
    planes = []
    coefficients = []
    i = 0

    while i < num_planes:
        # Encontrar un plano
           plane_model,  inliers  =  pcd.segment_plane(distance_threshold=0.01,
ransac_n=3, num_iterations=1000) # 0.01

        # Obtener los puntos del plano
        current_plane = pcd.select_by_index(inliers)

         # Almacenar los coeficientes del plano solo si no coincide con los
existentes
        current_coefficients = np.asarray(plane_model)
        duplicate = False

        # Se comprueba que el plano encontrado es distinto a los anteriores
        for coef in coefficients:
            if np.allclose(current_coefficients, coef,atol=1):#=0.1): Si no se
identifican bien los planos, se debe subir la tolerancia
                duplicate = True
                break

         # Si no esta duplicado se añade a la lista y se cuenta un plano mas
encontrado
        if not duplicate:
            planes.append(current_plane)
            coefficients.append(np.array(current_coefficients))
            i+=1 # Conteo de planos

        # Eliminar los puntos del plano encontrado para buscar el siguiente
        pcd = pcd.select_by_index(inliers, invert=True)
        #o3d.visualization.draw_geometries([pcd])
        #o3d.visualization.draw_geometries([current_plane])

    return planes, coefficients
```

```
def mesh_creation(pcd_mesh):
     # Fuención que sire para la creación de una malla para insertarla en la
estación de roboDK
    # input: - pcd_mesh: Nube de puntos del objeto que se desea reconstruir en
la estación
    # outpur: - mesh_item: Objeto creado en la estación

    # Para modificar los parametros de la malla es con las siguientes variables
    # Con los valores ya puestos funciona tanto para nubes de objetos simulados
como objetos reales
    O3D_NORMALS_K_SIZE = 100 #100
    O3D_MESH_POISSON_DEPTH = 9 #9
    O3D_MESH_DENSITIES_QUANTILE = 0.07 #0.05
    O3D_DISPLAY_POINTS = True
    O3D_DISPLAY_WIREFRAME = True

    pcd_mesh.estimate_normals()
    pcd_mesh.orient_normals_consistent_tangent_plane(O3D_NORMALS_K_SIZE)
                              mesh_poisson,              densities              =
o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(pcd_mesh,
depth=O3D_MESH_POISSON_DEPTH)
          vertices_to_remove    =    densities    <    np.quantile(densities,
O3D_MESH_DENSITIES_QUANTILE)
    mesh_poisson.remove_vertices_by_mask(vertices_to_remove)
    mesh_poisson.paint_uniform_color([0.5, 0.5, 0.5])

          #o3d.visualization.draw_geometries([pcd_mesh,    mesh_poisson]    if
O3D_DISPLAY_POINTS     else     [mesh_poisson],     mesh_show_back_face=True,
mesh_show_wireframe=O3D_DISPLAY_WIREFRAME)
    #--------------------------------------------

    # Import the mesh into RoboDK
    with TemporaryDirectory(prefix='robodk_') as td:
        tf = td + '/mesh.ply'
        o3d.io.write_triangle_mesh(tf, mesh_poisson, write_ascii=True)
        mesh_item = RDK.AddFile(tf)

    mesh_item.setPose(robot_base.Pose())
    mesh_item.setColor([0.5, 0.5, 0.5, 1])
    mesh_item.setName("Reconstructed Mesh")

    return mesh_item
```

```
def calculo_rotXY_hombro(vector_unitarioZ,p):
     # Función que a partir de la orientación de la componente Z que viene
determinada por la
    # pieza y un punto objetivo, se calcula la orientación X e Y, teniendo en
cuenta la forma
     # del robot colaborativo de esta forma se consigue cierta flexibilidad y
más alcance
     # input: - vector_unitarioZ: Vector unitario de la componente Z, donde
apunta el robot
    #        - p: Punto en el espacio objetivo del robot
    # output: - vector_unitarioX: Vector unitario de la componente X
    #         - vector_unitarioY: Vector unitario de la componente Y

    x, y, z = p
    dist_hombro_origen=119.85
    dist_muñeca_origen=110.4
     # Calculo de la posicion xy del hombro del robot (articulación 2) en el
punto deseado
    alpha = math.acos(x / math.sqrt(x*x + y*y))
    theta = math.acos(dist_muñeca_origen/math.sqrt(x*x + y*y))
    beta = alpha - theta


    # Calculo del vector entre el hombro y el punto
    hombrox = math.cos(beta) * dist_muñeca_origen
    hombroy = math.sin(beta) * dist_muñeca_origen
    vectorhombroapunto = np.array([x - hombrox, y - hombroy, 0])

    # Calculo de valores de la rotacion en X e Y
    vector_ortogonalx = np.cross(vector_unitarioZ, vectorhombroapunto)
    magnitud = np.linalg.norm(vector_ortogonalx)
    vector_unitarioX = vector_ortogonalx / magnitud

    vector_ortogonaly = np.cross(vector_unitarioZ, vector_ortogonalx)
    magnitud = np.linalg.norm(vector_ortogonaly)
    vector_unitarioY = vector_ortogonaly / magnitud

    return vector_unitarioX,vector_unitarioY
```

```
def figure_creator(points):
    # Función que crea objetos para ayudar a visualizar la trayectoria de la soldadura en el robot
    # input: - points: Puntos de interes de la soldadura
    # output: - objects: Lista que contendra todos los objetos creados

    objects = []

    if PLANE_OBJECT: # Creación de las figuras para el objeto plano
        p1,p2=points
        vz = p1-p2
        vx = np.array([-vz[1],vz[0],0])
        vy = np.cross(vz,vx)

        vdz = vz/np.linalg.norm(vz)
        vdx = vx/np.linalg.norm(vx)
        vdy = vy/np.linalg.norm(vy)
        # Creación de esferas , simulando los puntos
        for p in points:
            T = Mat([[1,0,0,p[0]],
                [0,1,0,p[1]],
                  [0,0,1,p[2]+200],  # Se suma 200 a Z, porque el sistema de referencia donde se van a introducir los objetos es 200 mm menor que el origen del robot
                [0,0,0,1]])
                                                                    esfera          = RDK.AddFile('C:/Users/abelg/Desktop/TFM/ObjetosRoboDK/Esfera5mmD.sld')
            esfera.setPose(T)
            objects.append(esfera)
        # Creación de cilindro, simulando la trayectoria
        T = Mat([[vdx[0],vdy[0],vdz[0],p2[0]],
                [vdx[1],vdy[1],vdz[1],p2[1]],
                [vdx[2],vdy[2],vdz[2],p2[2]+200],
                [0,0,0,1]])
                                                                cilindro            = RDK.AddFile('C:/Users/abelg/Desktop/TFM/ObjetosRoboDK/Cilindro2x1mm.sld')
        cilindro.setPose(T)
        cilindro.Scale([1,1,np.linalg.norm(p2 - p1)])
        objects.append(cilindro)

    if not PLANE_OBJECT: # Creación de las figuras para los demas objetos cuadrados
        paux = points[-1]
        for p in points:
            # Creación de esferas
            T = Mat([[1,0,0,p[0]],
                [0,1,0,p[1]],
                [0,0,1,p[2]+200],
                [0,0,0,1]])
                                                                esfera          = RDK.AddFile('C:/Users/abelg/Desktop/TFM/ObjetosRoboDK/Esfera5mmD.sld')
            esfera.setPose(T)
            objects.append(esfera)

            # Creación de cilindros
            vz = paux-p
            vdz = vz/np.linalg.norm(vz)
            vdx = np.array([-vdz[1],vdz[0],0])
            vdy = np.cross(vdz,vdx)
            T = Mat([[vdx[0],vdy[0],vdz[0],p[0]],
            [vdx[1],vdy[1],vdz[1],p[1]],
            [vdx[2],vdy[2],vdz[2],p[2]+200],
            [0,0,0,1]])
                                                            cilindro            = RDK.AddFile('C:/Users/abelg/Desktop/TFM/ObjetosRoboDK/Cilindro2x1mm.sld')
            cilindro.setPose(T)
            cilindro.Scale([0.5,0.5,np.linalg.norm(p - paux)])
            objects.append(cilindro)
            paux=p

    return objects
```

```
def plane_object_points(plane1,plane2,centroide,pcd_planos):
    # Función que obtiene el punto inicial y final de la soldadura en el objeto
dos planos
    # input: - plane1: Coeficientes del primer plano
    #        - plane2: Coeficientes del segundo plano
    #        - centroide: Punto aproximado donde se encuentran los objetos, no
es de gran relevancia
    #        - pcd_planos: Nube de puntos del objeto completo
    # output: - p1: Punto inicial de la soldadura
    #         - p2: Punto final de la soldadura
    #         - nube_coincidente: Nube de puntos de la intersección

    # Primero se obtiene el origen y vector director de la recta, igualando los
dos planos
    # Coeficientes
    A1, B1, C1, D1 = plane1
    A2, B2, C2, D2 = plane2
    # Forma de asegurar que la componente Z(C..) siempre tiene valores negativos
    if C1 > 0:
        A1 = -A1
        B1 = -B1
        C1 = -C1
        D1 = -D1
    if C2 > 0:
        A2 = -A2
        B2 = -B2
        C2 = -C2
        D2 = -D2

     # Busqueda del punto de origen, mas cercano al centroide, para conseguir
menor error en el calculo de la trayectoria

    # Se asume que la recta pasa por z = 0 y se resuelve el sistema
    # A1*x +B1*y = -D1
    # A2*x +B2*y = -D2

    A = np.array([[A1, B1], [A2, B2]])
    B = np.array([-D1, -D2])
    try:
        sol = np.linalg.solve(A, B)
        pz= [sol[0], sol[1], 0]
    except:
        pz= [10000,10000,10000]

    # y =0
    A = np.array([[A1, C1], [A2, C2]])
    B = np.array([-D1, -D2])
    try:
        sol = np.linalg.solve(A, B)
        py = [sol[0], 0, sol[1]]
    except:
        py = [10000,10000,10000]

    # x =0
    A = np.array([[B1, C1], [B2, C2]])
    B = np.array([-D1, -D2])
    try:
        sol = np.linalg.solve(A, B)
        px = [0, sol[0], sol[1]]
    except:
        px = [10000,10000,10000]

    # Se calcula la distancia del punto a uno cercano a la pieza y conocido
    distpx = np.linalg.norm(centroide - px)
    distpy = np.linalg.norm(centroide - py)
    distpz = np.linalg.norm(centroide - pz)

    # Se elige el más cercano
    if distpx < distpy and distpx < distpz:
        origen = px
    elif distpy < distpx and distpy < distpz:
```

```
        origen = py
    else:
        origen = pz

    origen[2]=origen[2]
    # Vector director de la intersección de los dos planos
    v = np.cross(np.array([A1, B1, C1]), np.array([A2, B2, C2]))
    magnitud = np.linalg.norm(v)
    v_interseccion = v / magnitud


    # A partir de aqui se va a recrear una nube de puntos de una recta a partir de:
    # - origen, un punto de origen
    # - v_intersección, vector de la interseción de los planos
    # - longitud: longitud de la recta, va en milimetros y el punto de origen quedara en el centro
    # - num_puntos: Número de puntos a generar en la recta, me gusta que sea como mínim la misma que la longitud

    longitud = 2000
    num_puntos = 2000

    # Generar puntos a lo largo de la recta
    t_valores = np.linspace(-longitud/2, longitud/2, num_puntos)
    puntos_recta = np.array([origen + t * v_interseccion for t in t_valores])

    # Filtros de la recta, asi solo tengo la recta en la zona de trabajo
    filtro_x = (puntos_recta[:, 0] >= 0) & (puntos_recta[:, 0] <= 500)
    filtro_y = (puntos_recta[:, 1] >= 60) & (puntos_recta[:, 1] <= 500)
    filtro_z = (puntos_recta[:, 2] >= -200) & (puntos_recta[:, 2] <= 350)

    # Aplicar los filtros
    puntos_recta = puntos_recta[filtro_x & filtro_y & filtro_z]

    # Crear la nube de puntos
    #pcd_recta = o3d.geometry.PointCloud()
    #pcd_recta.points = o3d.utility.Vector3dVector(puntos_recta)

    # -------- Comparación de los puntos de la recta con los planos ---------

    umbral_distancia = 5 # distancia a la que se va a considerar que coinciden
    puntos_coincidentes = []

    # Paso de nube de puntos a array
    planos = np.asarray(pcd_planos.points)

    # Verificar si cada punto de la recta está cerca del plano
    for punto_recta in puntos_recta:
        distancias_al_plano = np.linalg.norm(planos - punto_recta, axis=1)
        distancia_minima = np.min(distancias_al_plano)

        if distancia_minima < umbral_distancia:
            puntos_coincidentes.append(punto_recta)

    # Convertir la lista de puntos coincidentes a una nueva nube de puntos
    nube_coincidente = o3d.geometry.PointCloud()
                                    nube_coincidente.points                    =
o3d.utility.Vector3dVector(np.asarray(puntos_coincidentes))

    # Obtengo el cuarto punto que coincide por ambos lados, para asegurarme que estoy ya en la pieza
    p1 = puntos_coincidentes[3]
    p2 = puntos_coincidentes[-4]

    return p1, p2 ,nube_coincidente
```

```
def box_object_points(planes):
    # Función que encuentra los puntos de interes en los objetos box. Encuentra
el plano
    # principal, el inferior o superior y lo utilizara para hacer sistemas de
ecuaciones
    # con otros dos planos para encontrar los puntos
    # input: - planes: Lista con los coeficientes de los planos
    # output: - p: Lista con la información de los puntos de interes
    #         - centroid: Centroide que forman los puntos
    #         - main_plane: Plano principal

    # Obtención del plano prncipal
    main_plane = max(planes, key=lambda x: abs(x[2]))

    if any(np.array_equal(main_plane, plane) for plane in planes):
        planes = [plane for plane in planes if not np.array_equal(main_plane,
plane)]
    else:
        print("El plano no se ha podido eliminar, va a dar error")

    A1, B1, C1, D1 = main_plane
    A2, B2, C2, D2 = planes[0]
    A3, B3, C3, D3 = planes[1]
    A4, B4, C4, D4 = planes[2]
    A5, B5, C5, D5 = planes[3]

    A = np.array([[A1, B1, C1], [A2, B2, C2], [A3, B3, C3]])
    B = np.array([-D1, -D2, -D3])
    p1 = np.linalg.solve(A, B)

    if np.linalg.norm(p1)>1000: # Caso de que los planos 1 y 2 sean paralelos
        A = np.array([[A1, B1, C1], [A2, B2, C2], [A4, B4, C4]])
        B = np.array([-D1, -D2, -D4])
        p1 = np.linalg.solve(A, B)

        A = np.array([[A1, B1, C1], [A2, B2, C2], [A5, B5, C5]])
        B = np.array([-D1, -D2, -D5])
        p2 = np.linalg.solve(A, B)

        A = np.array([[A1, B1, C1], [A3, B3, C3], [A4, B4, C4]])
        B = np.array([-D1, -D3, -D4])
        p3 = np.linalg.solve(A, B)

        A = np.array([[A1, B1, C1], [A3, B3, C3], [A5, B5, C5]])
        B = np.array([-D1, -D3, -D5])
        p4 = np.linalg.solve(A, B)

    else: # Caso de que los planos 2 y 3 se crucen
        A = np.array([[A1, B1, C1], [A2, B2, C2], [A4, B4, C4]])
        B = np.array([-D1, -D2, -D4])
        p2 = np.linalg.solve(A, B)

       if np.linalg.norm(p2)>1000: # Caso de que los planos 2 y 4 sean paralelos
            A = np.array([[A1, B1, C1], [A2, B2, C2], [A5, B5, C5]])
            B = np.array([-D1, -D2, -D5])
            p2 = np.linalg.solve(A, B)

            A = np.array([[A1, B1, C1], [A3, B3, C3], [A4, B4, C4]])
            B = np.array([-D1, -D3, -D4])
            p3 = np.linalg.solve(A, B)

            A = np.array([[A1, B1, C1], [A4, B4, C4], [A5, B5, C5]])
            B = np.array([-D1, -D4, -D5])
            p4 = np.linalg.solve(A, B)

        else:
            A = np.array([[A1, B1, C1], [A3, B3, C3], [A5, B5, C5]])
            B = np.array([-D1, -D3, -D5])
            p3 = np.linalg.solve(A, B)

            A = np.array([[A1, B1, C1], [A4, B4, C4], [A5, B5, C5]])
            B = np.array([-D1, -D4, -D5])
```

```
            p4 = np.linalg.solve(A, B)

        p = [p1,p2,p3,p4]
        centroid = np.mean(p, axis=0)

        return p, centroid, main_plane
```

```
def plane_object_targets(plane1, plane2, p1, p2):
    # Función que a partir de los planos y el punto inicial y final,
    # se calculan los distintos targets de aproximación, ejecución del trabajo
y alejamiento
    # en esta función se usas otras ya creadas como calculo_rotXY_hombro()
    # input: - plane1: Coeficientes del primer plano
    #        - plane2: Coeficientes del segundo plano
    #        - p1: Punto inicial de la soldadura
    #        - p2: Punto final de la soldadura
    # output: - Tapro: Target de aproximación al objeto
    #        - T1: Target que se corresponde con el primer punto de la soldadura,
ya en el objeto
    #        - T2: Target que se corresponde con el ultimo punto de la soldadura
    #         - Tdepro: Target de alejamiento del robot del objeto

    # Puntos
    x1, y1, z1 = p1
    x2, y2, z2 = p2

    # Coeficientes
    A1, B1, C1, D1 = plane1
    A2, B2, C2, D2 = plane2

    # Forma de asegurar que la Z siempre tiene valores negativos
    if C1 > 0:
        A1 = -A1
        B1 = -B1
        C1 = -C1
        D1 = -D1
    if C2 > 0:
        A2 = -A2
        B2 = -B2
        C2 = -C2
        D2 = -D2

    # Se obtiene el vector z resultante de la normal los planos, para así pasar
por la intersección
    v_2planos = np.array([A1+A2, B1+B2, C1+C2])

    # Se obtiene el vector unitario, del vector resultante, este corresponde al
vector unitario de la componente Z
    magnitud = np.linalg.norm(v_2planos)
    vector_unitarioZ = v_2planos / magnitud
    # print("Vector Unitario Z: "+str(vector_unitarioZ))


     # Con calculo_rotXY_hombro() se obtiene los valores de las otras dos
componentes de orientación
                                vector_unitarioX,vector_unitarioY             =
calculo_rotXY_hombro(vector_unitarioZ,p1)

    # Construcción de la matriz de aproach con la orientación deseada y alejada
10 cm en la componente Z
                                                    Tapro                     =
Mat([[vector_unitarioX[0],vector_unitarioY[0],vector_unitarioZ[0],x1],
                    [vector_unitarioX[1],vector_unitarioY[1],vector_unitarioZ[
1],y1],
                    [vector_unitarioX[2],vector_unitarioY[2],vector_unitarioZ[
2],z1+100],
                    [0,0,0,1]])
    # Construcción de la matriz del primer punto con la orientación deseada
    T1 = Mat([[vector_unitarioX[0],vector_unitarioY[0],vector_unitarioZ[0],x1],
                    [vector_unitarioX[1],vector_unitarioY[1],vector_unitarioZ[
1],y1],
                    [vector_unitarioX[2],vector_unitarioY[2],vector_unitarioZ[
2],z1],
                    [0,0,0,1]])


                                vector_unitarioX,vector_unitarioY             =
calculo_rotXY_hombro(vector_unitarioZ,p2)
    # Construcción de la matriz del segudo punto con la orientación deseada
```

```
    T2 = Mat([[vector_unitarioX[0],vector_unitarioY[0],vector_unitarioZ[0],x2],
                      [vector_unitarioX[1],vector_unitarioY[1],vector_unitarioZ[
1],y2],
                      [vector_unitarioX[2],vector_unitarioY[2],vector_unitarioZ[
2],z2],
                     [0,0,0,1]])

     # Construcción de la matriz de alejamiento con la orientación deseada y
alejada 10 cm en la componente Z
                                                  Tdepro                    =
Mat([[vector_unitarioX[0],vector_unitarioY[0],vector_unitarioZ[0],x2],
                      [vector_unitarioX[1],vector_unitarioY[1],vector_unitarioZ[
1],y2],
                      [vector_unitarioX[2],vector_unitarioY[2],vector_unitarioZ[
2],z2+100],
                     [0,0,0,1]])

    return Tapro, T1, T2, Tdepro
```

```
def inside_box_object_targets(points,centroid,bot_plane,botplane_angle):
    # Función que a partir de los distintos puntos y la orientación de la caja
dada
    # por bot_plane, se calculan los distintos targets de aproximación, ejecución
del
    # trabajo y alejamiento teniendo en cuenta si el angulo de incisión deseado
    # en esta función se usas otras ya creadas como calculo_rotXY_hombro()
    # input: - points: Lista con la información de los puntos de interes
    #        - centroid: Centroide que forman los puntos
    #        - bot_plane: Plano principal
    #        - botplane_angle: Ángulo de incisión deseado
    # output: - targets: Lista de los targets de la soldadura en el objeto
     #          - targetAD: Target de aproximación alejamiento del robot del
objeto
    #         - p_target: Puntos finales objetivo del robot

    # Ordenación correcta de los puntos
    max_y = max(points, key=lambda x: x[1])
    max_x = max(points, key=lambda x: x[0])
    min_y = min(points, key=lambda x: x[1])
    min_x = min(points, key=lambda x: x[0])

    if np.array_equal(max_y, max_x):
         print("Caja es paralelela y no se puede identificar el orden de los
puntos")
        exit()
    points=[max_y, max_x, min_y, min_x]

    # Longitud arista
    b1 = np.linalg.norm(points[1] - points[0])
    b2 = np.linalg.norm(points[2] - points[1])
    b = (b1+b2)/2

     # Cálculo de un factor que servira para obtener la inclinación de la
herramienta respecto al plano inferior
    angle = botplane_angle*pi/180
    factor = tan(angle)*b/2

    # Orientación correcta de los vectores, con la componence C mirando hacia
abajo, la correspondiente a la z
    A,B,C,D=bot_plane
    if C > 0:
        A = -A
        B = -B
        C = -C
        D = -D

    # Creación de target para aproximarse centrando en la caja
    puntocentrico = centroid - ([factor*A,factor*B,factor*C])
    vz = [A,B,C]
    vx,vy = calculo_rotXY_hombro(vz,puntocentrico)

    targetAD = (Mat([[vx[0],vy[0],vz[0],puntocentrico[0]],
                     [vx[1],vy[1],vz[1],puntocentrico[1]],
                     [vx[2],vy[2],vz[2],puntocentrico[2]],
                     [0,0,0,1]]))


    # Creación de los targets para las aristas
    targets = []
    p_target = []
    for point in points:

        vzdir = point - puntocentrico
        # Normaliza el vector director dividiéndolo por su magnitud
        vz = vzdir / np.linalg.norm(vzdir)

        vx,vy = calculo_rotXY_hombro(vz,point)

        # Targets que pasan un milimetro mas cerca del punto centrico, de esta
forma es mas seguro que no atraviese la caja
        targets.append(Mat([[vx[0],vy[0],vz[0],point[0]-vz[0]],
```

```
                    [vx[1],vy[1],vz[1],point[1]-vz[1]],
                    [vx[2],vy[2],vz[2],point[2]-vz[2]],
                    [0,0,0,1]]))
            p_target.append(np.array([point[0]-vz[0],point[1]-vz[1],point[2]-
vz[2]]))
    return targets, targetAD, p_target
```

```
def outside_box_object_targets(points,bot_plane):
    # Función que a partir de los distintos puntos y la orientación de la caja
    # dada por bot_plane, se calculan los distintos targets de aproximación,
    # ejecución del  trabajo y alejamiento
    # en esta función se usas otras ya creadas como calculo_rotXY_hombro()
    # input: - points: Lista con la información de los puntos de interes
    #        - bot_plane: Plano principal
    # output: - targets: Lista de los targets de la soldadura en el objeto
     #          - targetAD: Target de aproximación alejamiento del robot del
objeto
    #         - p_target: Puntos finales objetivo del robot

    # Ordenación correcta de los puntos
    max_y = max(points, key=lambda x: x[1])
    max_x = max(points, key=lambda x: x[0])
    min_y = min(points, key=lambda x: x[1])
    min_x = min(points, key=lambda x: x[0])

    if np.array_equal(max_y, max_x):
         print("Caja es paralelela y no se puede identificar el orden de los
puntos")
        exit()
    points=[max_y, max_x, min_y, min_x]

    # Orientación correcta de los vectores, con la componence C mirando hacia
abajo, la correspondiente a la z
    A,B,C,D=bot_plane
    if C > 0:
        A = -A
        B = -B
        C = -C
        D = -D

    # Creación de los targets para las aristas
    AD = False
    vz = [A,B,C]
    targets = []
    p_target = []
    for point in points:

        vx,vy = calculo_rotXY_hombro(vz,point)

        if not AD:
        # Targets de aproximacion y alejamiento
            targetAD=(Mat([[vx[0],vy[0],vz[0],point[0]-100*vz[0]],
                    [vx[1],vy[1],vz[1],point[1]-100*vz[1]],
                    [vx[2],vy[2],vz[2],point[2]-100*vz[2]],
                    [0,0,0,1]]))
            AD = True

        # Targets que pasan un milimetro mas alejado de los puntos objetivo, de
esta forma es mas seguro que no atraviese la caja
        targets.append(Mat([[vx[0],vy[0],vz[0],point[0]-vz[0]],
                    [vx[1],vy[1],vz[1],point[1]-vz[1]],
                    [vx[2],vy[2],vz[2],point[2]-vz[2]],
                    [0,0,0,1]]))
             p_target.append(np.array([point[0]-vz[0],point[1]-vz[1],point[2]-
vz[2]]))
    return targets, targetAD, p_target
```

```
def identify_object_aplication():
    # Función que posiciona el robot en unos determiandos tarjets creados en la
estación y
    # adquirir las imágenes en dichas posiciones, así como la cración de la pcd
del objeto
    # usando otras funciones
    # output: - pcd: Nube de puntos del objeto
    #         - centroide: Centroide aproximado del objeto

    newtargets = [] # Lista de los target para hacer capturas
    if PLANE_OBJECT or ICP_OBJECT:
        newtargets.append(RDK.Item('Position1', ITEM_TYPE_TARGET))
    else:
        newtargets.append(RDK.Item('Position1', ITEM_TYPE_TARGET))
        newtargets.append(RDK.Item('Position2', ITEM_TYPE_TARGET))
        newtargets.append(RDK.Item('Position3', ITEM_TYPE_TARGET))
        newtargets.append(RDK.Item('Position4', ITEM_TYPE_TARGET))
        newtargets.append(RDK.Item('Position5', ITEM_TYPE_TARGET))

    depth ,cam_pose = get_image_and_camera_pose(newtargets)
    pcd = get_point_cloud(depth, cam_pose)
    #if VISUALICE_PCD: o3d.visualization.draw_geometries([pcd]) # Nube de puntos
completa
    # Guardar la nube de puntos en formato PCD
    if SAVE_PCD:
        output_directory = r'C:\Users\abelg\Desktop\TFM\pcd'
        output_pcd_file = 'point_cloud.pcd'
        output_path = os.path.join(output_directory, output_pcd_file)
        o3d.io.write_point_cloud(output_path, pcd)

     pcd = pcd.voxel_down_sample(voxel_size=2)  # Ajusta el tamaño del voxel
según sea necesario
    if VISUALICE_PCD: o3d.visualization.draw_geometries([pcd]) # Nube de puntos
reducida
    centroide = np.array([150,250,-50])
    return pcd, centroide
```

```
def plane_object_aplication():
    # Función realiza la identificación y ejecución de los movimientos para el
objeto plano usando diferentes funciones

    pcd, centroide = identify_object_aplication()

    robot.MoveJ(RDK.Item('Position6', ITEM_TYPE_TARGET)) # Posición de espera

    planes, coefficients = get_planes(pcd, 2)

                            p1,            p2,            pcd_recta            =
plane_object_points(coefficients[0],coefficients[1],centroide,pcd)
    if MESH_RECONSTRUCTION: mesh = mesh_creation(pcd)

    objetos = figure_creator([p1,p2])

        Tapro,   T1,   T2,   Tdepro   =   plane_object_targets(coefficients[0],
coefficients[1], p1, p2)

    # Uso de una interfaz para elegir simulacion o ejecución en el robot real
    actual_joints = robot.Joints()
    while(1):
        root = tk.Tk()
        option = tk.IntVar(value=0)
        interfaz = SimulationInterface(root,option)
        root.mainloop()

        if option.get() == 1: # Elección de visualizar movimiento en RoboDK
            RDK.setRunMode(RUNMODE_SIMULATE)
        elif option.get() == 2: # Elección de realizar el movimiento en el robot
real
            #RDK.setRunMode(RUNMODE_RUN_ROBOT)
            print("real")
        elif option.get() == 3: # Elección de finalizar
            print("Finalización forzada")
            for objeto in objetos:
                objeto.Delete()

            if MESH_RECONSTRUCTION:mesh.Delete()
            break

        robot.setPoseTool(welding_tool)
        robot.MoveJ(Tapro)
        robot.setZoneData(0)
        robot.setSpeed(75, 500)
        robot.MoveL(T1)
        robot.MoveL(T2)
        #robot.setZoneData(0)
        robot.setSpeed(150, 1000)
        robot.MoveL(Tdepro)

        if option.get()== 1:
            robot.setJoints(actual_joints)
        elif option.get()== 2:
            for objeto in objetos:
                objeto.Delete()

            if MESH_RECONSTRUCTION:mesh.Delete()
            break
    PLANE_OBJECT = False
```

```
def inside_box_object_aplication():
    # Función realiza la identificación y ejecución de los movimientos para el
objeto interior de la caja usando diferentes funciones

    pcd,_ = identify_object_aplication()
    robot.MoveJ(RDK.Item('Position6', ITEM_TYPE_TARGET))

    # Set a la herramienta de pintura/soldadura
    robot.setPoseTool(welding_tool)

    # Obtención de los distintos planos y los puntos objetivo
    planes, coefficients = get_planes(pcd, 5)

    if MESH_RECONSTRUCTION: mesh = mesh_creation(pcd)
    p, centroid, bottom_plane = box_object_points(coefficients)

    # Obtención de los movimientos y su posterior realización
      targets,  targetAD,  p_target  =  inside_box_object_targets(p,  centroid,
bottom_plane,ANGLE_INSIDE)

    # Creación de figuras para visualizar de manera mas facil en RoboDK
    objetos = figure_creator(p_target)

    # Uso de una interfaz para elegir simulacion o ejecución en el robot real
    actual_joints = robot.Joints()
    while(1):
        root = tk.Tk()
        option = tk.IntVar(value=0)
        interfaz = SimulationInterface(root,option)
        root.mainloop()

        if option.get() == 1:
            RDK.setRunMode(RUNMODE_SIMULATE)
        elif option.get() == 2:
            #RDK.setRunMode(RUNMODE_RUN_ROBOT)
            print("real")
        elif option.get() == 3:
            print("Finalización forzada")
            for objeto in objetos:
                objeto.Delete()

            if MESH_RECONSTRUCTION: mesh.Delete()
            break


        robot.MoveJ(targetAD) # Movimiento de aproximación
        try:
            robot.setZoneData(5)
            robot.setSpeed(75, 500)

            for target in targets: # Bucle para realizar los movimientos
                robot.MoveL(target)
                time.sleep(1)
        except:
             print("Si esto sucede se podria realizar la solución con alguna
orientacion de z no perpendicular al plano")

        robot.MoveL(targets[0]) # Ultimo movimiento del trabajo para cerrar ell
cuadrado
        robot.MoveL(targetAD) # Movimiento de alejamiento
        robot.setZoneData(0)
        robot.setSpeed(150, 1000)
        if option.get()== 1:
            robot.setJoints(actual_joints)
        elif option.get()== 2:
            for objeto in objetos:
                objeto.Delete()

            if MESH_RECONSTRUCTION:mesh.Delete()
            break
    INSIDE_BOX_OBJECT = False
```

```
def outside_box_object_aplication():
    # Función realiza la identificación y ejecución de los movimientos para el
objeto exterior de la caja usando diferentes funciones

    pcd,_ = identify_object_aplication()
    robot.MoveJ(RDK.Item('Position6', ITEM_TYPE_TARGET))
    # Set a la herramienta de pintura/soldadura
    robot.setPoseTool(welding_tool)

    # Obtención de los distintos planos y los puntos objetivo
    planes, coefficients = get_planes(pcd, 5)

    if MESH_RECONSTRUCTION: mesh = mesh_creation(pcd)
    p, centroid, bottom_plane = box_object_points(coefficients)

    # Obtención de los movimientos y su posterior realización
    targets, targetAD, p_target= outside_box_object_targets(p, bottom_plane)

    # Creación de figuras para visualizar de manera mas facil en RoboDK
    objetos = figure_creator(p_target)

    # Uso de una interfaz para elegir simulacion o ejecución en el robot real
    actual_joints = robot.Joints()

    while(1):
        root = tk.Tk()
        option = tk.IntVar(value=0)
        interfaz = SimulationInterface(root,option)
        root.mainloop()

        if option.get() == 1:
            RDK.setRunMode(RUNMODE_SIMULATE)
        elif option.get() == 2:
            #RDK.setRunMode(RUNMODE_RUN_ROBOT)
            print("real")
        elif option.get() == 3:
            print("Finalización forzada")
            for objeto in objetos:
                objeto.Delete()
            if MESH_RECONSTRUCTION: mesh.Delete()
            break

        robot.MoveJ(targetAD) # Movimiento de aproximación
        try:
            robot.setZoneData(5)
            robot.setSpeed(75, 500)

            for target in targets: # Bucle para realizar los movimientos
                robot.MoveL(target)
        except:
             print("Si esto sucede se podria realizar la solución con alguna
orientacion de z no perpendicular al plano")

        robot.MoveL(targets[0]) # Ultimo movimiento del trabajo para cerrar ell
cuadrado
        robot.MoveL(targetAD) # Movimiento de alejamiento
        robot.setZoneData(0)
        robot.setSpeed(150, 1000)

        if option.get()== 1:
            robot.setJoints(actual_joints)
        elif option.get()== 2:
            for objeto in objetos:
                objeto.Delete()
            if MESH_RECONSTRUCTION: mesh.Delete()

            break
    OUTSIDE_BOX_OBJECT = False
```

```
def icp_identification(source,target,threshold):
    # Función que aproxima la nueba nube de puntos a una ya previamente guardada
    # usando el algoritmo ICP
    # input: - source: Nube de puntos previamente guardada y conocida su posición
    #        - target: Nueva nube de puntos a identificar
    #        - threshold: Umbral utilizado para la aproximación
    # output: - matriz: Matriz de transformación que coloca la nube source en
la nube target

    # Estimar normales a las nubes
    source.estimate_normals()
    target.estimate_normals()

    # Obtener centroides y generar matriz de transformación inicial
    centroid_source = np.asarray(source.points).mean(axis=0)
    centroid_target = np.asarray(target.points).mean(axis=0)
    centroid_diff = centroid_target - centroid_source

    trans_init = np.array([[1, 0, 0, centroid_diff[0]],
                           [0, 1, 0, centroid_diff[1]],
                           [0, 0, 1, centroid_diff[2]],
                           [0, 0, 0, 1]])

    # Visualización antes de realizar ICP
    source.paint_uniform_color([1, 0.706, 0])
    target.paint_uniform_color([0, 0.651, 0.929])
    sourcecopy=copy.deepcopy(source)
    sourcecopy.transform(trans_init)
    if VISUALICE_PCD: o3d.visualization.draw_geometries([source, target])
    #o3d.visualization.draw_geometries([sourcecopy, target])

    """# Point to point ICP
    print("Apply point-to-point ICP")
    reg_p2p = o3d.pipelines.registration.registration_icp(
        source, target, threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPoint())
    print(reg_p2p)
    print("Transformation is:")
    print(reg_p2p.transformation)
    sourcecopy=copy.deepcopy(source)
    sourcecopy.transform(reg_p2p.transformation)
    o3d.visualization.draw_geometries([sourcecopy, target])
    """
    # Point to plane ICP
    print("Apply point-to-plane ICP")
    reg_p2l = o3d.pipelines.registration.registration_icp(
        source, target, threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    print(reg_p2l)
    print("Transformation is:")
    print(reg_p2l.transformation, "\n")
    sourcecopy=copy.deepcopy(source)
    sourcecopy.transform(reg_p2l.transformation)
    if VISUALICE_PCD: o3d.visualization.draw_geometries([sourcecopy, target])
    print(trans_init, "\n")

    matriz = Mat(reg_p2l.transformation.tolist())
    return matriz
```

```
def icp_object_targets(positions,trans):
     # Función que a partir de los puntos en la nube source le aplica la
transformación
     # correspondiente para conocer los puntos en la nueba nube, despues
proporciona
    # los targets creados en esos puntos
    # input: - positions: Puntos objetivos de la nube source, estos deben ser
previamente elegidos
    #        - trasn: Matriz de transformación para los puntos
    # output: - targets: Lista con los target de los puntos en el objeto

    targets=[]
    # Convertir las posiciones a poses
    for pos in positions:
        pose = Pose(*pos)
        pose = trans*pose
        vZ=(pose[0,2],pose[1,2],pose[2,2])
        p=(pose[0,3],pose[1,3],pose[2,3])
        vX,vY = calculo_rotXY_hombro(vZ,p)

        # Construcción de la matriz de target la orientación deseada
        targets.append(Mat([[vX[0], vY[0], vZ[0], p[0]],
              [vX[1], vY[1], vZ[1], p[1]],
              [vX[2], vY[2], vZ[2], p[2]],
              [0, 0, 0, 1]]))

    return targets
```

```
def icp_object_aplication():
    # Función realiza la identificación y ejecución de los movimientos para el
objeto ICP
    # usando diferentes funciones para su correcto funcionamiento se debe tener
una nube
     # source y sus puntos de interes del objeto previamente registradas en el
sistema

                                                    source                      =
o3d.io.read_point_cloud(r'C:\Users\abelg\Desktop\TFM\ICP\sourceplacapequeñav2.
pcd')
    pcd_target,_ = identify_object_aplication()
    trans = icp_identification(source,pcd_target,5)
    # Cargar los puntos
    file_path = r'C:\Users\abelg\Desktop\TFM\ICP\puntosplacapequeña.txt'

    # Cargar las posiciones desde el archivo de texto
    positions = np.loadtxt(file_path)
    targets = icp_object_targets(positions,trans)

    # Set a la herramienta de pintura/soldadura
    robot.setPoseTool(welding_tool)

    # Uso de una interfaz para elegir simulacion o ejecución en el robot real
    actual_joints = robot.Joints()
    while(1):
        root = tk.Tk()
        option = tk.IntVar(value=0)
        interfaz = SimulationInterface(root,option)
        root.mainloop()

        if option.get() == 1:
            RDK.setRunMode(RUNMODE_SIMULATE)
        elif option.get() == 2:
            #RDK.setRunMode(RUNMODE_RUN_ROBOT)
            print("real")
        elif option.get() == 3:
            print("Finalización forzada")
            break
        robot.setZoneData(0)
        robot.setSpeed(75, 500)
        robot.MoveJ(Offset(targets[0],0,0,200))
        robot.MoveL(targets[0])
        robot.MoveC(targets[1],targets[2])
        robot.MoveL(Offset(targets[2],0,0,100))
        robot.MoveJ(Offset(targets[3],0,0,100))
        robot.MoveL(targets[3])
        robot.MoveC(targets[4],targets[5])
        robot.MoveL(targets[6])
        robot.MoveC(targets[7],targets[8])
        robot.MoveL(Offset(targets[8],0,0,200))
        robot.setZoneData(0)
        robot.setSpeed(150, 1000)

        if option.get()== 1:
            robot.setJoints(actual_joints)
        elif option.get()== 2:
            break
    ICP_OBJECT = False
```

```
# ---------------------------------- main ----------------------------------

# ------ Interfaz para la configuración -----
root = tk.Tk()
configinterface = ConfigurationInterface(root)
root.mainloop()

# ---- Bucle donde se realiza la aplicación ----
while(1):
    # --------- Interfaz Inicial ---------
    root = tk.Tk()
    end_aplication = tk.BooleanVar(value=False)
    interfaz = MainInterface(root,end_aplication)
    root.mainloop()
    if end_aplication.get():
        print("Fin de la aplicación")
        break

    # Get cam_mtx , intrinsic parameters camera activation
    if CAMERA_SIMULATION:
        cam_item.setParam('Open', 1)
        cam_settings = settings_to_dict(cam_item.setParam('Settings'))
        w, h = cam_settings['SIZE']
        fy = h / (2 * np.tan(np.radians(cam_settings['FOV']) / 2))
         cam_mtx = o3d.camera.PinholeCameraIntrinsic(width=w, height=h, fx=fy,
fy=fy, cx=w / 2, cy=h / 2)

        RDK.setRunMode(RUNMODE_SIMULATE)

        # Para probar el detector de manos con la cámara del ordenador
        #hilofuncionamiento = True
        #hilo_camara = threading.Thread(target=webcam_hand_tracking, args=())
        #hilo_camara.start()
        #time.sleep(4) # Espera a que inicie la cámara

    else: # Real camera
        hilofuncionamiento = True

          #cam_mtx = o3d.camera.PinholeCameraIntrinsic(width=1280, height=720,
fx=934.082, fy=934.082, cx=632.428, cy= 365.502)
          cam_mtx = o3d.camera.PinholeCameraIntrinsic(width=1280, height=720,
fx=920, fy=922.5, cx=655, cy= 365)
        # Crear una cola para comunicarse entre el hilo de la cámara y main
        queue = Queue()

        # Crear un evento para solicitar la imagen de profundidad
        depth_event = threading.Event()

        # Creación del hilo y comienzo
        hilo_camara = threading.Thread(target=realsense_streaming, args=(queue,
depth_event))
        hilo_camara.start()
        #RDK.setRunMode(RUNMODE_RUN_ROBOT)
        RDK.setRunMode(RUNMODE_SIMULATE)
        time.sleep(4) # Espera a que inicie la cámara

    # --------- Distintas aplicaciones para los distintos objetos ---------
    robot.setPoseTool(cam_tool)

    if PLANE_OBJECT: plane_object_aplication()

    elif INSIDE_BOX_OBJECT: inside_box_object_aplication()

    elif OUTSIDE_BOX_OBJECT: outside_box_object_aplication()

    elif ICP_OBJECT: icp_object_aplication()

    # --------- Cierre de las camaras, tanto simulada como real ---------
    if CAMERA_SIMULATION:
        cam_item.setParam('Close', 1)
        #hilofuncionamiento = False # Cierra el thread de funcionamiento de la
cámara es para
```

```
            #hilo_camara.join()  # simular la cámara del ordenador para el
reconocimiento de manos

    else:
        hilofuncionamiento = False # Cierra el thread de funcionamiento de la
cámara
    hilo_camara.join()
```

UNIVERSITAT POLITÉCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un sistema de soldadura robotizado inteligente usando una cámara de profundidad y un robot colaborativo.

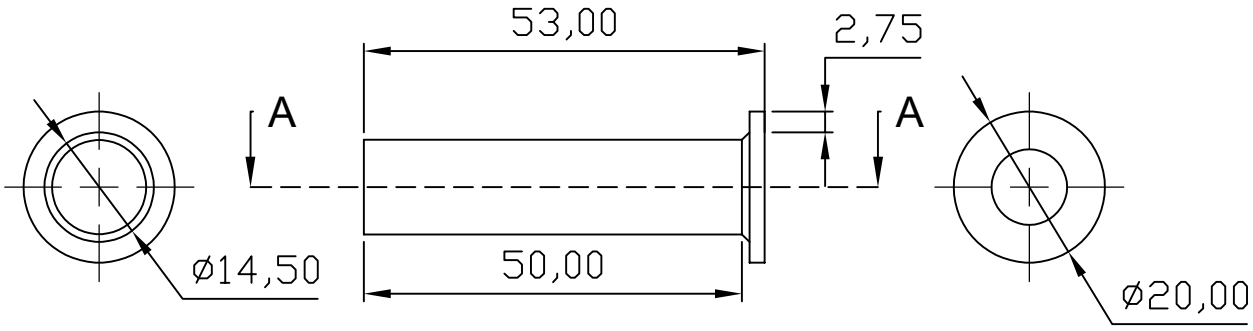
DOCUMENTO Nº2: Planos

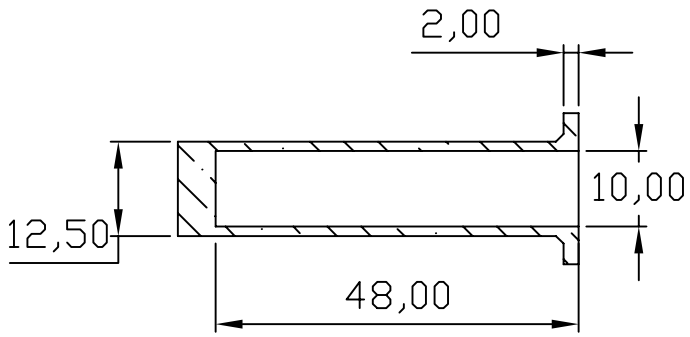
Máster Universitario en Automática e Informática Industrial

AUTOR/A: Górriz Aliaga, Abel

Tutor/a: Ivorra Martínez, Eugenio

CURSO ACADÉMICO:2023-2024





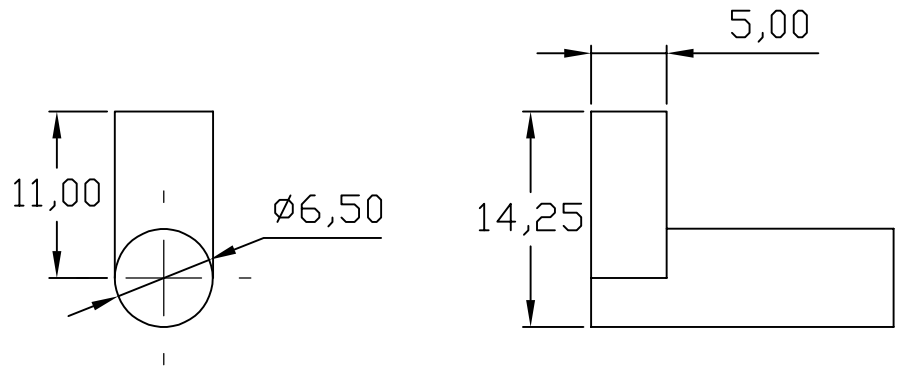
Sección A-A

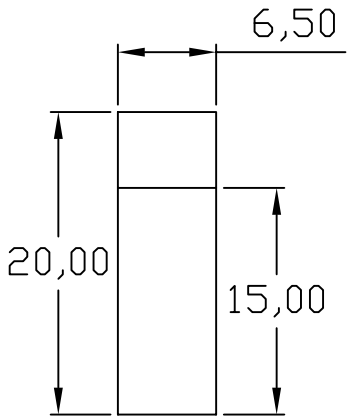
DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo

Autores: ABEL GÓRRIZ ALIAGA

Escala	Denominación del plano	Número del plano
1:1	Acople de la pluma o rotulador	1



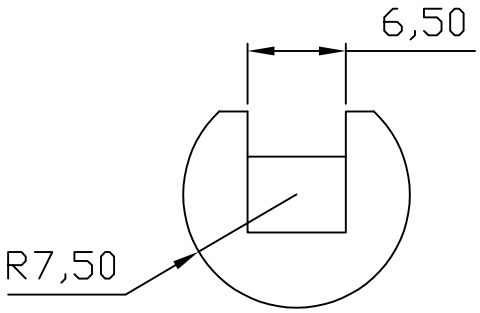


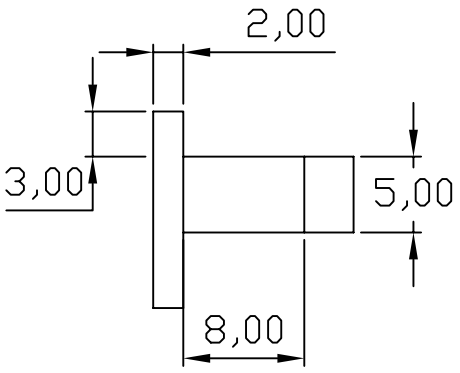
DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

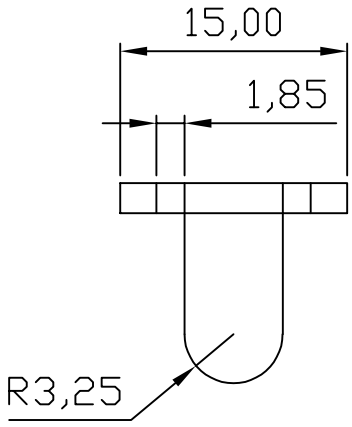
Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo

Autores: ABEL GÓRRIZ ALIAGA

Escala	Denominación del plano	Número del plano
2:1	Pieza nº1 sujeción de la cámara	2





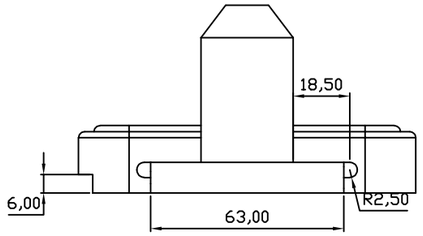


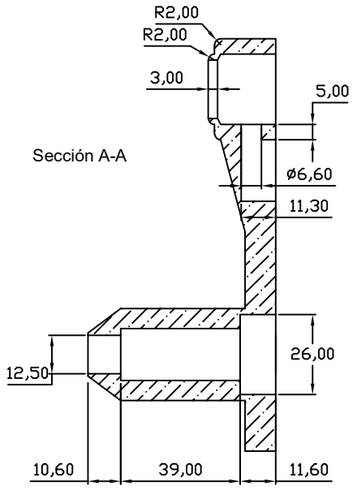
DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

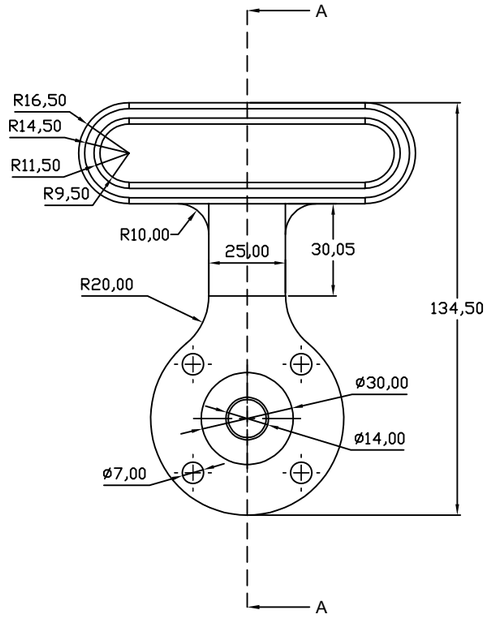
Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo

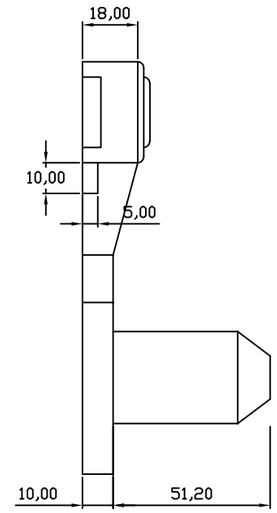
Autores: ABEL GÓRRIZ ALIAGA

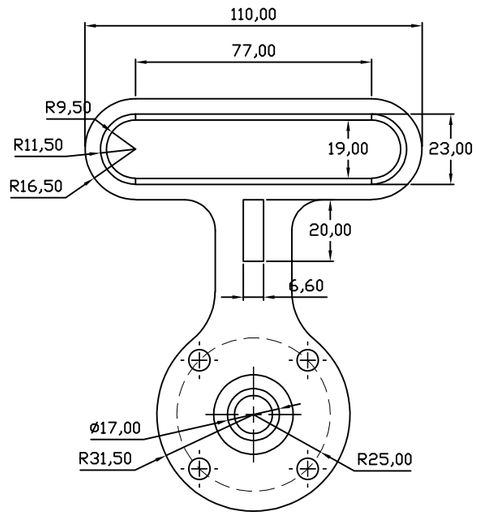
Escala	Denominación del plano	Número del plano
2:1	Pieza nº2 sujeción de la cámara	3

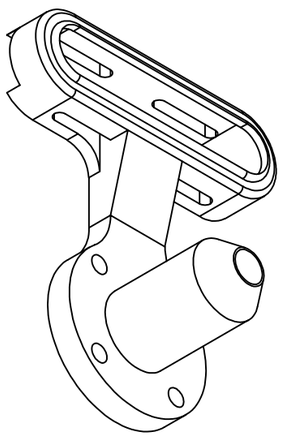


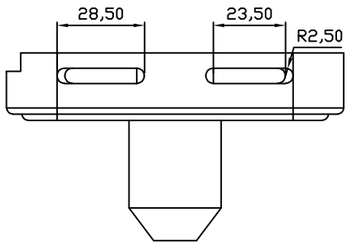










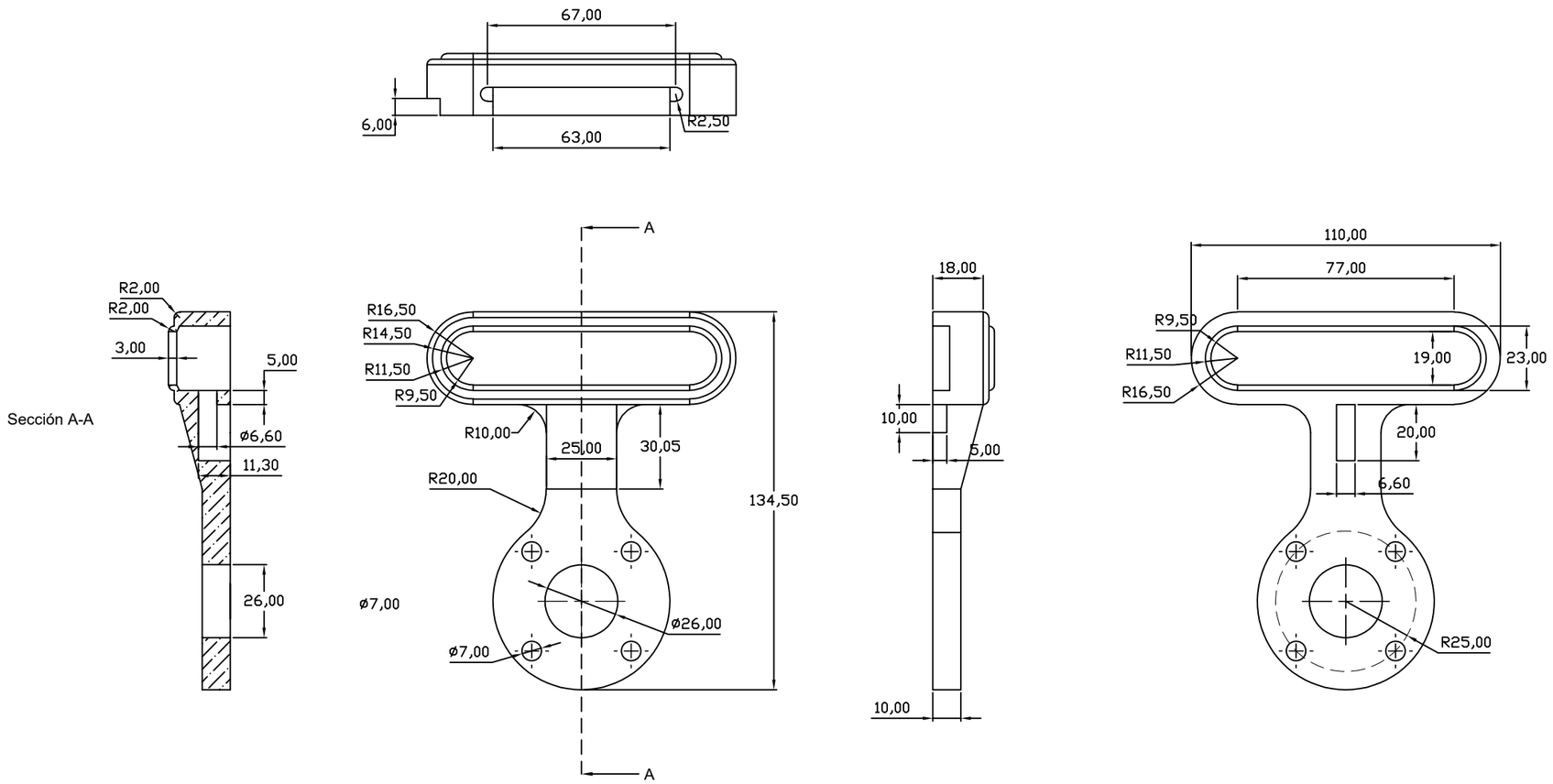


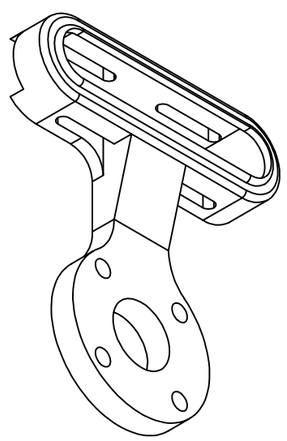
DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo

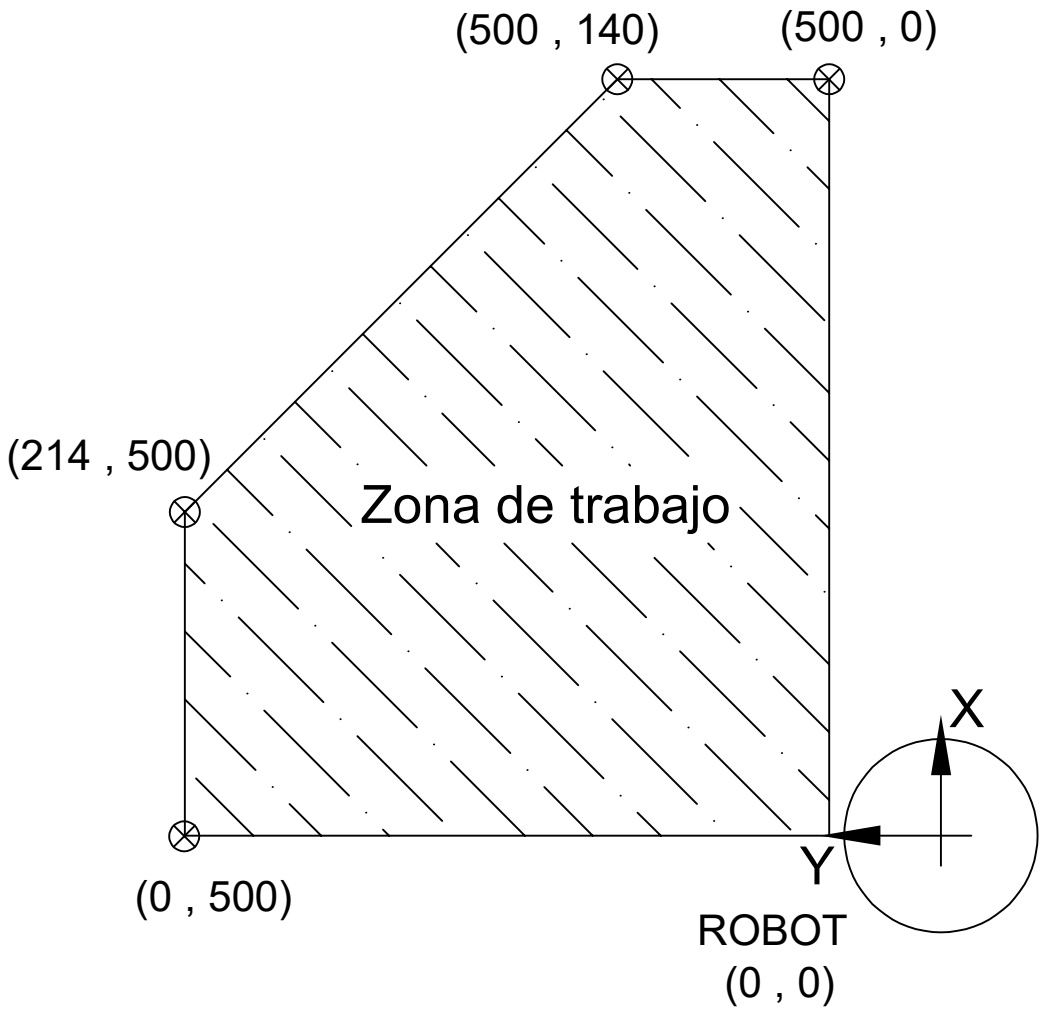
Autores: ABEL GÓRRIZ ALIAGA

Escala	Denominación del plano	Número del plano
1:2	Soporte para cámara y pluma o rotulador	4





DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA		
Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo		
Autores: ABEL GÓRRIZ ALIAGA		
Escala 1:2	Denominación del plano Soporte para cámara	Número del plano 5



DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Titulo del proyecto: Desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo

Autores: ABEL GÓRRIZ ALIAGA

Escala	Denominación del plano	Número del plano
1:5	Plano de situación	6

UNIVERSITAT POLITÉCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un sistema de soldadura robotizado inteligente usando una cámara de profundidad y un robot colaborativo.

DOCUMENTO Nº3: Pliego de Condiciones

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Górriz Aliaga, Abel

Tutor/a: Ivorra Martínez, Eugenio

CURSO ACADÉMICO:2023-2024

Índice de contenidos

1. CONDICIONES GENERALES ____ *141*
1.1. Vigencia ____ 141
1.2. Descripción ____ 141
1.3. Modificaciones ____ 141
1.4. Dirección e inspección ____ 141
2. CONDICIONES FACULTATIVAS ____ *142*
3. OBJETO ____ *142*
4. CONDICIONES DE LOS MATERIALES ____ *142*
4.1. Robot UR3e ____ 142
4.2. Intel RealSense D415 ____ 142
4.3. Quick Changer Tool OnRobot ____ 142
4.4. Diseños 3D ____ 142
4.5. Python ____ 143
4.6. RoboDK ____ 143
5. CONDICIONES DE LA EJECUCIÓN ____ *143*
5.1. Materiales ____ 143
5.2. Ejecución del sistema ____ 144

1. CONDICIONES GENERALES

Este proyecto tiene carácter de obligado cumplimiento una vez sellado y legalizado, debiendo ser objeto de aprobación previa todas aquellas modificaciones al mismo durante su ejecución.

La presente especificación técnica se refiere al desarrollo de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo.

1.1. Vigencia

Este Pliego de Condiciones, con todos sus articulados, estará en vigor durante la ejecución de la instalación y hasta la terminación de esta, entendiéndose que las partes a que hace referencia éste, se aceptarán en todos sus puntos por el adjudicatario de la instalación. Frente a posibles discrepancias, el orden de prioridad de los documentos básicos del Proyecto será el siguiente:

1).-Planos.

2).- Pliego de Condiciones.

3).- Presupuesto.

4).- Memoria.

1.2. Descripción

Este proyecto regula la instalación e implementación de un sistema de soldadura inteligente usando una cámara de profundidad y un robot colaborativo.

1.3. Modificaciones

Durante la ejecución del proyecto, se podrán realizar cuantas modificaciones se estimen oportunas, siempre que las mismas sean aprobadas por el responsable de la Dirección del Proyecto, y en todo momento, de acuerdo con la entidad contratante.

1.4. Dirección e inspección

La dirección de la instalación eléctrica estará a cargo del responsable de la dirección del proyecto, pudiendo éste delegar en personal a cargo de la ejecución práctica de la instalación.

2. CONDICIONES FACULTATIVAS

Las funciones de director de la Instalación son las de revisión del trabajo realizado, programación de los trabajos, reconocimiento de los materiales utilizados y autorizaciones referentes al proyecto

3. OBJETO

La presente especificación técnica se refiere al desarrollo e implementación de un sistema de soldadura usando una cámara de profundidad, Intel Realsense D415, y un robot colaborativo para realizar los movimientos, UR3e del fabricante Universal Robots.

4. CONDICIONES DE LOS MATERIALES

Los materiales deberán ser los finalmente elegidos y encontrarse en buen estado para la correcta instalación y posterior puesta en marcha y uso.

4.1. Robot UR3e

- Alcance: 500 mm
- Carga útil: 3 kg
- Huella: Ø128 mm
- Peso: 11.2 kg

4.2. Intel RealSense D415

- RGB frame resolution: 1920x1080
- RGB frame rate: 30fps
- RGB sensor technology: Rolling Shuther

4.3. Quick Changer Tool OnRobot

- Modelo: QC-R v2
- Carga útil: 20 kg
- Tamaño: Ø63 mm
- Peso total: 0.2 Kg

4.4. Diseños 3D

- Material de la impresión: ácido poliláctico (PLA)

4.5. Python

- Python 3.10.10 (Embedded in RoboDK)

4.6. RoboDK

- RoboDK v5.6.5

5. CONDICIONES DE LA EJECUCIÓN

5.1. Materiales

Robot UR3e:

- Deberá ser colocado en un entorno seguro y en la posición y orientación que marca el plano nº6, en dicho plano se ha tomado como origen la base del robot.
- Se conectará todo el sistema a la electricidad correctamente.

Quick Changer Tool OnRobot:

- Se deberá acoplar la parte fija al extremo final del robot mediante 4 tornillos.
- Se deberá atornillar la parte de la herramienta al soporte del cámara diseñado mediante 4 tornillos

Intel RealSense D415:

- Se colocará dentro del soporte y se fijará mediante las distintas piezas diseñada
- Se conectará mediante un cable USB 3.0 Type-C al ordenador que proceda.
- La lente se deberá mantener limpia en todo momento.

5.2. Ejecución del sistema

Una vez este la instalación terminada se deberá conectar al software de RoboDK y ejecutar el archivo denominado ProgramaSoldadura

Para la ejecución correcta del sistema robotizado sin ningún riesgo, se deberá proceder de la siguiente forma:

1. Se deberá elegir la configuración con la cual se va a proceder, una vez marcadas las opciones pulsar, Guardar Configuración. Estas variables y su efecto se mantendrán contantes para el resto del programa
2. Se deberá colocar el objeto a soldar en el espació delimitado y elegir en la interfaz de Soldadura Robotizada el tipo de aplicación y tipo de simulación. Una vez se este seguro pulsar, Comenzar aplicación.
3. Una vez generadas las trayectorias se puede elegir que acción realizar. Una vez realice la soldadura en el objeto, el robot volverá a su posición inicial y el programa se mantendrá a la espera de recibir una nueva pieza para procesar.
4. Cuando se desee terminar la jornada, se deberá pulsar en la interfaz, Finalizar aplicación.

UNIVERSITAT POLITÉCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un sistema de soldadura robotizado inteligente usando una cámara de profundidad y un robot colaborativo.

DOCUMENTO Nº4: Presupuesto

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Górriz Aliaga, Abel

Tutor/a: Ivorra Martínez, Eugenio

CURSO ACADÉMICO:2023-2024

1. PRESUPUESTO POR COSTES SEGÚN SU NATURALEZA

MATERIALES				
Uds	Denominación	Cantidad	Precio (€)	Total
u	Robot UR3e	1,00	25000,00	25000,00
u	Intel RealSense Depth Camera D415	1,00	447,07	447,07
u	Piezas 3D	1,00	10,00	10,00
u	Ordenador	1,00	600,00	600,00
u	Python	1,00	0,00	0,00
u	Licencia RoboDK	1,00	145,00	145,00
			SUBTOTOTAL MATERIALES	26.202,07 €

MANO DE OBRA				
Uds	Denominación	Cantidad	Precio (€)	Total
h	Desarrollo del sistema por el ingeniero	375,00	15	5625,00
h	Instalacion del material	16,00	15	240,00
			SUBTOTOTAL MANO DE OBRA	5.865,00 €

TOTAL PRESUPUESTO DE EJECUCIÓN MATERIAL	32.067,07 €

Asciende el presente Presupuesto de Ejecución Material a la cantidad de treinta y dos mil sesenta y siete euros con siete céntimos (32.067,07 €)

2. RESUMEN DEL PRESUPUESTO

RESUMEN DEL PRESUPUESTO				
Uds	Denominación	Cantidad	Precio (€)	Total
u	Total presupuesto de ejecución material	1,0	32067,07	32067,07
%	Gastos Generales	0,13	32067,07	4168,7191
%	Beneficio Industrial	0,06	32067,07	1924,0242
%	Honorarios	0,1	32067,07	3206,707
%	IVA	0,21	32067,07	6734,0847
			TOTAL PROYECTO	48.100,61 €

Asciende el presente Presupuesto Total del Proyecto a la cantidad cuarenta y ocho mil cien euros con sesenta y un céntimos (48.100,61 €)