UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Identificación de piezas en problemas de bin-picking sobre NVIDIA ORIN NX

Trabajo Fin de Máster

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Pérez Sánchez, Víctor Antoni

Tutor/a: Blanes Noguera, Juan Francisco

CURSO ACADÉMICO: 2022/2023

Resumen

El *bin picking* es un proceso comúnmente utilizado en cadenas de producción industrial. Esto consiste en, con un brazo robótico, coger objetos de un contenedor y colocarlos en otro lugar. Estos sistemas pueden ser manuales o estar automatizados mediante distintas maneras.

En este proyecto se ha desarrollado un sistema inteligente de *bin picking* con la intención de poder realizar el proceso automáticamente y con la mayor eficacia posible. Para ello se ha empleado una red neuronal para identificar las piezas que se quieren coger. Posteriormente se utiliza la información obtenida por la red neuronal para realizar una tarea de pick and place con estas piezas usando un brazo robótico industrial.

El dispositivo encargado de procesar las imágenes y la red neuronal es una NVIDIA Jetson AGX Orin, un sistema empotrado altamente eficiente diseñado específicamente para tareas relacionadas con la visión por computador.

El resto del sistema está compuesto por una cámara Intel RealSense Depth Camera D415, una cámara especializada en tareas de visión artificial que permite saber a qué distancia se encuentran los objetos. Esta cámara está situada cerca del extremo de un Sawyer Research Robot, un robot colaborativo de Rethink Robotics que se encargará de coger y mover las piezas.

Palabras clave: Visión por computador, Deep learning, Robótica, Pick and place

Abstract

Bin picking is a process commonly used in industrial production chains. It consists in using a robotic arm to pick up objects from a container and place them somewhere else. These systems can be manual or automated in various ways.

In this project, an intelligent bin picking system has been developed with the intent of carrying out the process automatically and with the maximum possible efficiency. To achieve this, a neural network has been employed to identify the items to be picked. Subsequently, the information obtained from the neural network is used to perform a pick and place task with these items using an industrial robotic arm.

The device responsible for processing the images and the neural network is an NVIDIA Jetson AGX Orin, a highly efficient embedded system specifically designed for computer vision-related tasks.

The rest of the system consists of an Intel RealSense Depth Camera D415, a specialized camera for computer vision tasks that allows determining the distance to objects. This camera is positioned near the end of a Sawyer Research Robot, a collaborative robot from Rethink Robotics responsible for picking up and moving the components.

Key words: Computer vision, Deep learning, Robotics, Pick and place

Índice

1. Introducción

En este primer apartado se va a comentar qué motivos han impulsado a realizar este trabajo, así como los objetivos que se pretenden lograr.

1.1 Motivación

El desarrollo de este proyecto viene impulsado a nivel tecnológico por la búsqueda de una solución robusta al problema de la detección de objetos para tareas de *bin picking*. Este tipo de procesos son una práctica relativamente reciente por lo que puede resultar interesante encontrar una solución más moderna para el problema.

Por otro lado, a nivel personal la elección de este trabajo viene impulsada por la intriga generada por los avances en la inteligencia artificial. Día a día se ven más ejemplos de usos de inteligencia artificial en distintos ámbitos de nuestra vida cotidiana, ya sea en asistentes inteligentes en nuestros móviles, sistemas de conducción automática o semiautomática en nuestros coches o en distintas partes de nuestras casas, como puede ser un robot aspirador o un sistema de iluminación inteligente.

Otro punto interesante de este trabajo es el poder trabajar de primera mano con un robot industrial moderno. La robótica es un campo que siempre ha despertado mi interés. Desde pequeño he jugado con Legos, coches teledirigidos y otros tipos de juguetes móviles o articulados, por lo que realizar este TFM involucrando un robot programable de alta gama era una oportunidad que no podía dejar pasar.

El desarrollo de este proyecto me permitiría trabajar, a pequeña escala, con una inteligencia artificial y con un brazo robótico moderno, lo cual me permitirá aprender más tanto sobre su funcionamiento como por el nivel de evolución que pueden alcanzar ambas ramas en un futuro cercano.

1.2 Objetivos

El objetivo de este proyecto es desarrollar un sistema de *bin picking* robusto. Las características que se desea que este sistema presente son tasas muy pequeñas de falsos positivos y falsos negativos en las detecciones de los objetos, movimientos precisos del robot y una alta eficiencia a la hora de realizar la tarea repetidas veces. Para conseguirlo se deben cumplir dos subojetivos.

1.2.1 Despliegue de una red neuronal en un sistema empotrado

En primer lugar se debe implementar una red neuronal entrenada para detección de objetos en una NVIDIA Jetson AGX Orin, una placa de desarrollo especializada en tareas de visión artificial. Los objetos que la red neuronal detectará son los cilindros de color gris con los que se ha entrenado la red (figura 1). Para realizar esta tarea se ha hecho uso de Python como lenguaje de programación principal. Concretamente se han utilizado distintas librerías relacionadas con la visión por computador y el procesamiento de imagen, como pueden ser *OpenCV*[1] o *tensorflow*[2].

Figura 1. Cilindros con los que se ha entrenado la red

Cabe destacar que el entrenamiento de dicha red neuronal no está incluido dentro del desarrollo de este proyecto. Simplemente se partió de una red neuronal previamente entrada y optimizada y se trabajó a partir de ella. Se recomienda recurrir a la bibliografía[5] de este documento para más información al respecto.

1.2.2 Control del robot

La otra parte del proyecto consiste en implementar el sistema de visión artificial de la Jetson en un sistema físico compuesto por un brazo robótico (figura 2) y una cámara. Se pretende utilizar la información obtenida con la red neuronal para crear un sistema automático e inteligente que realice una tarea de pick and place de manera indefinida con una serie de piezas.

1.3 Estructura de la memoria

Esta memoria documenta el trabajo realizado durante este proyecto. Está dividida en los siguientes capítulos:

- Capítulo 2. Estado del arte. En este capítulo se introducen conceptos importantes para el desarrollo de este proyecto y se expone el estado del arte actual de estas disciplinas.
- Capítulo 3. Integración y aplicación de la red neuronal. Este capítulo enumera y detalla todas las necesidades que han debido de satisfacerse para poder hacer uso de la red neuronal de la que parte en este proyecto. También se incluyen qué pasos se han seguido para poder obtener las detecciones de la red y cómo obtener las características representativas de los objetos detectados.
- Capítulo 4. Control del brazo robótico. En este capítulo se explica cómo controlar el brazo robótico que se ha usado en el proyecto. Esto incluye tanto la comunicación con el robot como el control del movimiento del brazo.
- Capítulo 5. Pruebas de validación. Este apartado expone las pruebas realizadas para verificar el correcto funcionamiento del sistema. También se incluye una comparativa temporal del sistema de detección de objetos en un ordenador común y en la placa de desarrollo especializada utilizada en el proyecto.
- Capítulo 6. Resultados y conclusiones. En este apartado se evalúan los resultados obtenidos y se plantean posibles trabajos futuros

2. Estado del arte

En este apartado se van a introducir distintos conceptos importantes involucrados en el desarrollo de este proyecto. Se hará una explicación fundamental de los conceptos seguido de la exposición del estado del arte de dichos campos.

2.1 Inteligencia artificial y redes neuronales

La inteligencia artificial es un campo de la informática que se centra en crear sistemas y programas capaces de imitar la inteligencia humana para realizar distintas tareas y son capaces de mejorar conforme recopilan más información. Estas tareas incluyen el aprendizaje, la percepción, el razonamiento y la toma de decisiones.

El *machine learning* (aprendizaje automático) es una subdisciplina de la inteligencia artificial que se enfoca en desarrollar algoritmos y modelos que permiten a los ordenadores aprender de datos y mejorar su rendimiento en una tarea específica sin ser programados explícitamente para ella.

Las redes neuronales son un componente fundamental del *machine learning*. Son modelos matemáticos inspirados en la estructura y funcionamiento del cerebro humano. Se utilizan para procesar datos de entrada, aprender patrones y realizar tareas específicas sin necesidad de programación explícita.

En otras palabras, las redes neuronales son una tecnología que permite a las máquinas realizar tareas de aprendizaje automático. Se componen de capas de neuronas artificiales interconectadas que procesan datos y ajustan sus conexiones a través del entrenamiento con datos de entrada. Esto permite que las redes neuronales aprendan y generalicen a partir de ejemplos, lo que las convierte en una poderosa herramienta para tareas de clasificación y reconocimiento de patrones entre otras cosas.

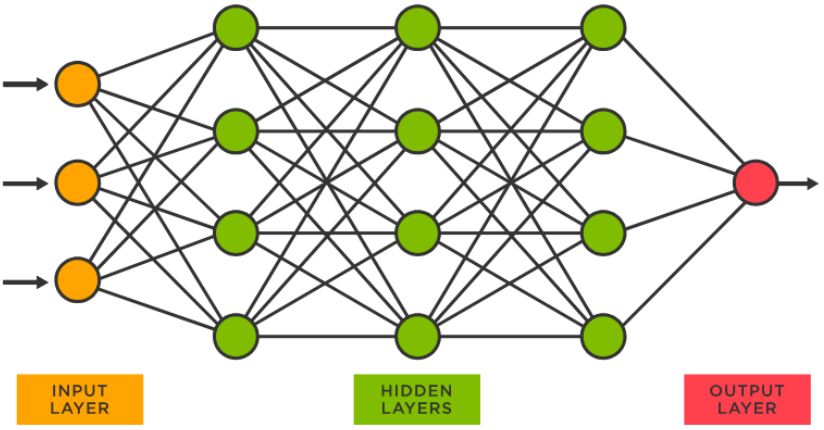

Figura 2. Esquema representativo de una red neuronal

2.2 Sistemas embebidos

Los sistemas embebidos son la forma más sencilla de aplicar una solución a un problema que requiera capacidad computacional. En los últimos años se ha avanzado mucho en el desarrollo de estos ordenadores de bolsillo, especialmente para su aplicación en el campo de la visión artificial.

Los sistemas embebidos son componentes clave en aplicaciones de visión artificial, como pueden ser vehículos autónomos, cámaras de seguridad o equipamiento sanitario. Para una mayor efectividad, una de las tendencias más notables es el aumento en la potencia de procesamiento de estos sistemas, gracias a la integración de GPUs de alto rendimiento. Esto ha permitido convertir tareas de visión artificial complejas, como el reconocimiento de objetos en tiempo real, en tareas rápidas y sencillas.

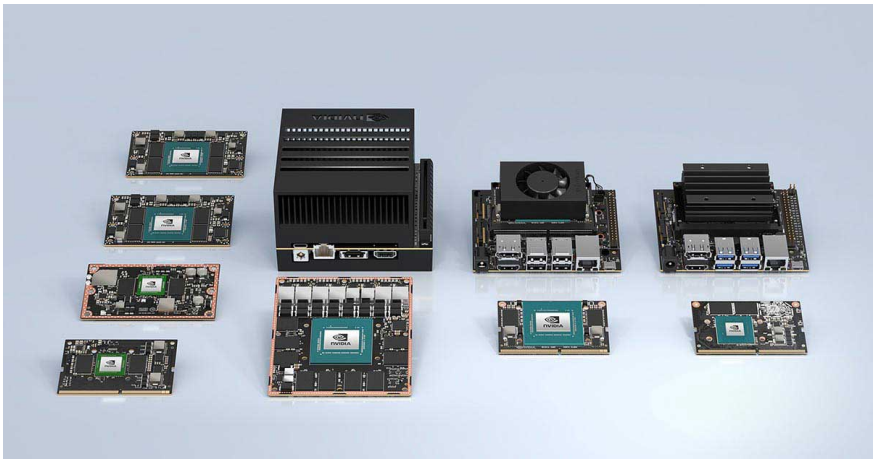

Figura 3. Distintos ejemplos de sistemas embebidos

Otro avance importante es la mejora en la eficiencia energética de los sistemas embebidos. Los diseños de bajo consumo de energía son esenciales para aplicaciones alimentadas por batería, como drones, robots móviles y cámaras de vigilancia inalámbricas.

No solo ha mejorado la capacidad de procesamiento de los sistemas embebidos, sino que además se han estado desarrollando sistemas embebidos especializados en visión artificial e inteligencia artificial. La finalidad del desarrollo de estos es proporcionar una alternativa más rápida y más eficiente para realizar el entrenamiento de modelos de *machine learning*.

2.3 Cámaras de visión artificial

Un sistema de visión no puede funcionar sin unos ojos. Es por ello que la cámara de un sistema de visión artificial es una parte fundamental del mismo, por lo que cuanta mayor calidad de imagen proporcione y mejor capacidad de procesamiento tenga mejor será el rendimiento del sistema.

Las cámaras para aplicaciones de visión por computador se han vuelto cada vez más pequeñas, eficientes y con mayor de capacidad de procesamiento. La integración de sensores avanzados, como cámaras de profundidad, sensores LiDAR y cámaras multiespectrales, permiten una percepción más precisa y detallada del entorno, lo que es esencial para aplicaciones en vehículos autónomos, robótica y realidad aumentada.

Una tendencia reciente en el desarrollo de aplicaciones en entornos en los que las tres dimensiones son relevantes es el uso de cámaras de profundidad (depth cameras). Estas cámaras capturan información tridimensional, permitiendo una percepción precisa de la distancia a la que se encuentran los objetos. Esto es posible gracias al uso de tecnologías como la estereovisión, lo cual ha mejorado la precisión y la velocidad de procesamiento de las cámaras de profundidad.

La estereovisión es una técnica de percepción en visión por computadora que se basa en el uso de dos cámaras o sensores ópticos para capturar imágenes desde dos puntos de vista ligeramente diferentes, simulando la visión binocular humana. Al comparar las diferencias entre las imágenes capturadas, se puede calcular la distancia y la profundidad de los objetos en una escena tridimensional.

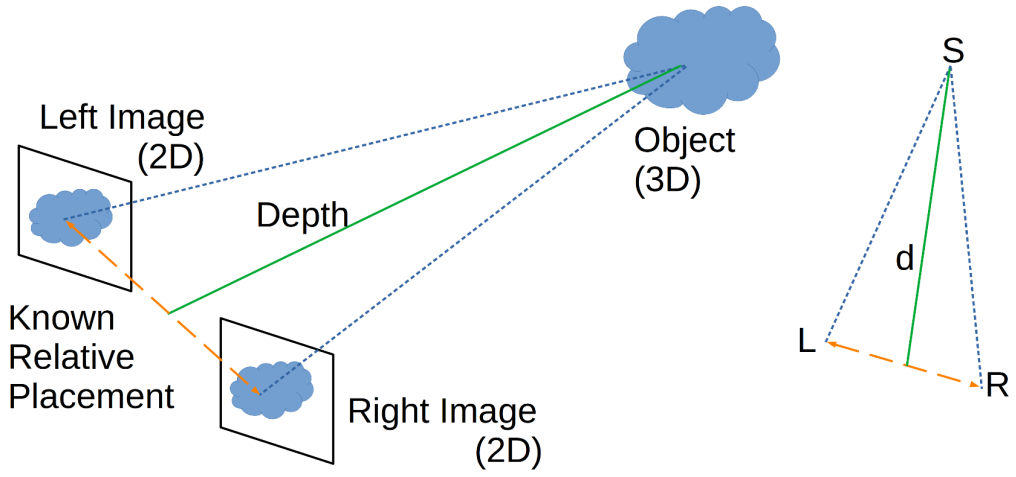

Figura 4. Visualización del cálculo de la profundidad mediante estereovisión

2.4 Manipuladores robóticos

Los robots manipuladores son brazos robóticos diseñados para realizar tareas de manipulación en entornos industriales y no industriales. Por motivos de seguridad deben tener su propio espacio de trabajo aislado de los humanos.

Por otro lado, los conocidos como robots colaborativos están diseñados para poder trabajar conjuntamente con seres humanos sin necesidad de jaulas de seguridad. Esto ha impulsado la adopción de robots en entornos de manufactura y logística, donde pueden ayudar a aumentar la eficiencia y mejorar las condiciones laborales.

Una de las tareas más comunes para las que se emplea este tipo de robots es el *pick and place*. Esta tarea consiste en, como su nombre indica, coger un objeto y depositarlo en otro lugar. A continuación se hablará brevemente sobre este proceso

2.4.1 Tareas inteligentes de pick and place

Para poder realizar una tarea de pick and place es necesario conocer dos cosas. Dónde está el objeto que se desea coger y donde se quiere depositar. Hay distintos métodos de obtener esta información pero el más extendido por la industria es usar los sistemas CAT90-SA[6].

Los sistemas CAT90-SA pick and place son máquinas semiautomáticas con orientación de software y la última tecnología de visión de alta resolución. Estos sistemas funcionan detectando e identificando los componentes que se deben recoger y colocar. Esto se logra mediante cámaras de alta resolución y algoritmos de visión por computador. El software identifica los componentes basándose en sus características, como forma, tamaño y guiándose por un objeto de referencia.

Este método, pese a ser muy eficiente, tiene el inconveniente de que, al usar una referencia, el sistema es propenso a tener ligeros problemas cuando hay ligeras variaciones entre los objetos.

Este proyecto pretende mejorar esta solución, empleando una detección de piezas mediante una red neuronal, lo que permitirá adaptarse mejor a variaciones en los objetos y permitirá obtener un sistema más robusto.

3. Integración y aplicación de la red neuronal

La red neuronal de la que se parte para el desarrollo de este proyecto se ha entrenado utilizando las librerías de tensorflow. TensorFlow es una biblioteca de código abierto desarrollada por Google para aprendizaje automático e inteligencia artificial. Puede ser utilizada en una variedad de tareas, pero se enfoca especialmente en el entrenamiento y la inferencia de redes neuronales.

La red neuronal utilizada en el proyecto es una red neuronal U-net. U-net es un tipo de arquitectura de red neuronal utilizada principalmente en aplicaciones de segmentación de imágenes. Este tipo de arquitectura se utiliza generalmente en en tareas de segmentación semántica, donde se deben identificar objetos individuales en una imagen.

Los elementos más importantes en la arquitectura de una red neuronal son un codificador y un decodificador. En el codificador, también conocido como downsampling, a partir de la imagen de entrada se generará un mapa de características de baja resolución. Tras esto, en el decodificador, también conocido como upsampling, se aumentará la dimensión espacial del mapa de características anterior con el fin de obtener una imagen del mismo tamaño que la imagen de entrada. Finalmente, la salida final del decodificador se alimenta a un clasificador multiclase, produciendo así las probabilidades de clase para cada píxel de forma independiente y extrayendo la segmentación.

U-net es una arquitectura simétrica ya que cuenta con las mismas capas de codificador y decodificador. Por ello, tiene una conexión por cada una de las capas de codificador o decodificador. Esto conlleva a que la red tenga un mayor número de mapas de características en la fase de decodificación, lo que permite transferir más información.

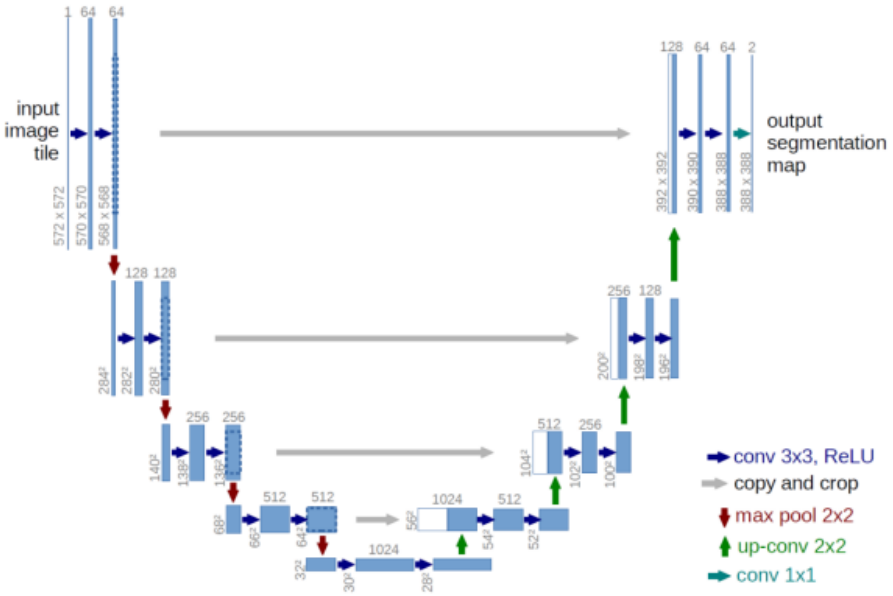

Figura 5. Esquema de la estructura de una red neuronal U-net

El proceso de entrenamiento de la red, al no ser parte de este proyecto, no se detalla en este documento, y solo se explicará cómo implementarla dentro del sistema que se ha desarrollado. Se recomienda recurrir a la bibliografía[5] de este documento para más información del entrenamiento de la red.

Antes de detallar cómo hacer uso de la red neuronal, es necesario comentar algunas características y el funcionamiento de la cámara y la placa de desarrollo que se van a utilizar.

3.1 NVIDIA Jetson AGX Orin

El dispositivo encargado de gestionar la red neuronal es una NVIDIA Jetson AGX Orin[9] (figura 6), una placa de desarrollo de NVIDIA diseñada específicamente para aplicaciones de *Deep learning* y de inteligencia artificial. Es parte de la serie de productos Jetson de NVIDIA, que incluye sistemas en módulo y tarjetas de desarrollo que están optimizados para tareas de procesamiento de datos y visión por computadora en tiempo real en entornos de bajo consumo de energía.

Figura 6. NVIDIA Jetson AGX Orin

Las Jetson Orin se basan en la arquitectura NVIDIA Ampere y ofrecen un alto rendimiento computacional gracias a su GPU integrada y unidades de procesamiento tensorial. Está diseñada para aplicaciones de alto rendimiento, como vehículos autónomos, robots, sistemas de visión por computador avanzados, y otros dispositivos inteligentes que requieren procesamiento de datos en tiempo real.

Esta plataforma presenta su propio software de desarrollo, que incluye el kit de desarrollo de software (SDK) de JetPack de NVIDIA, que facilita el desarrollo de aplicaciones de inteligencia artificial y *machine learning* en la propia Jetson Orin. Además, es capaz de adaptarse a diversas necesidades de rendimiento, lo que la hace versátil para una amplia gama de aplicaciones de inteligencia artificial.

3.2 Intel RealSense Depth Camera D415

3.2.1 Características

La cámara utilizada en el proyecto es la RealSense Depth Camera D415 de Intel[8] (figura 7). Esta cámara ofrece una alta calidad en la detección de la profundidad además de una buena calidad de imagen. Además, sus reducidas dimensiones y ligero peso hacen que sea una muy buena solución para problemas relacionados con la robótica móvil.

Figura 7. Intel RealSense Depth Camera D415

La RealSense D415 tiene un campo de visión estándar, el cual permite obtener una mayor resolución de imagen y precisión en la profundidad cuando se requieren medidas más precisas. Los sensores y las lentes pequeñas que presenta esta cámara permiten una cámara de profundidad de menor coste pero altamente capaz.

Para la obtención de la profundidad esta cámara utiliza sensores de obturador (global shutter) junto a estereovisión mediante sus dos lentes. Los sensores de obturador son una tecnología que permite detectar a qué distancia se encuentran los objetos de la escena. Funcionan de la siguiente manera:

Las cámaras emiten un pulso de luz láser hacia la escena y se activa un obturador en la cámara. El obturador bloquea la entrada de luz hasta que el pulso de luz láser regresa después de rebotar en los objetos de la escena. Al medir el tiempo que tarda la luz en regresar después de que se abre el obturador, la cámara puede calcular la distancia entre la cámara y los objetos en la escena, creando así un mapa de profundidad en tiempo real.

Además del obturador, la cámara presenta dos lentes (figura 8), lo cual permite obtener la distancia de los objetos de la escena por estereovisión, lo cual resulta en una mayor precisión en el cálculo de estas distancias.

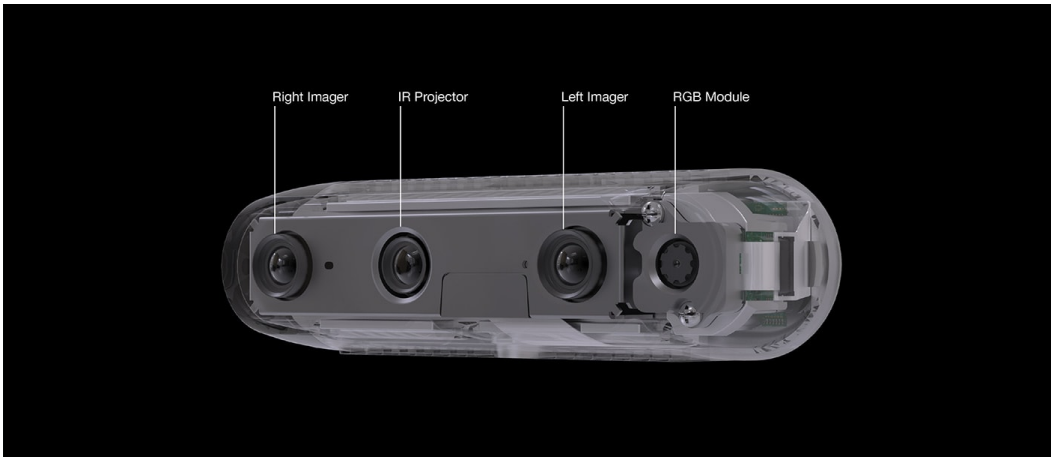

Figura 8. Distribución de las lentes en la D415

Por último, las cámaras RealSense de Intel ofrecen, mediante la librería de python *pyrealsense2*, un software personalizable, el cual permite utilizar las cámaras de la manera adecuada para el sistema en el que se vayan a implementar.

3.2.2 Utilización

Para usar la cámara se ha creado en un script una clase con una función de inicialización de la cámara (*__init__)* y una función para obtener las imágenes que detecte la cámara (*get_frame())*.

La función *__init__* es llamada al principio del código principal del sistema y se encarga de realizar todas las acciones necesarias para poner en funcionamiento la cámara. Esto incluye:

- Configurar los streaming de los canales RGB y Depth, especificando el tamaño deseado de imagen (640x480) y la tasa de fotogramas (30).
- Configurar el filtro de exposición (*exposure*) para ajustar la luminosidad de las imágenes. Este valor depende de la luminosidad del entorno en el que se encuentre el sistema. En las pruebas realizadas se ha buscado un valor que permite visualizar correctamente las imágenes obtenidas. Este valor ha resultado ser 150.
- Mediante la función *align*, sincronizar los streaming de los canales RGB y Depth para que, dado un instante concreto, se pueda obtener los frames de ambos canales de ese preciso instante.

La otra función utilizada (*get_frame()*) se llama cuando se quiera obtener un fotograma de lo que esté capturando la cámara. El funcionamiento de la función es el siguiente:

- La función obtiene los frames de ambos streaming (RGB y depth) alineados.
- Convierte los frames en arrays para poder usarlos como imágenes.
- Por último, devuelve las imágenes para poder usarlas en el programa principal.

Con el funcionamiento de la cámara definido, a continuación se va a explicar como se ha trabajado con la red neuronal.

3.3 Red neuronal de tensorflow

3.3.1 Características de la red

El primer paso para poder utilizar una red neuronal entrenada con tensorflow es saber que tipo de variable espera la red a la entrada y que tipo de variable devuelve en la salida. Para averiguarlo simplemente se debe cargar el modelo en una variable con la función *“tf.keras.models.load_model()”* y podremos ver toda la información sobre sus capas usando el atributo *“model.summary”.* En concreto, la primera y la última capa de la red representan la entrada y la salida respectivamente.

La red que se ha utilizado para el proyecto espera a la entrada una variable con la forma *(None, 480, 640, 4)* y devuelve a la salida una variable con la forma *(None, 480, 640, 1)*. Sabiendo que la red trabaja con imágenes, se deduce fácilmente el significado de estos valores.

En primer lugar, el valor *None* indica que la red espera recibir un número indeterminado de imágenes. Los valores 480 y 640 son las dimensiones en píxeles (ancho y largo) de las imágenes. Por último, el último valor indica el número de canales de la imagen. Un 1 indica que la imagen solo tiene un canal (monocromo), mientras que un 4 indica que la imagen tiene cuatro canales, los canales RGB y un cuarto canal D que representa la profundidad.

Con esta información ya nos podemos disponer a inferir con la red neuronal sobre el modelo.

3.3.2 Inferencia

En el ámbito del *machine learning,* la inferencia es el proceso en el cual se introducen una serie de datos en un modelo y se obtiene una salida. En nuestro caso, vamos a introducir las imágenes detectadas por la cámara en nuestro modelo y obtendremos otras imágenes con la información de qué piezas se han detectado.

En este proyecto, como se ha comentado previamente, la inferencia será realizada por la NVIDIA Jetson Orin. Para ello, es necesario tener el archivo del modelo en el sistema empotrado. Esto se puede realizar, por ejemplo, trasladando el archivo mediante un USB.

Una vez tengamos el modelo en la Jetson Orin, utilizaremos un script de python para realizar el procesado de las imágenes y la inferencia. El primer paso es obtener las imágenes de entrada. Para ello usaremos la función *get_frame()* definida previamente. Ésta nos proporciona dos imágenes separadas: una de 3 canales RGB (*rgb* en la figura 9*)* y una de un canal D (*depth* en la figura 9). La red espera una sola imagen de 4

canales así que debemos combinarlas. Otro requerimiento es que los valores de los píxeles estén normalizados entre 0 y 1, así que realizaremos ese cambio aquí también. Este proceso se puede ver en la figura 9.

```
rgbd = np.zeros((rgb.shape[0], rgb.shape[1], 4))
rgbd[:, :, 0] = rgb[:, :, 0]
rgbd[:, :, 1] = rgb[:, :, 1]
rgbd[:, :, 2] = rgb[:, :, 2]
rgbd[:, :, 3] = depth[:, :]
rgbd = rgbd / 255.0
rgbd = np.expand_dims(rgbd, axis=0)
```

Figura 9. Tratamiento de las imágenes de entrada

Con la imagen combinada (*rgbd*) podemos realizar la inferencia simplemente con *model.predict(rgbd).* En las figuras 10 y 11 se muestra una imagen de ejemplo y el resultado que se obtiene al realizar la inferencia.

Figura 10. Imagen RGB introducida a la entrada de la red neuronal

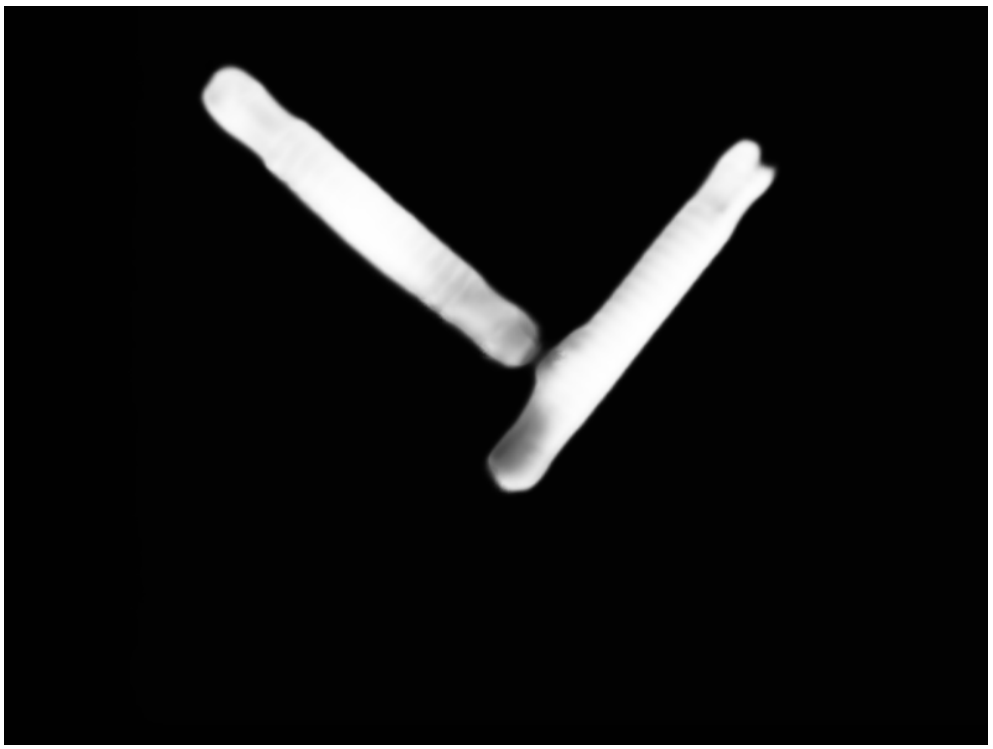

Figura 11. Imagen de salida de la red neuronal

El modelo detecta los dos cilindros que no se encuentran tapados por ningún otro, justo como se espera. Sin embargo, debemos pulir la imagen para que luego sea posible obtener su centroide para poder agarrar las piezas con el robot.

Para obtener una imagen más nítida (figura 12) simplemente realizaremos un sencillo proceso de umbralización, es decir, a todos los píxeles con valor superior a 0.5 se les asignará un valor de 1, y a los píxeles con valor inferior a 0.5 se les asignará un valor de 0.

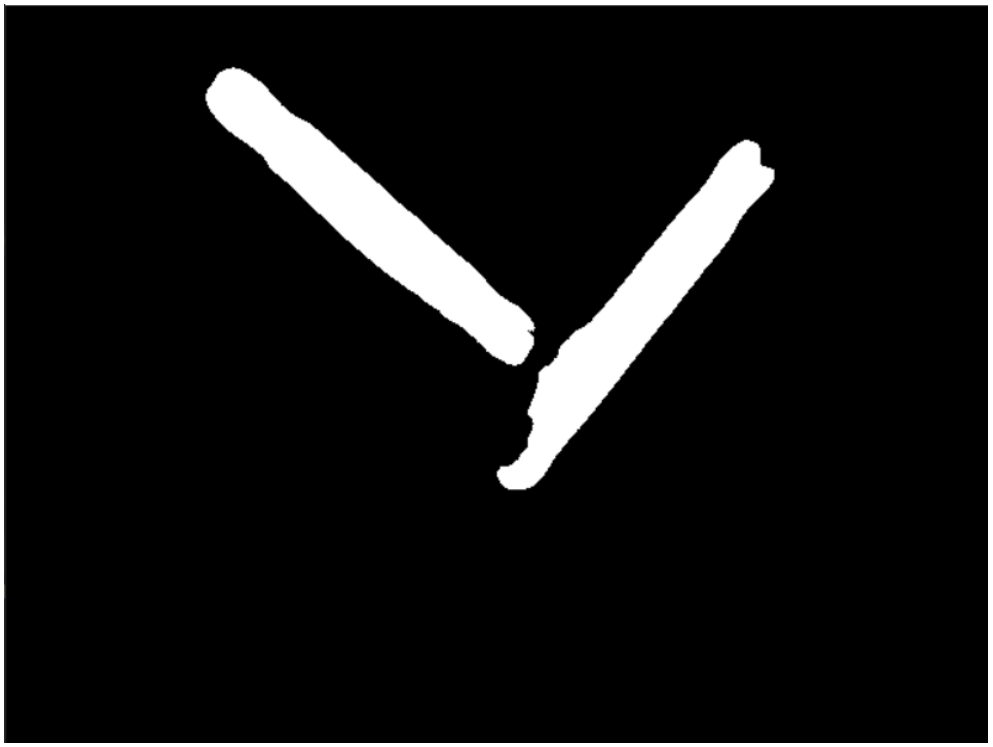

Figura 12. Imagen de salida tras umbralizar

Con las piezas detectadas correctamente, el último paso antes de poder utilizar el robot es obtener el centroide y la orientación de las piezas.

3.4 Obtención de las características

La finalidad de obtener el centroide y la orientación de las piezas es poder indicar al robot exactamente donde tiene que moverse para agarrar los cilindros. Para conseguirlo se ha creado una función haciendo uso de funciones de la librería de OpenCV para calcular las características mediante métodos geométricos. El siguiente flowchart describe de manera resumida el funcionamiento de esta función

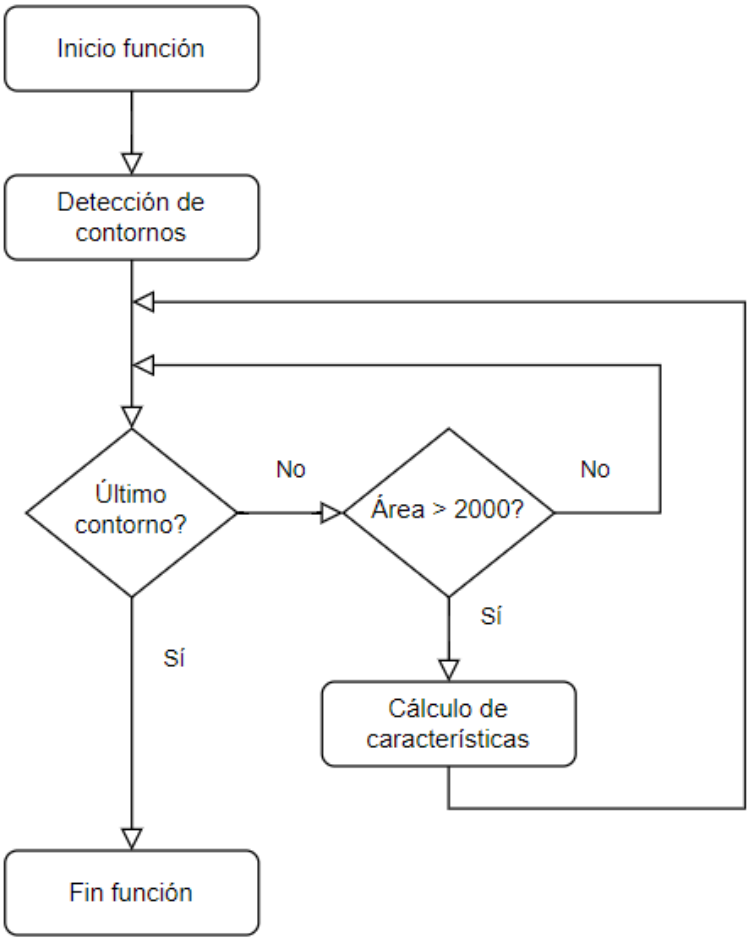

Figura 13. Flowchart de la función de obtención de características

3.4.1 Obtención del centroide

El primer paso es realizar la preparación de la imagen mediante la conversión a escala de grises. Además, se aplica un filtrado gaussiano mediante la función *cv2.GaussianBlur* y se ha realizado una segunda umbralización.

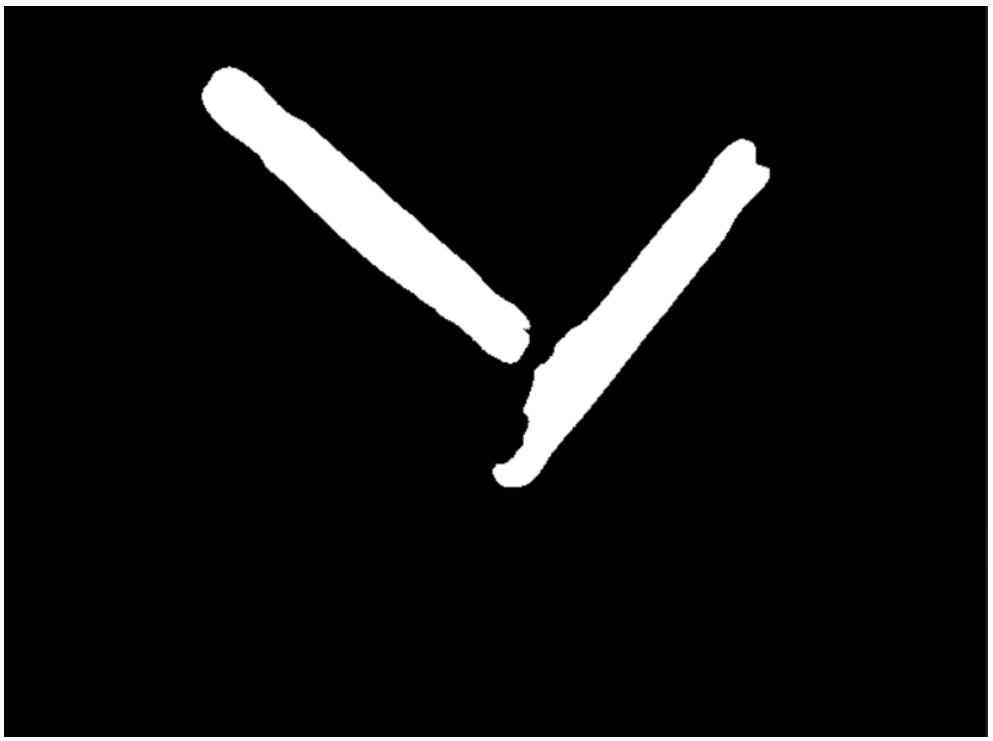

Figura 14. Detección de las piezas tras el filtrado

El siguiente paso es usar la función cv2.findContours para detectar los contornos en la imagen procesada, generando una lista de contornos identificados. A continuación se realizan una serie de cálculos a todos los elementos de la lista.

En primer lugar se calcula el área de los contornos y se someten a un triaje en el que se discriminan objetos muy pequeños, ya que estos representan ruido en las detecciones. A continuación se calculan los momentos de los contornos mediante la función *cv2.moments*. Los momentos son valores que representan las intensidades de los píxeles de una imagen o, en nuestro caso, de los contornos. Con ellos se pueden calcular distintas propiedades de una imagen, como el centroide o el área.

Nosotros vamos a utilizar los momentos de orden 0 y 1 (m_{00}, m_{01} y m_{10}) para obtener los valores de las coordenadas del centroide del objeto usando las siguientes fórmulas:

$$Cx = \frac{m10}{m00}$$
$$Cy = \frac{m01}{m00}$$

En la siguiente figura se pueden observar los centroides detectados en las piezas.

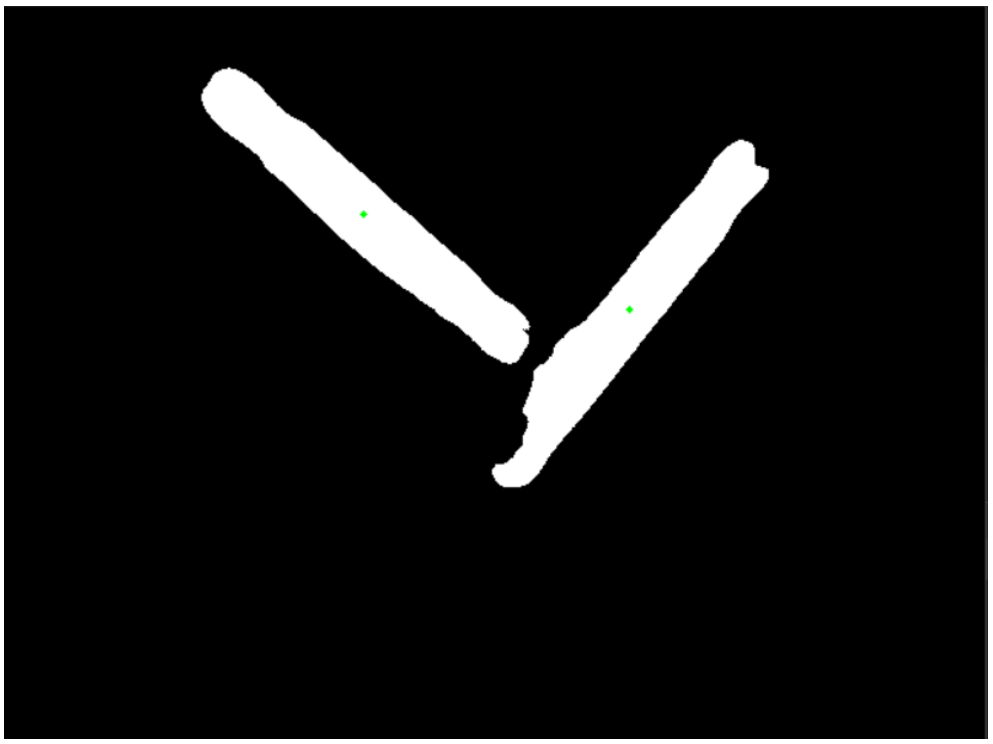

Figura 15. Centroides de los objetos detectados

3.4.2 Obtención de la orientación

El último parámetro que necesitamos obtener es el ángulo de rotación del objeto respecto al marco de la imagen, para poder orientar la garra del robot correctamente para agarrar el cilindro.

Para hallar el ángulo vamos a recurrir a los vectores unitarios del eje de la pieza y de la perpendicular a la base de la imagen. Estos los podemos calcular con la función *np.linalg.norm()* de la librería Numpy. Al calcular el arcoseno de estos dos valores obtenemos el ángulo que forman las dos líneas.

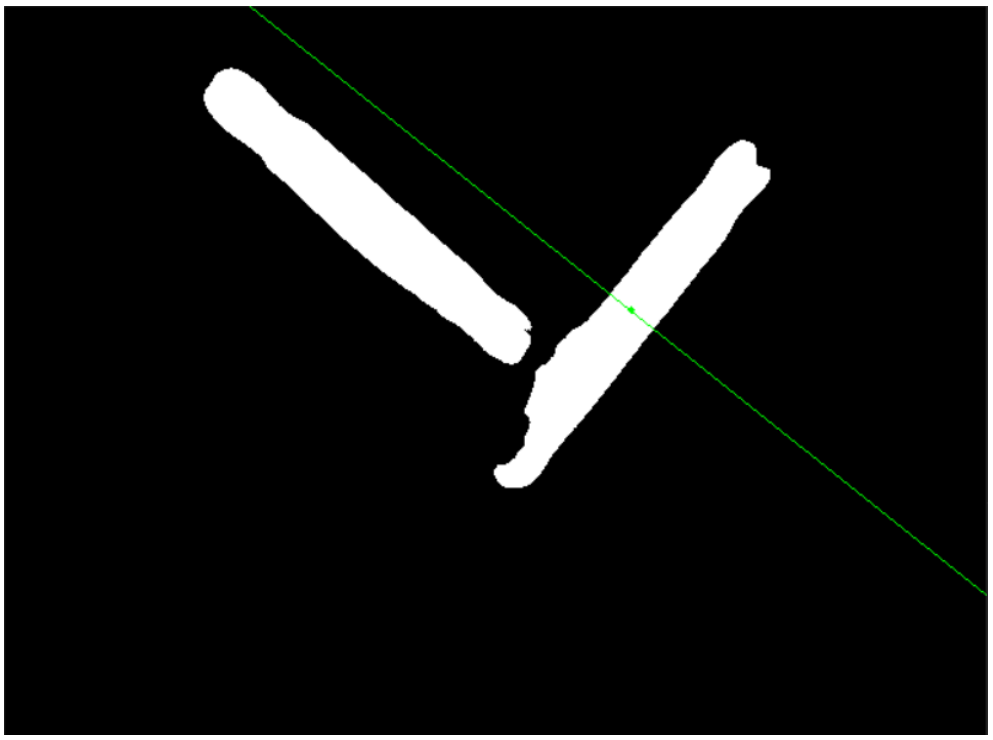

Figura 16. Línea utilizada para calcular el ángulo de rotación de la pieza

Con las coordenadas de los centroides de las piezas y sus ángulos de rotación, el último paso necesario para que el robot pueda operar es obtener las coordenadas del mundo real de las piezas.

3.5 Obtención de las coordenadas del mundo real

Antes de obtener las coordenadas del mundo real, se debe realizar un filtrado necesario para determinar qué objeto debe coger el robot. Con el objetivo de obtener un sistema más robusto, se tratará de coger la pieza que se encuentre en la parte superior de la pila, es decir, la más cercana a la cámara. De esta manera será más sencillo que el robot coja el objeto sin ningún problema.

Para conseguir este propósito, en primer lugar se debe obtener la distancia a la que se encuentran las piezas de la cámara. Esto se consigue llamando a la variable que almacena el frame de profundidad (*depth*) especificando, en nuestro caso, las coordenadas del centroide de la pieza. A continuación se define una variable (*z_min*) con un valor elevado (1000). Seguidamente, mediante un bucle for, se compara la distancia entre la cámara y el objeto de cada uno de los objetos. Si la distancia es menor, que la actual registrada como menor en *z_min,* se actualiza el valor y se calculan las coordenadas en el mundo real.

Para calcular las coordenadas en el mundo real en centímetros, se ha definido la función *get_3D_points()*, la cual, usando una serie de funciones de la librería

pyrealsense2 nos permite obtener los datos que deseamos. Estas coordenadas, junto al ángulo calculado anteriormente, son los datos necesarios para que el robot pueda actuar.

En las figuras 17 y 18 se pueden observar el proceso definido anteriormente y la función utilizada para el cálculo de las coordenadas en centímetros.

```
objs = puntosMedios_Segmentaciones(rgb,y_out)
z_min = 1000.0
pts_mtrs = []

for i in range(1, len(objs)+1):
   x = int(objs[i][0][0])
   y = int(objs[i][0][1])
   z = int(depth[y,x])

   if z <= z_min and z > 0.0:
       z_min = z
       angle = objs[i][2]

       [x,y,z] = dc.get_3D_points(x,y,z)
       pts_mtrs = [int(x),int(y),int(z)]
```

Figura 17. Selección de pieza y btención de las coordenadas del mundo real

```
def get_3D_points(self, x_pix, y_pix, depth):
   frames = self.pipeline.wait_for_frames()
   aligned_frames = self.align.process(frames)
   aligned_depth_frame = aligned_frames.get_depth_frame()
   color_frame = aligned_frames.get_color_frame()

   depth_intrin =
aligned_depth_frame.profile.as_video_stream_profile().intrinsics

   points_meters =
rs.rs2_deproject_pixel_to_point(depth_intrin,[x_pix,
y_pix],depth)
   return points_meters
```

Figura 18. Función para obtener las coordenadas del mundo real

4. Control del brazo robótico

El brazo robótico utilizado es el Sawyer Research Robot, un robot colaborativo desarrollado por Rethink Robotics[3]. Este robot presenta una pinza en su efector final y está equipado con sensores que le permiten interactuar de manera segura con los trabajadores humanos en tareas de ensamblaje o inspección entre otras aplicaciones.

Figura 19. Robot Sawyer de Rethink Robotics

El robot es sencillo de usar y está listo para ello en cuestión de pocos minutos. Además, está diseñado para ser fácil de programar y reconfigurar, lo que lo hace versátil y adecuado para una variedad de aplicaciones.

Para controlar el brazo robótico se hace uso de dos scripts que se comunican entre ellos. Uno de ellos se encarga principalmente del procesado y análisis de la imagen, mientras que el otro se encarga del movimiento del brazo. A continuación se muestran dos flowcharts, los cuales representan las acciones que realizan ambos scripts. Más adelante se explica detalladamente el funcionamiento de estos dos programas.

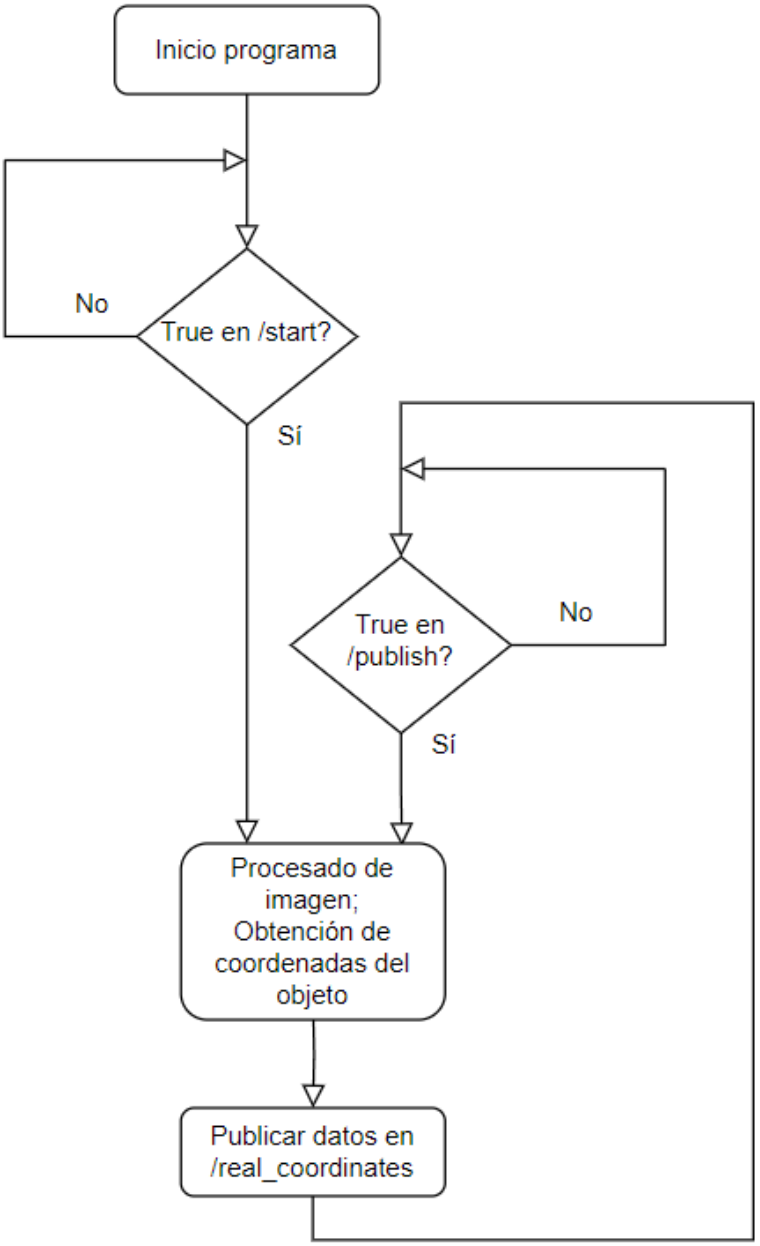

Figura 20. Flowchart del script de procesado de imagen

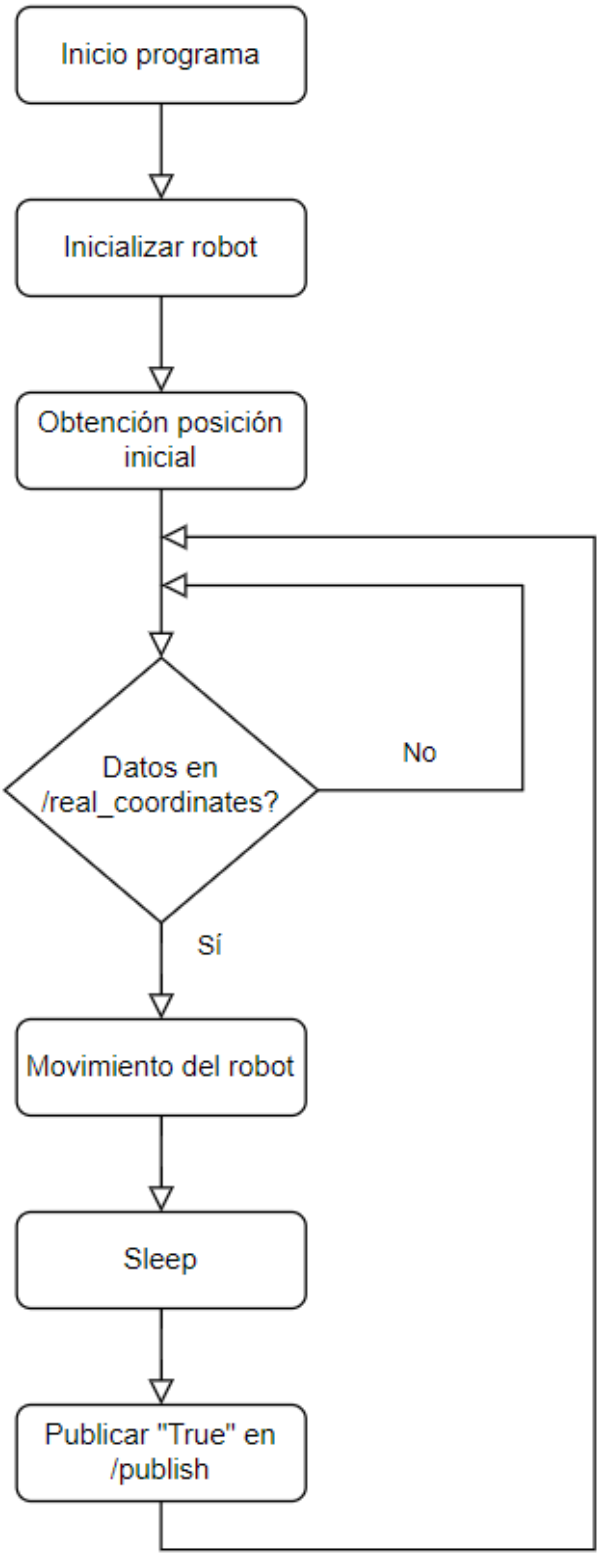

Figura 21. Flowchart del script de control del robot

4.1 Comunicación

El robot Sawyer utiliza ROS[4] para comunicarse con la estación de trabajo de desarrollo del usuario. Para ello ha de crearse una red local con la que se establecerá una conexión Ethernet entre Sawyer y la estación de trabajo con conectividad bidireccional completa. Para ello haremos uso de un router y conectaremos a él tanto el robot como la Jetson Orin.

El siguiente paso es obtener el workspace donde se va a poder trabajar con el robot. Desde la página web de Rethink Robotics se puede obtener dicho espacio de trabajo. A continuación debemos informar al robot y a la Jetson de las IPs del otro. Para ello accederemos al archivo "intera.sh" presente en el workspace y declaramos en "*your_ip=*" la IP de la Jetson Orin y en "*robot_hostname=*" el hostname del robot. Este último dato lo podemos averiguar buscando en un buscador, como Google, la IP del router. Desde ahí podremos obtener más detalles sobre nuestra red local, incluyendo el hostname del robot que necesitamos.

Una vez definido los datos necesarios en "intera.sh" ya podemos interactuar con el robot Sawyer. Para comunicarnos con él como usuario disponemos de un modelo de comunicación basado en el modelo de publicador/suscriptor a través de su integración con el protocolo ROS.

Este modelo funciona de la siguiente manera: ROS presenta una serie de tópicos en los que se presentan distintos tipos de información. Uno puede suscribirse a estos tópicos para acceder a la información que almacenan o se puede publicar en ellos para enviar mensajes o información importante. A continuación se detalla como se ha trabajado con esta arquitectura.

4.2 Topics

Los topics de ROS pueden almacenar cada uno un tipo de información concreta. Esto puede ser un número, un booleano o una estructura de datos más compleja. Para trabajar con ellos se va a utilizar *rospy*, una librería de python que permite fácilmente interactuar con todos los aspectos de ROS, incluyendo los topics.

En caso de querer publicar en un topic se debe en primer lugar crear un publicador con la función *Publisher()*. En esta se debe especificar el topic en el que se quiere publicar, el tipo de dato que se va a publicar y el tamaño de la cola de datos que se pueden enviar. Cuando se quiera publicar información se debe llamar al publicador y con la función *publish()* se publicará la información que se especifique.

Para suscribirse a tópicos se procede de manera similar. En este caso se debe utilizar la función *Subscriber()*, en la cual se debe especificar el topic al que suscribirse, el tipo de dato que se obtendrá y el nombre de una función de callback que se ejecutará cuando se obtengan nuevos datos en el topic.

A continuación se explica detalladamente el funcionamiento del sistema desarrollado y de todos los topics involucrados en el proceso. En la figura xX se muestra un esquema de cómo se comunican las distintas partes del sistema.

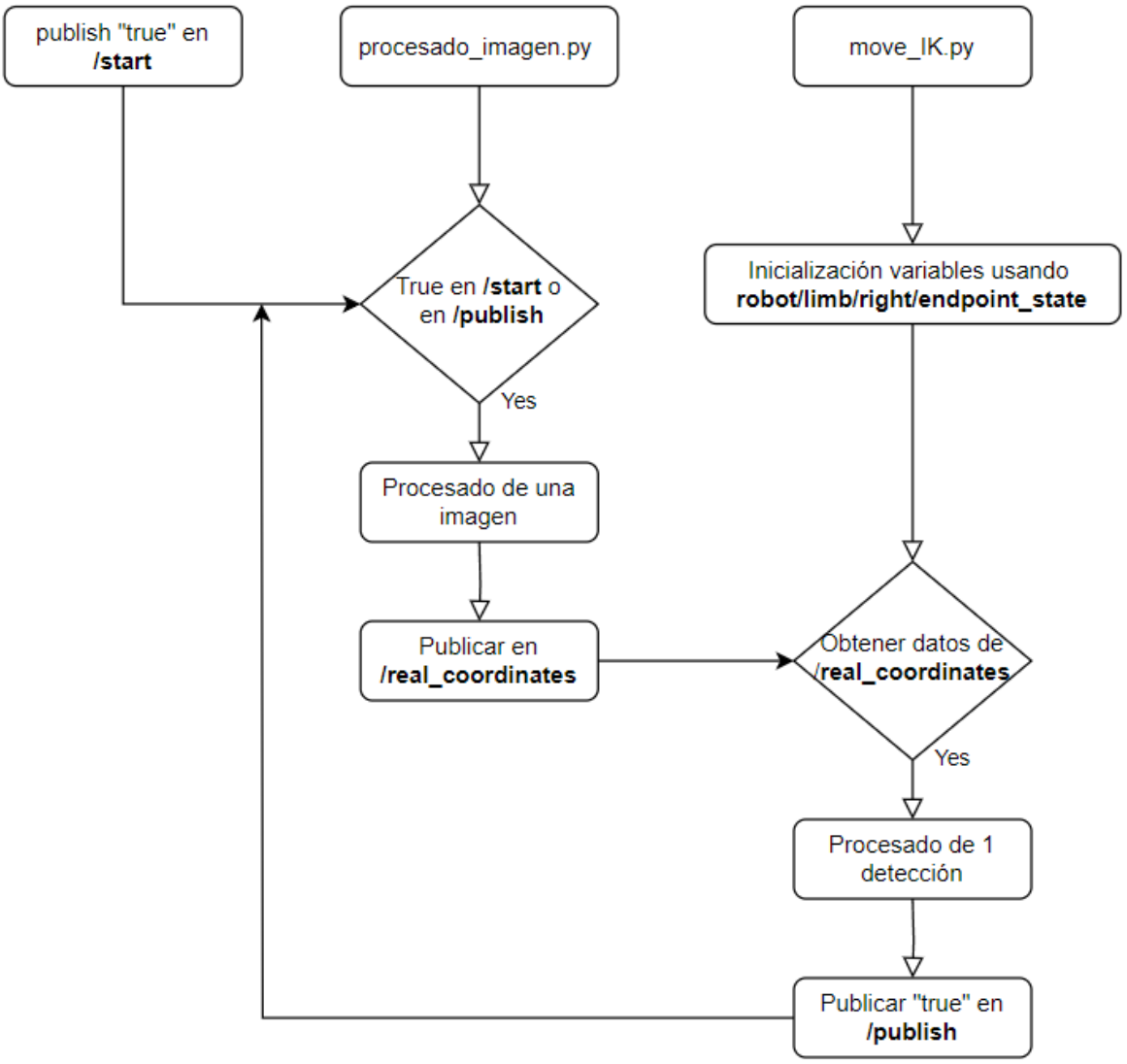

Figura 22. Esquema de la comunicación entre las partes del sistema

4.2.1 start

Este topic almacena un valor booleano, inicialmente *False,* que se utiliza para poner en marcha el sistema. Al poner el sistema en funcionamiento, el sistema está en pausa

hasta que se publique manualmente un *True* en este topic. Tras ocurrir esto, el sistema entra en su bucle de ejecución normal hasta que éste se detenga también manualmente.

4.2.2 robot/limb/right/endpoint_state

Este topic almacena un tipo de dato llamado *EndpointState.* Este tipo de dato contiene mucha información acerca de la posición del efector final del robot. En nuestro caso vamos a utilizar la información del campo *pose*, siendo esto los valores de posición y rotación en cuaterniones del efector final.

Esta información se utiliza en el script de control del movimiento del robot. En primer lugar se inicializa el robot, colocándolo en una posición inicial predefinida. Tras esto, el script se suscribe al topic. La callback se encarga de almacenar en variables los valores de la posición y rotación inicial del efector final en cuaterniones y en ángulos de Euler, los cuáles se utilizarán más tarde.

4.2.3 real_coordinates

El topic *real_coordinates* se utiliza para el control del movimiento del brazo. En este topic se publican manualmente datos del tipo *Pose.* Este tipo de dato contiene los valores de posición y rotación en cuaterniones del efector final.

En este topic se publican datos al final del bucle de procesado de imagen. La información publicada son las coordenadas tridimensionales de la pieza que se debe coger, junto a la rotación necesaria de la pinza.

Por otro lado, el script del control del movimiento del brazo se suscribe a este topic. Al recibir datos, se ejecuta la función de callback, la cual ejecuta el movimiento del robot, el cual consiste en ir a la posición de la pieza, cogerla y llevarla a un lugar especificado con anterioridad. Tras esto, se publica un True en el topic *publish*, el cual sirve para lanzar otra iteración del bucle de procesado de imagen, permitiendo que el sistema se ejecute continuamente de manera automática.

4.2.4 publish

Como se ha indicado anteriormente, el topic *publish* se usa como medio de comunicación entre el script encargado de controlar el robot y el script que se encarga de procesar las imágenes. En él se publican valores booleanos.

El script de procesado de la imagen se suscribe a este topic. Cuando en él se publica un *True*, la función de callback cambia el valor de una variable (*publish*) a 1, lo cual

causa que se realice una iteración del bucle de procesado de imagen. Al final de este bucle, *publish* vuelve a establecerse a 0.

Por otro lado, el script de control del robot publica un *True* en este topic tras realizar un bucle de movimiento del brazo. Esto permite que se analice otra imagen cuando el robot ha vuelto a su posición inicial, de manera que los dos scripts se turnan para realizar sus instrucciones.

4.3 Movimiento del brazo

El movimiento del brazo es un pick and place secuencial básico que se ejecuta cada vez que la red neuronal identifica una pieza. Mientras se esté ejecutando el movimiento no podrán llegarle al robot otras instrucciones excepto una parada forzosa.

Para mover el brazo, el sistema operativo del robot utiliza movimientos absolutos respecto al origen del robot. Sin embargo, las coordenadas que proporciona nuestro sistema son respecto a la cámara, la cual al mismo tiempo no está perfectamente colocada en el centro del efector final. Es por ello que debe realizarse una conversión entre las distancias que proporciona la red neuronal y las distancias que debe moverse el robot.

Esta conversión se detalla a continuación, ilustrada con ayuda de la figura 23:

- El punto A es el origen de coordenadas del robot.
- La posición del efector final (B) es conocida respecto al punto A.
- La distancia de la cámara (C) respecto a la posición del efector final (B) es conocida. Llamaremos a esta distancia $\overline{CB}$
- La posición de la pieza a coger (D) es conocida respecto a la cámara (C). Llamaremos a esta distancia $\overline{DC}$.
- Se busca conocer la posición de la pieza (D) respecto al robot (A). Dadas las afirmaciones anteriores esta se puede calcular de la siguiente manera:

$$D = B + \overline{CB} + \overline{DC}$$

- Debido a que la posición de la pinza (B) está al límite de la misma, se ha declarado que la coordenada z de la posición de D sea 1,5 cm más baja, de manera que se obtiene un agarre de las piezas más fiable.

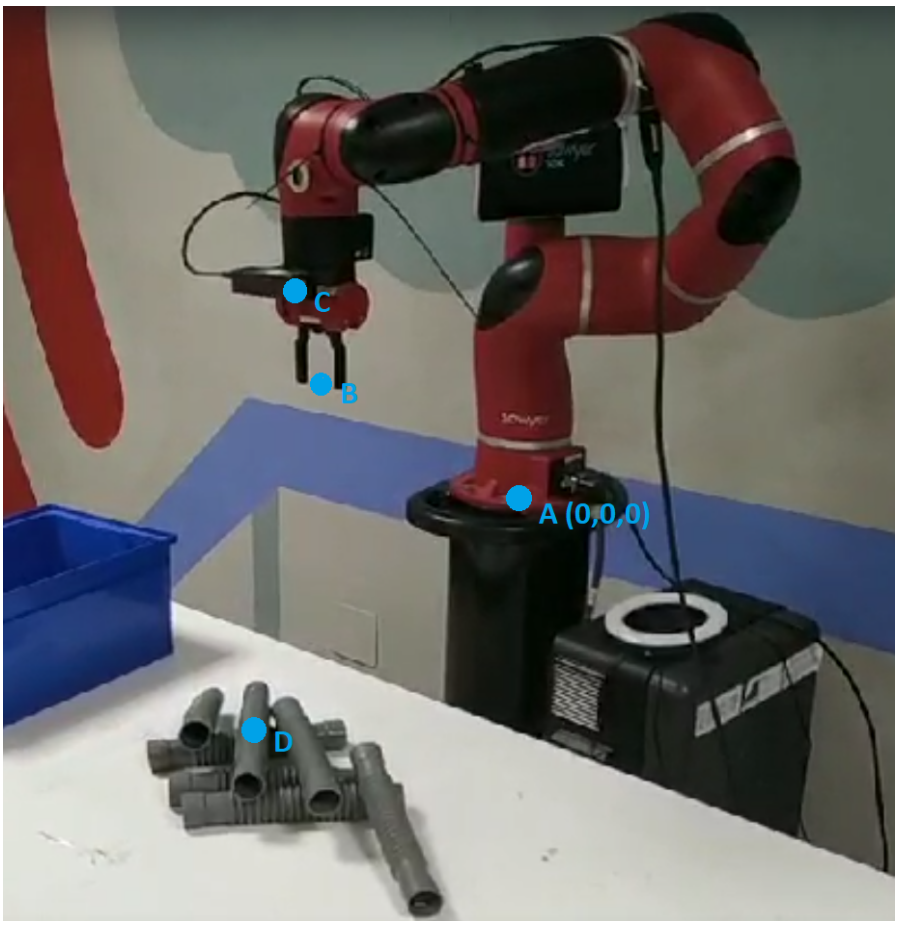

Figura 23. Mapa de coordenadas de los puntos de interés del sistema

Para realizar correctamente el movimiento se ha definido una función la cual se encargará de realizar el movimiento deseado.

4.3.1 Pick and place

La función de pick and place recibe tres parámetros. La posición a la que debe moverse el robot, la posición inicial del robot y su orientación inicial. Estos dos últimos son necesarios porque el movimiento se realiza con coordenadas absolutas respecto al origen del robot y no respecto a su posición actual.

El algoritmo de pick and place diseñado sigue la estructura mostrada en el siguiente flowchart:

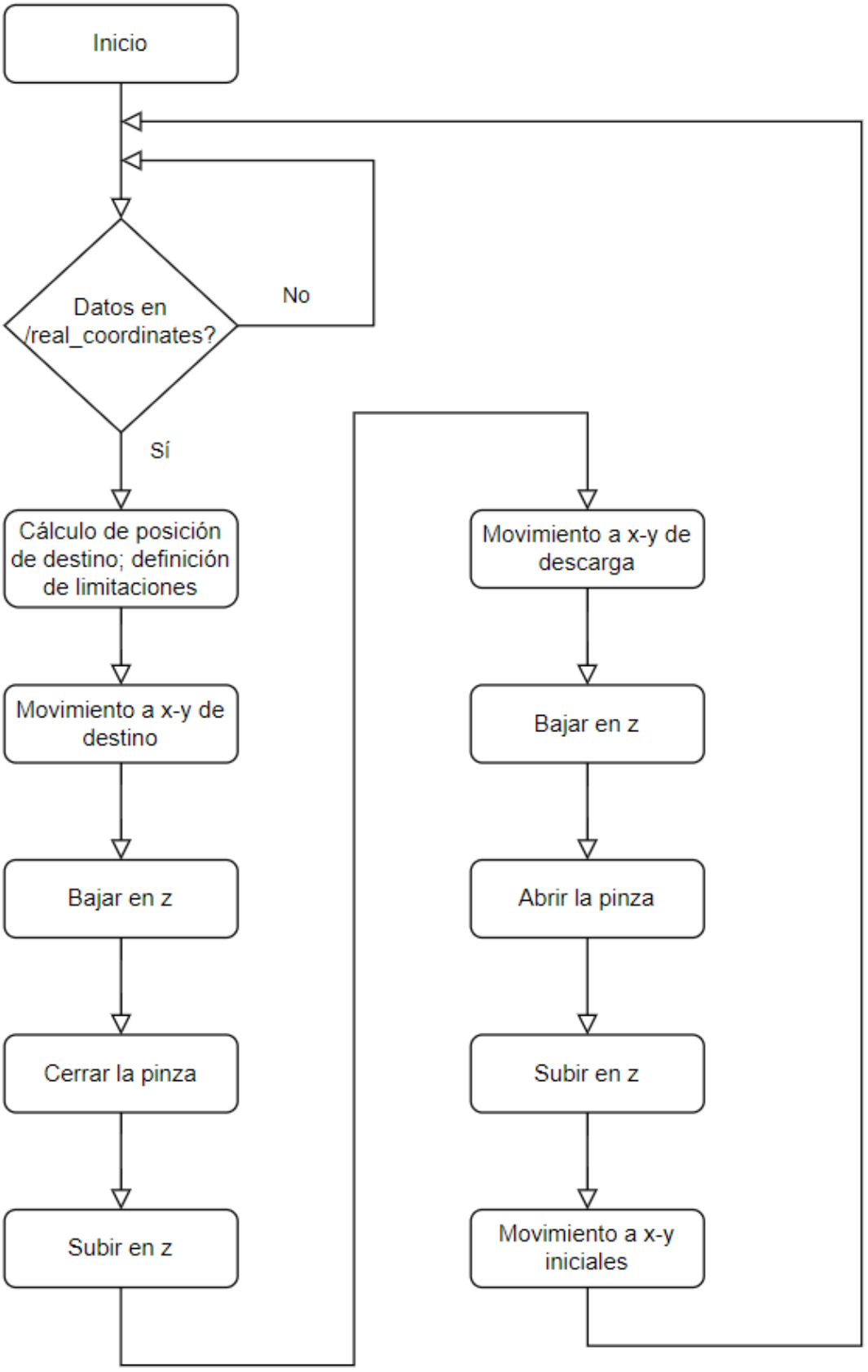

Figura 24. Flowchart del algoritmo de pick and place

El funcionamiento del algoritmo se detalla a continuación. En primer lugar se deben almacenar en variables los valores de posición y rotación del efector final del robot convertidos a las unidades en las que trabaja el robot. En nuestro caso debemos convertir las distancias a metros. Con estos valores se procede a calcular las coordenadas de la pieza respecto al robot conforme se ha explicado en el apartado anterior. Por otro lado, también es necesario especificar las coordenadas de descarga de los objetos.

Otra restricción que se debe aplicar es limitar el movimiento del robot en el eje z, con la intención de evitar que el robot se choque contra la superficie de trabajo. Esta restricción es meramente una precaución, ya que el sistema nunca detectará un objeto más bajo que la superficie de trabajo.

Con todas estas limitaciones y condiciones definidas, el último paso antes de mover el brazo es calcular la posición y la rotación objetivos, las cuales se calculan en base a los parámetros definidos anteriormente y a los parámetros recibidos por la función.

Finalmente, se puede proceder a realizar el movimiento del robot. Éste se realiza en 9 pasos, tal y como se muestra en el flowchart anterior. Una vez terminado el movimiento el robot está preparado para recibir nuevas coordenadas y volver a ejecutar sus instrucciones.

5. Pruebas de validación

Con la intención de verificar el correcto funcionamiento del sistema desarrollado, se han realizado un par de pruebas experimentales. Por un lado se ha comprobado la efectividad de la red neuronal y del tiempo de ejecución necesario para realizar las instrucciones que se le piden. Por otro lado, se ha verificado que el movimiento del robot funcione como se espera.

A continuación se detallan las pruebas realizadas junto a un resumen de los resultados obtenidos en las distintas pruebas.

5.1 Identificación de objetos

Las predicciones de la red neuronal son muy precisas cuando hay pocos objetos en la zona de trabajo (figura 25). No obstante, conforme aumenta el número de objetos las predicciones empeoran (figuras 26 y 27). Estas predicciones erráticas no tienen porqué implicar un mal funcionamiento del sistema ya que, en primer lugar, el sistema trata de agarrar el objeto más cercano a la cámara. por lo que el ruido de las piezas de la parte inferior no afecta en nada. Además, el sistema no necesita reconocer perfectamente las piezas, solo debe ser capaz de cogerlas, por lo que una predicción de una calidad lejana a la perfecta, puede ser suficiente para coger la pieza correctamente.

Figura 25. Ejemplo de predicción de la red neuronal

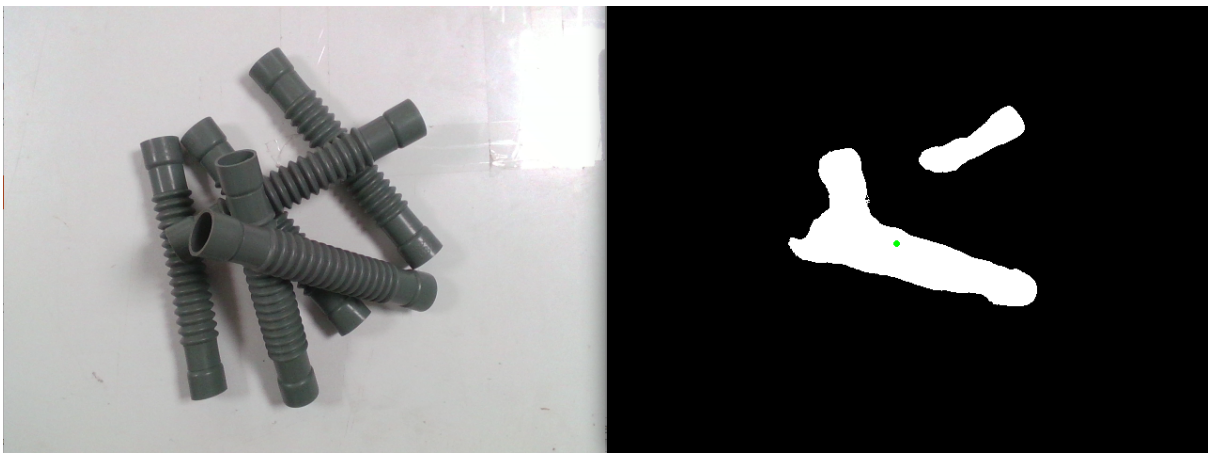

Figura 26. Ejemplo de predicción de la red neuronal

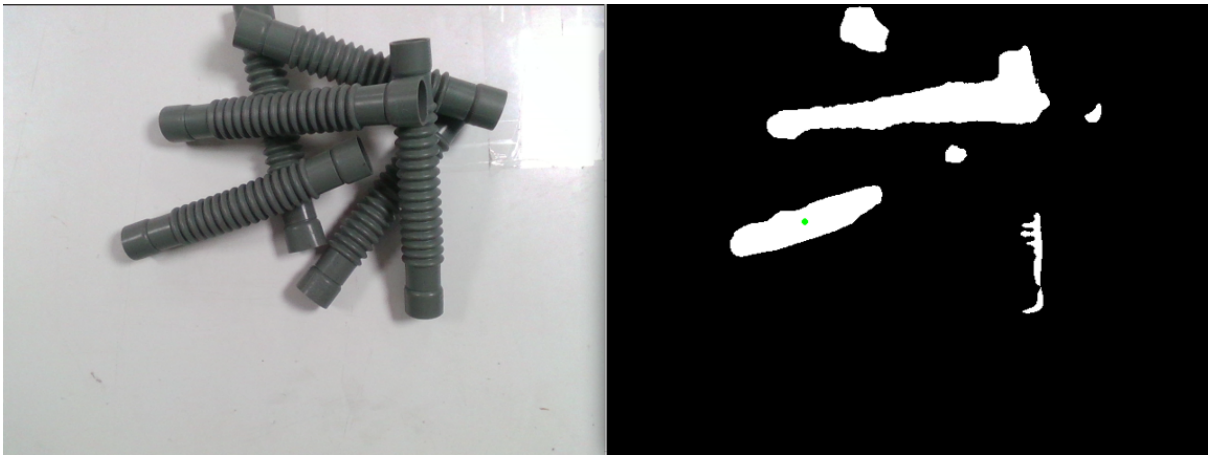

Figura 27. Ejemplo de predicción de la red neuronal

Teniendo estos dos puntos en cuenta, se han realizado 150 pruebas de agarre de piezas, con distintas distribuciones y con distintas cantidades de piezas. De todas ellas, solo en 3 el robot fallaba en reconocer la pieza, resultando en un 98% de efectividad.

5.1.1 Comparativa de tiempos

Anteriormente se ha hablado sobre las ventajas de utilizar la NVIDIA Jetson AGX Orin para aplicaciones de este tipo. Para verificar su efectividad se han realizado y cronometrado el tiempo de cómputo de una serie de pruebas en la propia Jetson y en un ordenador portátil. Las especificaciones de ambos se pueden ver en la siguiente tabla:

	Ordenador portátil	NVIDIA AGX Jetson Orin
CPU	AMD Ryzen 7 5800H, 8 núcleos	CPU Arm® Cortex®-A78AE v8.2, 12 núcleos
GPU	NVIDIA GeForce RTX 3050 Ti	GPU de arquitectura NVIDIA Ampere de 2048 núcleos con 64 núcleos Tensor
Frecuencia de la GPU	1,6 MHz	1,3 GHz

Estas pruebas se han realizado colocando las piezas en distintas posiciones y realizando varias pruebas para cada disposición de piezas. En total se han realizado más de 600 pruebas. En estas pruebas se ha cronometrado el tiempo de carga y procesado de imagen, la detección de objetos mediante la red neuronal y el tiempo necesario para la extracción de las características de los objetos. Las pruebas realizadas son las de las imágenes siguientes:

Figura 28. Disposición 1 de las pruebas realizadas

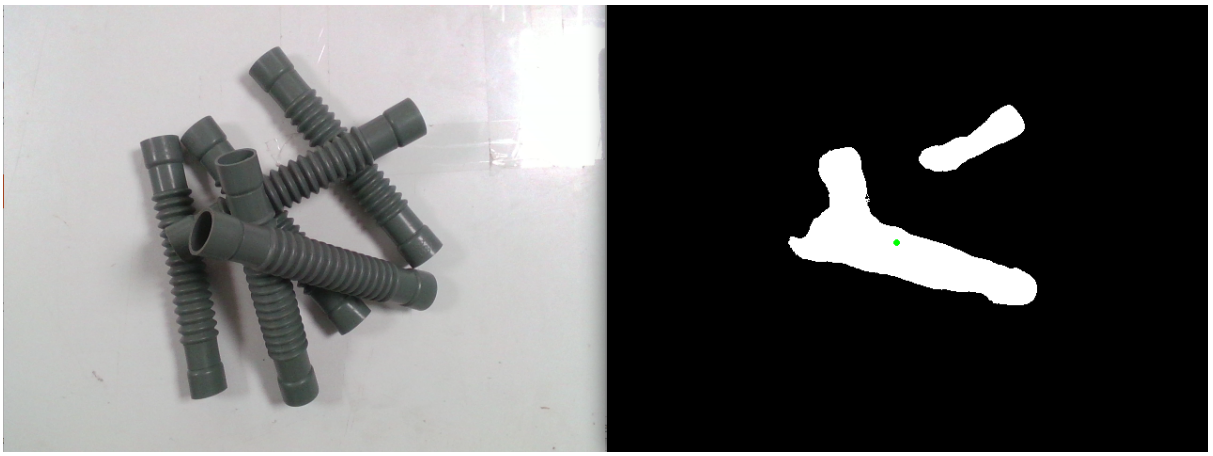

Figura 29. Disposición 1 de las pruebas realizadas

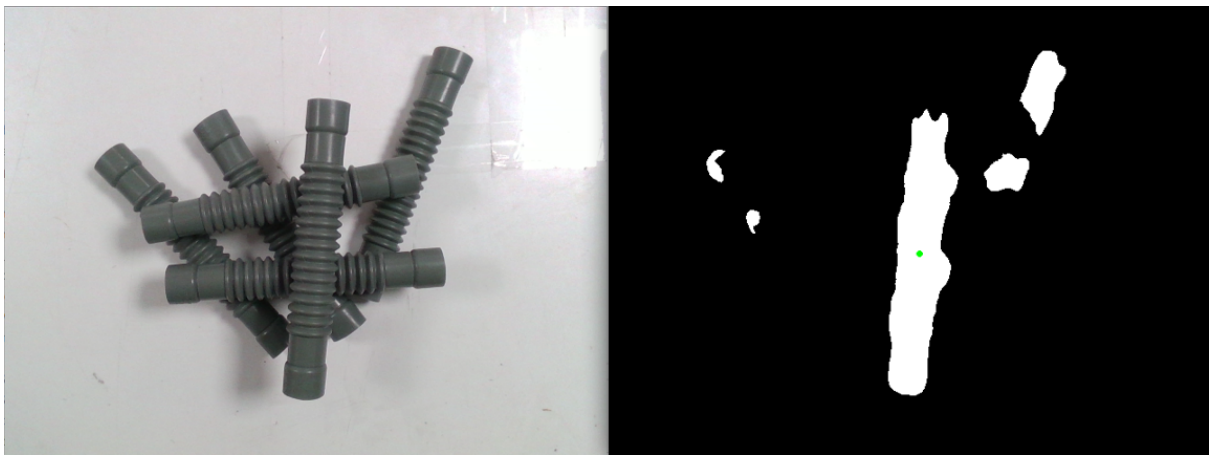

Figura 30. Disposición 1 de las pruebas realizadas

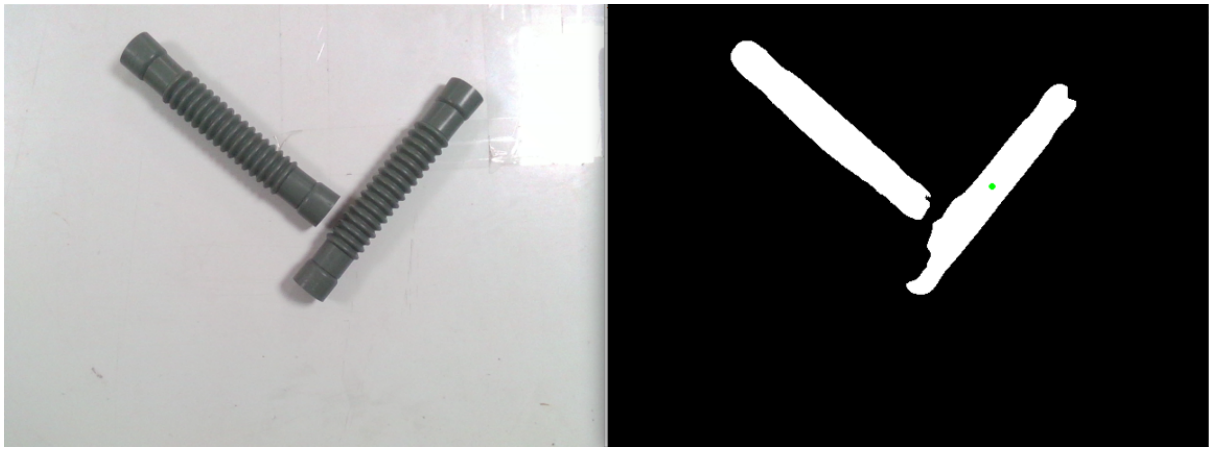

Figura 31. Disposición 1 de las pruebas realizadas

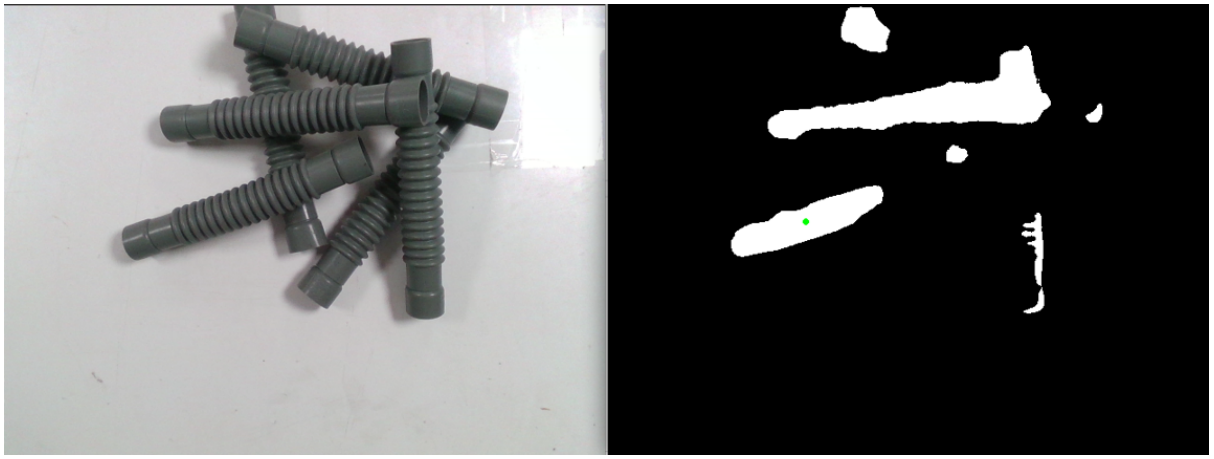

Figura 32. Disposición 1 de las pruebas realizadas

Antes de analizar los resultados, cabe destacar que los tiempos de ejecución son los mismos para todos los casos estudiados, por lo que se van a analizar los resultados generales.

Dicho esto, en primer lugar podemos ver en la figura 33 como el tiempo de adquisición de la imagen es prácticamente el mismo tanto en la Jetson como en el ordenador. Esto es probablemente debido a que este tiempo simplemente es el tiempo que tarda la información en viajar de la cámara al sistema. Además este tiempo representa una fracción muy pequeña del tiempo total del proceso, apenas unos 10 milisegundos, por lo que esta comparativa no es del todo relevante.

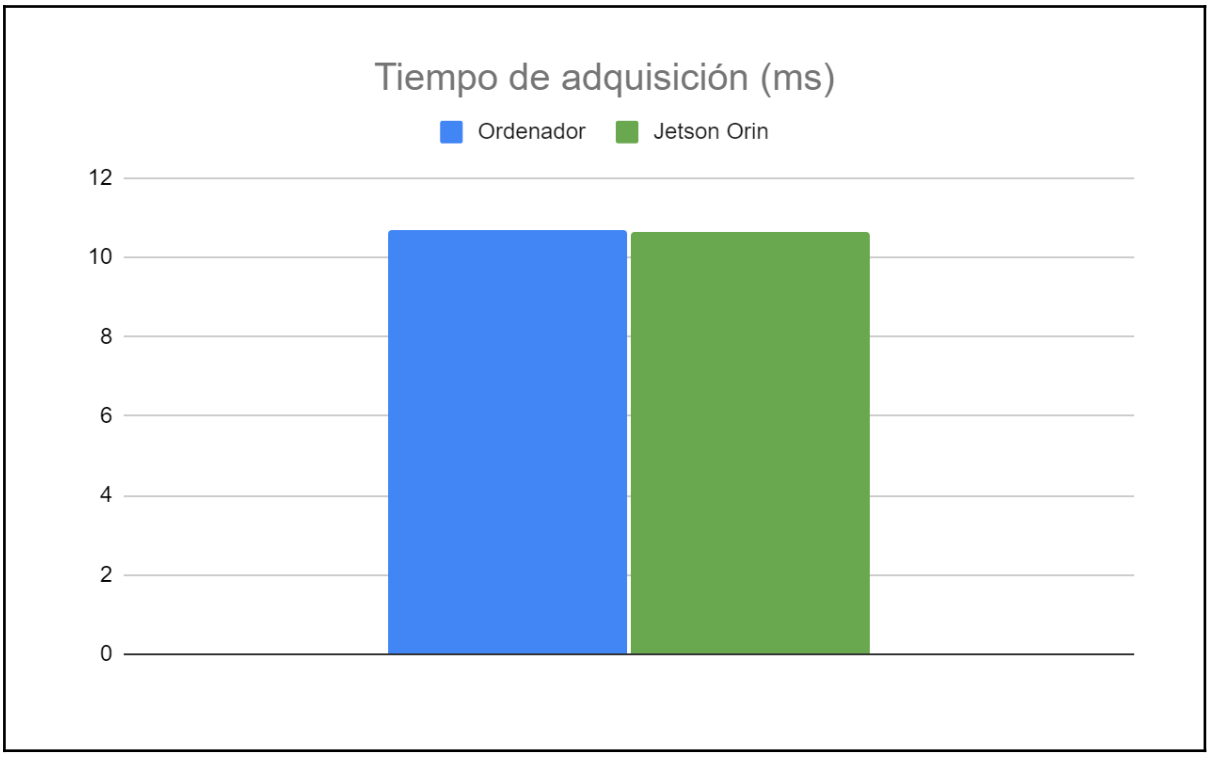

Figura 33. Comparativa del tiempo de adquisición de las imágenes

El tiempo de inferencia representa la mayor parte del tiempo de procesado de la imagen. Aquí se puede ver claramente la ventaja que supone utilizar la Jetson Orin para aplicaciones de este tipo, pudiendo inferir con la red neuronal en 750 ms contra los 1,5 s del ordenador, lo cual significa que la Jetson es el doble de rápida en esta tarea que el ordenador

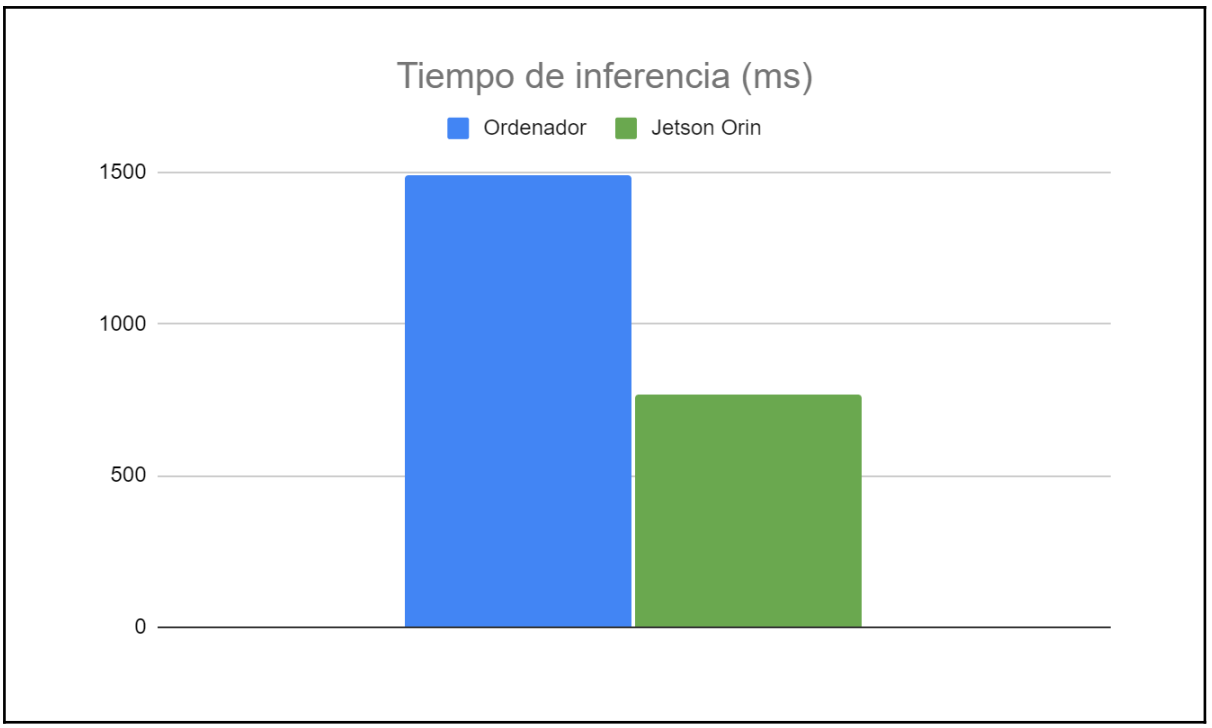

Figura 34. Comparativa del tiempo de inferencia con la red

Por último, se ha comparado el tiempo de extracción de las características relevantes de las piezas. En este aspecto la Jetson es también más rápida, aunque su superioridad no es tan evidente, siendo aproximadamente un 50% más rápida que el ordenador. Sin embargo, este tiempo representa también una fracción muy reducida del procesado total.

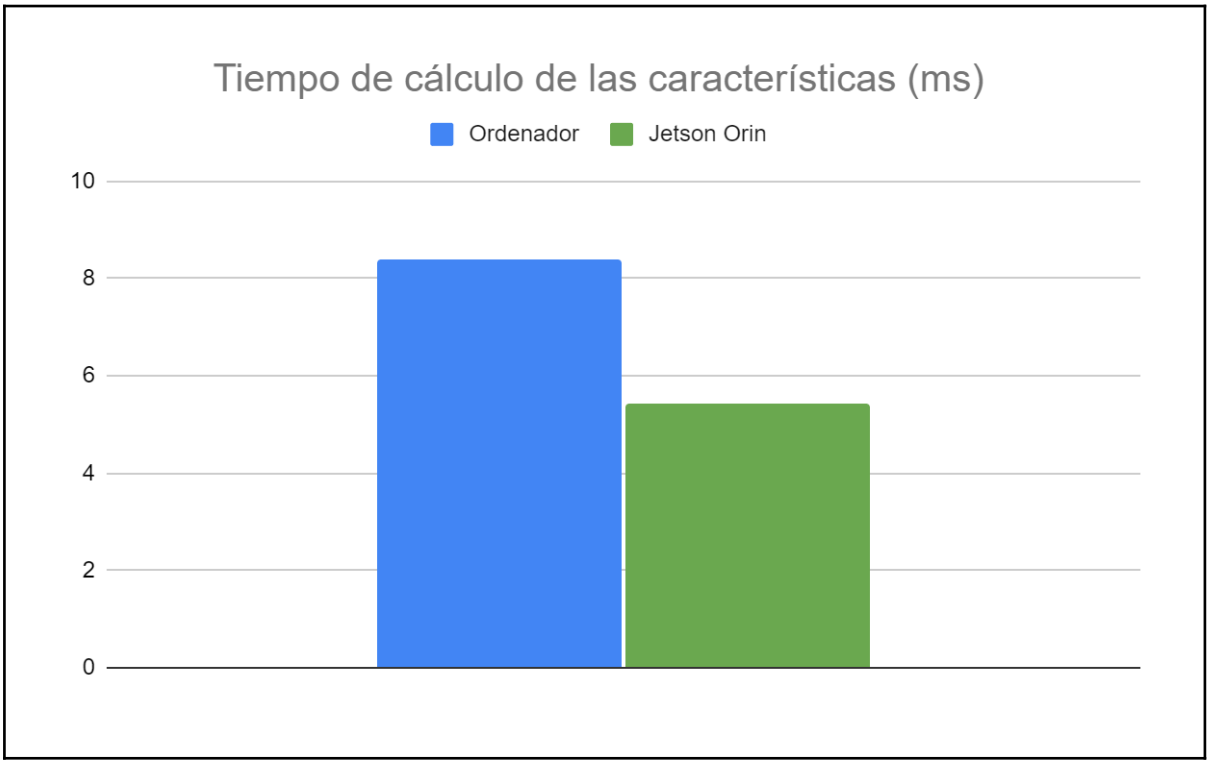

Figura 35. Comparativa del tiempo de cálculo de las características

En conclusión y como se puede ver en la figura 36, la comparativa de estas pruebas muestra una clara superioridad de la Jetson. Es capaz de realizar la misma tarea en la mitad del tiempo y de manera constante. Estas evidencias demuestran la utilidad y la importancia de utilizar un sistema empotrado especializado en este tipo de tareas.

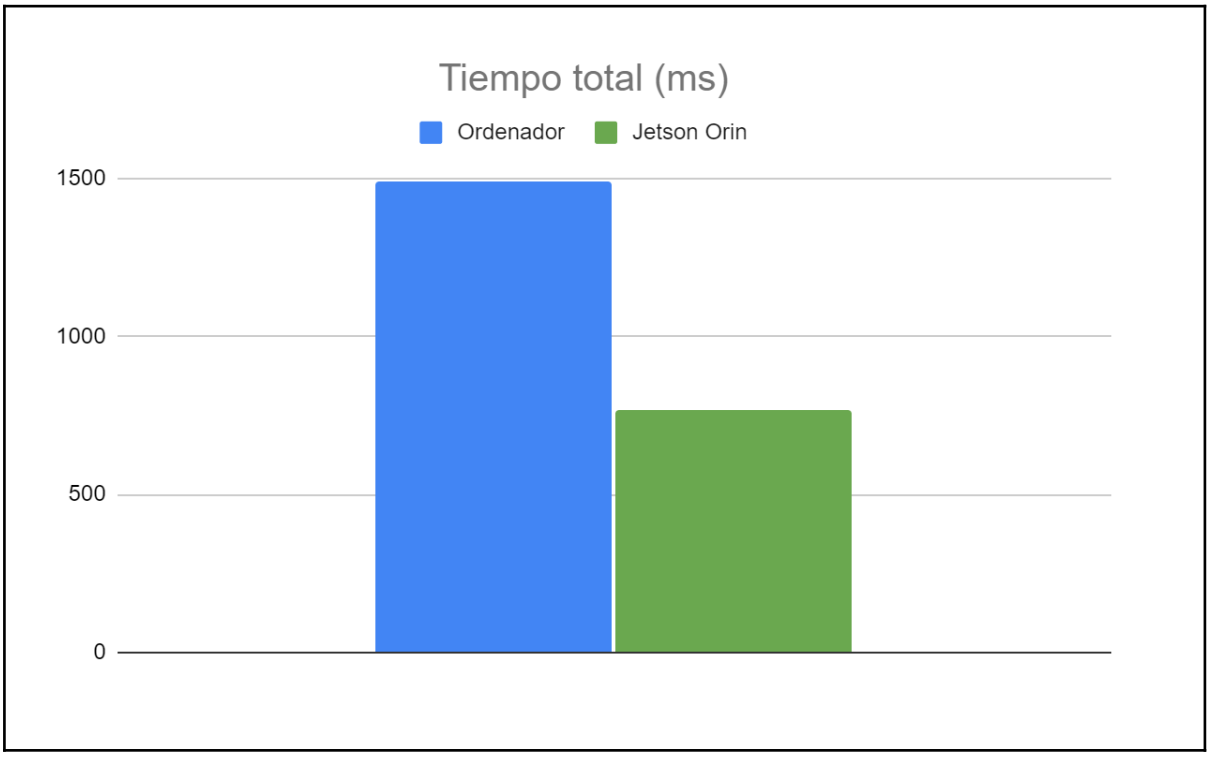

Figura 36. Comparativa del tiempo total de procesado de imagen

En la siguiente tabla se muestran detalladamente los resultados mostrados en las gráficas anteriores.

	Ordenador portátil	**NVIDIA AGX Jetson Orin**
Tiempo de adquisición	10,67 ms	10,62 ms
Tiempo de inferencia	1489,51 ms	767,74 ms
Tiempo de cálculo de las características	8,39 ms	5,44 ms
Tiempo total	**1508,57 ms**	**783,93 ms**

5.2 Movimiento del robot

Para la validación del funcionamiento del robot se ha puesto en funcionamiento el sistema mientras se realizaban una serie de pruebas. En total se han realizado 120 pruebas que han consistido en:

- Verificar el funcionamiento del sistema. Se ha puesto el sistema en funcionamiento en condiciones normales y se ha comprobado que opere como se espera durante un periodo prolongado.
- Alterar la posición de las piezas. Mientras el robot operaba se ha cambiado la posición de los cilindros. También se han añadido y se han retirado piezas mientras el robot operaba.
- Comprobar el comportamiento del sistema ante objetos extraños. Se han introducido en el campo de visión del sistema objetos distintos a los esperados por el sistema. El comportamiento esperado es que el sistema ignore estos objetos.
- Comprobar el comportamiento del sistema ante ningún objeto. Se ha dejado el espacio de trabajo libre de objetos. El comportamiento esperado es que el sistema no se mueva.

Al llevar a cabo los experimentos se ha podido verificar que el sistema responde satisfactoriamente en el 94% de los casos. Ni los objetos extraños ni la abundancia de piezas han causado que el sistema se comporte erráticamente. El causante de los experimentos fallidos ha sido la disposición de dos cilindros en paralelo y muy juntos. Pese a que la predicción se realiza correctamente, el problema se daba cuando el robot trataba de agarrar una pieza pero un extremo de la pinza chocaba con la pieza contigua, por lo que no se podía realizar el agarre.

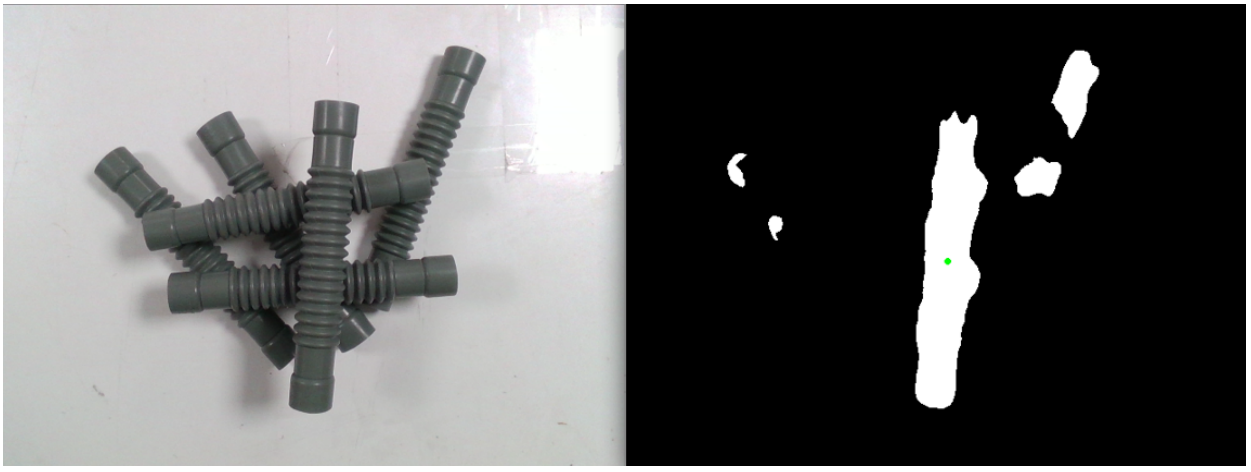

Figura 37. Predicción del sistema ante varios objetos

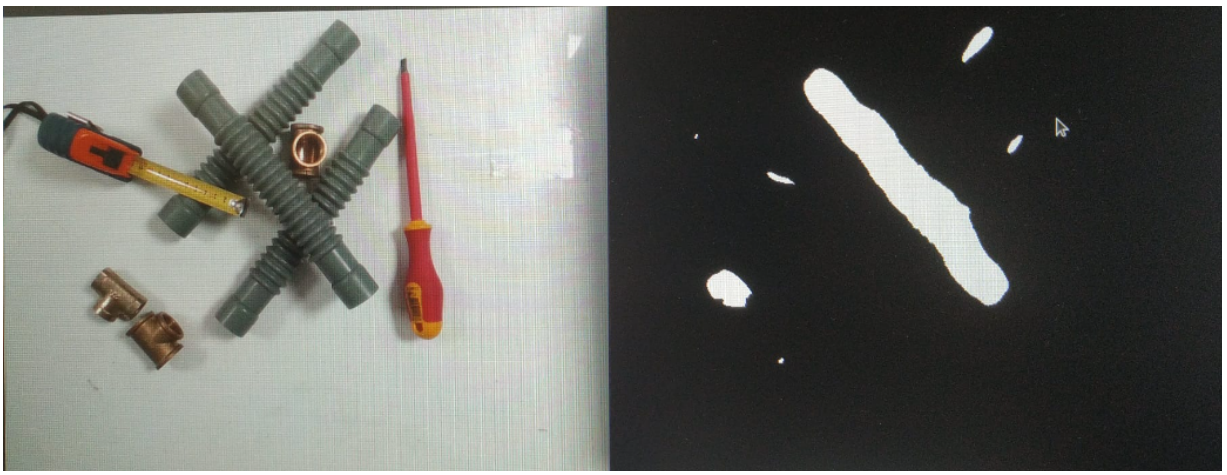

Figura 38. Predicción del sistema ante objetos extraños

Los resultados de las pruebas anteriores muestran que el sistema es responsivo, eficaz y capaz de funcionar ininterrumpidamente. En vista de todo ello se puede afirmar que el control y movimiento del robot se realiza correctamente tanto en situaciones normales como en situaciones adversas.

6. Resultados y conclusiones

En vista de lo enumerado en el apartado anterior se puede afirmar que el resultado del trabajo ha sido satisfactorio. Se han cumplido todos los objetivos propuestos de manera correcta.

Por un lado, se ha implementado la red neuronal de la que se disponía y se ha podido exprimir todo su potencial. Los resultados que ésta proporciona están a la altura de las expectativas, proporcionando información muy valiosa y en tiempo real, lo que permite analizar las piezas que detecta y extraer de manera precisa la información necesaria para poder maniobrar con las piezas.

Por otro lado, el control del robot se ha podido realizar de manera exitosa. Se ha conseguido establecer la comunicación con el robot para poder operar con él. Una vez hecho eso, se ha definido un algoritmo de control que permite realizar las operaciones que se desean. Los movimientos del robot son rápidos y la precisión en el agarre es casi perfecta, lo que permite al robot trabajar de manera fluida.

En conclusión, los resultados obtenidos se pueden definir como excelentes. Pese a ello y, como suele ser frecuente en proyectos relacionados con la visión por computador y la automatización, hay margen de mejora

6.1 Trabajos futuros

Aunque esto no esté incluido en el desarrollo del proyecto, un apartado en el que se puede mejorar es en la eficiencia de la red neuronal. Si bien los resultados que ésta proporciona son excelentes, cualquier mínima mejora puede ayudar al buen rendimiento del sistema, siempre que se dispongan de los recursos y el tiempo necesarios para poder trabajar en mejorar la red neuronal.

Otro apartado de posible mejora es la inferencia sobre la red neuronal. Se estudió realizar este proceso mediante TensorRT (Tensor Realtime). TensorRT es una biblioteca de inferencia de alta eficiencia desarrollada por NVIDIA para acelerar el despliegue de modelos de *deep learning* en hardware NVIDIA. TensorRT optimiza y acelera el proceso de inferencia de los modelos al aprovechar al máximo las capacidades de cómputo de la GPU.

Finalmente no se desarrolló está solución debido a algunos problemas ocasionados al convertir el modelo del que se disponía al formato de TensorRT. Pese a que la solución adoptada es también muy rápida, se podría en un futuro tratar de mejorar el sistema implementando esta solución.

Por último, entre cada iteración de funcionamiento del robot el sistema debe detenerse durante unos breves segundos. Esto es debido a que el robot y la cámara deben estar en completo reposo al realizar la captura de imágenes con el fin de obtener una imagen nítida en la que se distingan bien los objetos. Se podría tratar de reducir este tiempo de espera, ya sea bien realizando algunos ajustes en la programación o incrementando la rigidez del acoplador de la cámara al brazo robótico.

7. Bibliografía

[1] (n.d.). OpenCV - Open Computer Vision Library. https://opencv.org/

[2] (n.d.). TensorFlow.org. https://www.tensorflow.org/

[3] (n.d.). Rethink Robotics: Smart Collaborative Robots. https://www.rethinkrobotics.com/

[4] (n.d.). ROS: Home. https://www.ros.org/

[5] Aparicio Alonso, E. (2022). *Detección en tiempo real del punto de agarre del objeto más accesible en entornos industriales mediante aprendizaje profundo*.

[6] *CAT90-SA Semi Automated SMT Pick and Place Machine*. (n.d.). ATCO-US. https://www.atco-us.com/cat90-sa-smt-pick-and-place-systems/

[7] Dekhtiar, J., Zheng, B., Verma, S., & Tekur, C. (2021, January 28). *Leveraging TensorFlow-TensorRT integration for Low latency Inference*. The TensorFlow Blog. https://blog.tensorflow.org/2021/01/leveraging-tensorflow-tensorrt-integration.html

[8] *Depth Camera D415 – Intel® RealSense™ Depth and Tracking Cameras*. (n.d.). Intel RealSense. https://www.intelrealsense.com/depth-camera-d415/

[9] *Jetson Orin para robótica de próxima generación*. (n.d.). NVIDIA. https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-orin/

[10] Luqman, M. (2022, October 18). *ROS/Tutorials/WritingPublisherSubscriber(python)*. ROS Wiki. http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

[11] *Networking*. (2022, February 8). Rethink Robotics. https://support.rethinkrobotics.com/support/solutions/articles/80000980135-networking

[12] *Quick Start Guide :: NVIDIA Deep Learning TensorRT Documentation*. (2023, May 1). NVIDIA Docs. https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html

8. Anexos

8.1 Script de procesado de imagen

```
import tensorflow as tf
import PIL
import numpy as np
import cv2
import rospy
import sys
import time

from data_loaderD import DataLoader
from realsense_depth import *
from extraer_agarre import *
from geometry_msgs.msg import Pose
from math import radians
from std_msgs.msg import Bool

#Callback suscrita al topic /start donde se publica True para indicar
que se empieza
def startCallback(data):
    global publish
    if data.data is True:
        publish = 1

#Callback suscrita al topic /publish donde se publica un True cada vez
que el robot a cogido una pieza
def publishCallback(data):
    global publish
    if data.data is True:
        publish = 1


def main():
    global publish
    publish = 0

    model_path='/home/labi40_ai2/TFM/modelos/unet_247_D_32_1024_Adam.h5'

    height = 480
    width = 640

    # Carga el modelo
    model = tf.keras.models.load_model(model_path)

    # Variables for publisher
    pose = Pose()
    pub = rospy.Publisher('/real_coordinates', Pose, queue_size=1)
```

```
rospy.init_node('camera_part', anonymous=True)
rate = rospy.Rate(100)

# Subscriber
rospy.Subscriber('/start', Bool, startCallback)
rospy.Subscriber('/publish', Bool, publishCallback)

dc = DepthCamera()

while True:
    ret, depth_frame, color_frame = dc.get_frame()
    rgb = color_frame
    depth = depth_frame
    rgbd = np.zeros((rgb.shape[0], rgb.shape[1], 4))
    rgbd[:, :, 0] = rgb[:, :, 0]
    rgbd[:, :, 1] = rgb[:, :, 1]
    rgbd[:, :, 2] = rgb[:, :, 2]
    rgbd[:, :, 3] = depth[:, :]
    rgbd = rgbd / 255.0
    rgbd = np.expand_dims(rgbd, axis=0)

    y_pred = model.predict(rgbd)
    y_pred = y_pred[0, :, :, 0]
    y_pred[y_pred >= 0.5] = 1.0
    y_pred[y_pred < 0.5] = 0.0
    y_out = np.zeros(y_pred.shape + (3,))
    y_out[:, :, 0] = y_pred[:, :] * 255
    y_out[:, :, 1] = y_pred[:, :] * 255
    y_out[:, :, 2] = y_pred[:, :] * 255
    y_out = y_out.astype(np.uint8)

    if publish == 1:
        objs = puntosMedios_Segmentaciones(rgb,y_out)
        z_min = 1000.0
        pts_mtrs = []

        for i in range(1, len(objs)+1):
            x = int(objs[i][0][0])
            y = int(objs[i][0][1])
            z = int(depth[y,x])
            if z <= z_min and z > 0.0:
                z_min = z
                angle = objs[i][2]

                [x,y,z] = dc.get_3D_points(x,y,z)
                pts_mtrs = [int(x),int(y),int(z)]

        ## Usar x y z angle
        if not pts_mtrs:
            print("No detections")
        else:
```

```
                publishPose(pose, pub, rate, pts_mtrs, angle, 1, 1)
                print("Publicado", pts_mtrs, angle)
                publish = 0

        cv2.imshow('Color', color_frame)
        cv2.imshow('Prediccion', y_out)
        cv2.waitKey(1)

if __name__ == '__main__':
    sys.exit(main() or 0)
```

8.2 Funciones de extracción del agarre

```
import numpy as np
import cv2
from numpy import asarray

import imutils
import cv2
from scipy.spatial import distance
import math
from collections import OrderedDict


def puntosMedios_Segmentaciones(img_filename, seg_filename):

   objetos = {}
   n = 1

   img = img_filename
   seg = seg_filename
   gris = cv2.cvtColor(seg, cv2.COLOR_BGR2GRAY)
   blurred = cv2.GaussianBlur(gris, (5, 5), 0)
   thresh = cv2.threshold(blurred, 60, 255, cv2.THRESH_BINARY)[1]

      contornos = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
   contornos = imutils.grab_contours(contornos)

   # print(" ----- EXTRAER AGARRE ------ ")
   for c in contornos:
       M = cv2.moments(c)

       if M["m00"] == 0.0:
           continue
       area = cv2.contourArea(c)

       print("Area: " + str(area))
       if area > 2000.0:
           cX = int(M["m10"] / M["m00"])
           cY = int(M["m01"] / M["m00"])

           rows , cols = seg.shape[:2]
           vx,vy,x,y = cv2.fitLine(c, cv2.DIST_L2,0,0.01,0.01)

           v = (vx[0],vy[0])
           vP = (-vy[0],vx[0]) # Perpendicular al vector
           vComparar = (0,1)

           unit_vP = vP / np.linalg.norm(vP)
           unit_vComparar = vComparar / np.linalg.norm(vComparar)
           dot_product = np.dot(unit_vP, unit_vComparar)
```

```
            angle = np.arccos(dot_product)

            dirGiro = "derecha"
            if vP[0] != 0:
                pendiente = vP[1] / vP[0]
                    dirGiro = "derecha" if np.sign(pendiente) == -1 else
"izquierda"


            # Sacar ancho del agarre

            new_mask = np.zeros((seg.shape[0],seg.shape[1],3), np.uint8)

            c_idx = new_mask[...,1] == 255

            lines_idx = new_mask[...,0] == 255


            overlap = list(np.where(c_idx * lines_idx))
            if not overlap:

                overlap[0] = list(OrderedDict.fromkeys(overlap[0]))
                overlap[1] = list(OrderedDict.fromkeys(overlap[1]))
                            overlap  =  [(overlap[1][0],  overlap[0][0]),
(overlap[1][-1], overlap[0][-1])]


            wObj = distance.euclidean(overlap[0], overlap[1])


            if dirGiro == "derecha":
                angle = angle * -1 +3.14159265
                # dirGiro = "izquierda"

            else:
                angle = angle

            objetos[n] = [(cX, cY), wObj, angle, dirGiro]
            n += 1

        else:
            a=1

    return objetos
```

```
# Función para publicar los datos del agarre
def publishPose(pose, pub, rate, center, orientationZ, orientationX,
orientationY):
   pose.position.x = center[0]
   pose.position.y = center[1]
   pose.position.z = center[2]
   pose.orientation.x = orientationX
   pose.orientation.y = orientationY
   pose.orientation.z = orientationZ
   pub.publish(pose)
   rate.sleep()
```

8.3 Funciones de la cámara

```
import pyrealsense2 as rs
import numpy as np

class DepthCamera:
   def __init__(self):

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

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

        config.enable_stream(rs.stream.depth,640,480, rs.format.z16,30)
        config.enable_stream(rs.stream.color,640,480, rs.format.bgr8,30)

        # Start streaming
        profile = self.pipeline.start(config)
                                                                sensor       =
self.pipeline.get_active_profile().get_device().query_sensors()[1]
        sensor.set_option(rs.option.exposure, 150.000)

          # Getting the depth sensor's depth scale (see rs-align example
for explanation)
        depth_sensor = profile.get_device().first_depth_sensor()
        depth_scale = depth_sensor.get_depth_scale()
        print("Depth Scale is: " , depth_scale)

        # We will be removing the background of objects more than
        #  clipping_distance_in_meters meters away
        clipping_distance_in_meters = 0.5 #1 meter
        clipping_distance = clipping_distance_in_meters / depth_scale

        # Align
        align_to = rs.stream.color
        self.align = rs.align(align_to)


   def get_frame(self):
        frames = self.pipeline.wait_for_frames()

        # Align the depth frame to color frame
        aligned_frames = self.align.process(frames)

        # Get aligned frames
```

```
            aligned_depth_frame = aligned_frames.get_depth_frame() #
aligned_depth_frame is a 640x480 depth image
        color_frame = aligned_frames.get_color_frame()

        colorizer = rs.colorizer()
          colorizer.set_option(rs.option.visual_preset, 1) # 0=Dynamic,
1=Fixed, 2=Near, 3=Far
        colorizer.set_option(rs.option.min_distance, 0.01)
        colorizer.set_option(rs.option.max_distance, 1)

        depth_image = np.asanyarray(aligned_depth_frame.get_data())
        color_image = np.asanyarray(color_frame.get_data())

        if not aligned_depth_frame or not color_frame:
            return False, None, None

        return True, depth_image, color_image


    def get_3D_points(self, x_pix, y_pix, depth):
        frames = self.pipeline.wait_for_frames()
        aligned_frames = self.align.process(frames)
        aligned_depth_frame = aligned_frames.get_depth_frame()
        color_frame = aligned_frames.get_color_frame()

                                                    depth_intrin        =
aligned_depth_frame.profile.as_video_stream_profile().intrinsics

                                                  points_meters        =
rs.rs2_deproject_pixel_to_point(depth_intrin,[x_pix, y_pix],depth)
        return points_meters


    def release(self):
        self.pipeline.stop()
```

8.4 Script de control del robot

```
#!/usr/bin/env python
import sys
import rospy
import time

from tasks.pnp import PickAndPlace
from intera_core_msgs.msg import EndpointState
from toQuaternion import toEuler, toQuaternion
from geometry_msgs.msg import Pose
from math import pi
from std_msgs.msg import Bool


# Solo guarda posicion y orientacion inicial. Para que sea más seguro
cogen las #primeras 5 veces (solo se guarda la última de ellas)
def initialCallback(data):
   global firstPosition, posInit, rpy
   if firstPosition < 5:
       # Guarda la posicion inicial del endeffector
       posxInit = data.pose.position.x
       posyInit = data.pose.position.y
       poszInit = data.pose.position.z
       orientxInit = data.pose.orientation.x
       orientyInit = data.pose.orientation.y
       orientzInit = data.pose.orientation.z
       orientwInit = data.pose.orientation.w
       # Guarda en un vector la posicion inicial
       posInit = [posxInit, posyInit, poszInit]
       # Transforma los cuaterniones en angulos de euler
       rpy = toEuler(orientxInit, orientyInit, orientzInit,
orientwInit)
       firstPosition = firstPosition + 1


def moveCallback(data):
   global firstPosition, posInit, rpy, pub

   print("inicio moveCallback")
   pnp.run(data, rpy, posInit)
   time.sleep(0.5)
   pub.publish(True)


if __name__ == '__main__':
   global firstPosition, posInit, rpy, pub
   firstPosition = 0
   # Create publisher para publicar cuando ya se haya tomado una foto y
se publique otra
   pub = rospy.Publisher('/publish', Bool, queue_size=1)
```

```
    # Se inicializa pnp.
    pnp = PickAndPlace()
    # Se inicializa el robot, la pinza y se sitúa el robot en la
posición inicial
    pnp.initializeRobot()
    # Suscripcion al topic que ofrece información sobre el endeffector
    rospy.Subscriber("/robot/limb/right/endpoint_state", EndpointState,
initialCallback)
    # Subscribe to the topic that contains real point detection
information /real_coordinates
    rospy.Subscriber("/real_coordinates", Pose, moveCallback)
    rospy.spin()
```

8.5 Script del algoritmo de pick and place

```
#!/usr/bin/env python
import rospy
from intera_motion_msgs.msg import TrajectoryOptions
from geometry_msgs.msg import PoseStamped
from intera_interface import Limb
from hardware.robot import Robot
from .cinematica import tCamRobot
from toQuaternion import toQuaternion, toEuler
from math import pi
from intera_motion_interface import
(MotionTrajectory,MotionWaypoint,MotionWaypointOptions)

class PickAndPlace:
  def __init__(self):
     self.robot = Robot()
     self.limb = Limb('right')
     self.traj_options = TrajectoryOptions()
     self.traj_options.interpolation_type =
TrajectoryOptions.CARTESIAN
     self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
     self.wpt_opts = MotionWaypointOptions(max_linear_speed=10.0,
                                  max_linear_accel=10.0,
                                  max_rotational_speed= pi,
                                  max_rotational_accel= pi,
                                  max_joint_speed_ratio=1.0)
     self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
     self.endpoint_state = self.limb.tip_state('right_hand')
     self.pose = self.endpoint_state.pose

  #Inicializa el robot, la pinza y situa en la posicion inicial
  def initializeRobot(self):
     # Connect to robot
     self.robot.connect()

     # Calibrate gripper
     self.robot.calibrate_gripper()

     #Open Gripper
     self.robot.open_gripper()

     #Go to intial pose
     joint_angles = {'right_j6': -1.6581796875,
           'right_j5':-1.3590087890625,
           'right_j4': -2.1288427734375,
           'right_j3': 1.163734375,
           'right_j2': -1.0893388671875,
           'right_j1': -0.43348828125,
```

```
            'right_j0':  2.2774306640625}
        self.robot.move_joints(joint_angles,timeout=10)


    #Recibe los datos del robot y del topic de las coordenadas del objeto
respecto la camara
    def run(self, data, initialRpy, posInit):
        time = 2

        #Posicion para dejar el objeto
        posX = 0.35
        posY = 0.65

        #Distancias camara tcp
        distX = -0.03
        distY = 0.075
        distZ = 0.09

        #Posicion que tiene el robot al principio de todo
        xInitial = posInit[0]
        yInitial = posInit[1]
        zInitial = posInit[2]
        #Orientacion inicial
        xOrientation = self.robot.current_pose()["orientation"][0]
        yOrientation = self.robot.current_pose()["orientation"][1]
        zOrientation = self.robot.current_pose()["orientation"][2]
        wOrientation = self.robot.current_pose()["orientation"][3]
        initialR = toEuler(xOrientation, yOrientation, zOrientation,
wOrientation)
        # Rotacion en angulos de Euler de robot
        r = initialRpy[0]
        p = initialRpy[1]
        y = initialRpy[2]
        #Posicion del objeto respecto la camara
        pointXcam = data.position.x/1000
        pointYcam = data.position.y/1000
        pointZcam = data.position.z/1000
        #Calculo de la posicion del objeto respecto la base del robot
        poseGoalX = xInitial + distX + pointXcam
        poseGoalY = yInitial + distY - pointYcam
        poseGoalZ = zInitial + distZ - pointZcam - 0.015


        #Limita la z
        if poseGoalZ < (-0.050):
            poseGoalZ = -0.050
        print(poseGoalZ)

        #Calculo de la orientacion con la orientacion del objeto
        orientation_goal = toQuaternion(r, p, y + (data.orientation.z)
-1.57079632)    # (- Pi/2)
```

```
        #Mover a x-y del objeto
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(poseGoalX),4)
        self.pose.position.y = round(float(poseGoalY),4)
        self.pose.position.z = round(float(zInitial),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)

        #Bajar en z
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(poseGoalX),4)
        self.pose.position.y = round(float(poseGoalY),4)
        self.pose.position.z = round(float(poseGoalZ),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)

        #Se cierra la pinza
        self.robot.close_gripper()

        #Subir en z
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(poseGoalX),4)
        self.pose.position.y = round(float(poseGoalY),4)
        self.pose.position.z = round(float(poseGoalZ + 0.30),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
```

```
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)

        #Transladar a x-y para dejar el objeto
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(posX),4)
        self.pose.position.y = round(float(posY),4)
        self.pose.position.z = round(float(poseGoalZ + 0.30),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)

        #Bajar en z
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(posX),4)
        self.pose.position.y = round(float(posY),4)
        self.pose.position.z = round(float(poseGoalZ + 0.10),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)

        #Abrir pinza
        self.robot.open_gripper()

        #Subir en z
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
```

```
        self.pose.position.x = round(float(posX),4)
        self.pose.position.y = round(float(posY),4)
        self.pose.position.z = round(float(poseGoalZ + 0.3),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(timeout = time - 1)


        #Calculo de la orientacion sin orientacion del objeto
        orientation_goal = toQuaternion(r, p, y)
        self.waypoint = MotionWaypoint(options = self.wpt_opts.to_msg(),
limb = self.limb)
        self.traj = MotionTrajectory(trajectory_options =
self.traj_options, limb = self.limb)
        self.pose.position.x = round(float(xInitial),4)
        self.pose.position.y = round(float(yInitial),4)
        self.pose.position.z = round(float(zInitial),4)
        self.pose.orientation.x = round(orientation_goal[0],10)
        self.pose.orientation.y = round(orientation_goal[1],10)
        self.pose.orientation.z = round(orientation_goal[2],10)
        self.pose.orientation.w = round(orientation_goal[3],10)
        self.poseStamped = PoseStamped()
        self.poseStamped.pose = self.pose
        self.waypoint.set_cartesian_pose(self.poseStamped, 'right_hand')
        self.traj.append_waypoint(self.waypoint.to_msg())
        result = self.traj.send_trajectory(wait_for_result=True, timeout =
None)
```